Dynamic Programing and Heuristics

Module Overview

In the previous two modules you learned more about scoring methods, patterns, and PSSMs associated with alignments, but so far you haven't seen how this translates to the programs that will almost certainly be frequently used tools in your bioinformatics toolkit – BLAST and FASTA. BLAST and FASTA are alignment programs that use heuristics. To understand alignment heuristics, you first need to understand the dynamic programming methods on which they are based. The first dynamic programming algorithm applied to sequence analysis was Needleman-Wunsch, used for global sequence alignments. Smith-Waterman modified and extended this for use in local alignments. Needleman-Wunsch and Smith-Waterman produce mathematically optimal alignments, but they are too computationally intensive for use in genome-scale alignments. Heuristics make alignments practical for the vast collections of sequences available at NCBI and through Next-Generation Sequencing (NGS).

Learning Objectives

- Align two short sequences using a dynamic programming algorithm.
- Build a BLAST database from a FASTA file
- Run BLAST and FASTA alignments from the command line
- Parse BLAST and FASTA tabular output and convert to GFF3 format

Required Reading

Understanding Bioinformatics chapters 5 and 6 were required reading in previous modules, but you may want to review them again before reading the Altschul paper assigned for this module. It would also be better to read the text within this module to get an overview before getting into the technical details of the paper. If you are new to reading original scientific papers they can seem overwhelming at first, but the ability to read, understand, and write papers like this is an important skill for a bioinformatician.

Paper	**Section(s) to Concentrate On**
Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res, 25(17), 3389-3402.	**Refinement of the basic algorithm: The two-hit method** **Sequence weights** **BRCT proteins**

Optional Reading

This module covers some of the techniques used in CRISPR design. For background on CRISPR design read the papers listed below. This reading is optional and there are no quiz

questions on these papers. The bioinformatics work for the PNAS paper was a co-op project, so you may find these papers useful to get a feel for the type of work done on co-op.

- The CRISPR Craze
- Optimized gene editing technology for Drosophila melanogaster using germ line-specific Cas9

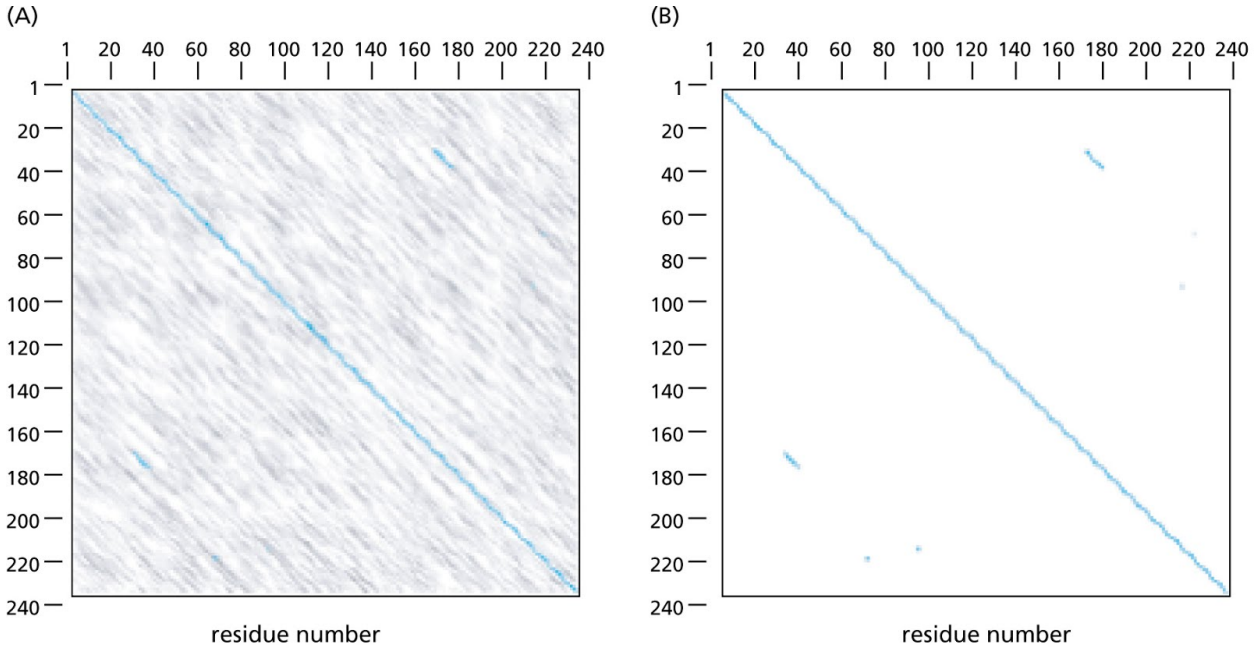
YouTube Playlist

Sequence Analysis

Sequence analysis methods can be categorized as intrinsic or extrinsic, and exact or heuristic. Intrinsic versus extrinsic differentiates comparison with self from comparison to others. Exact versus heuristic differentiates algorithms that guarantee maximization of a score according to a given scoring scheme from those that gain speed using “rules of thumb” at the risk of missing an optimal alignment. The risk of missing an optimal alignment may sound like a high price to pay for speed, but this module will show you examples of alignments for which the determination of an optimal alignment would be computationally intractable.

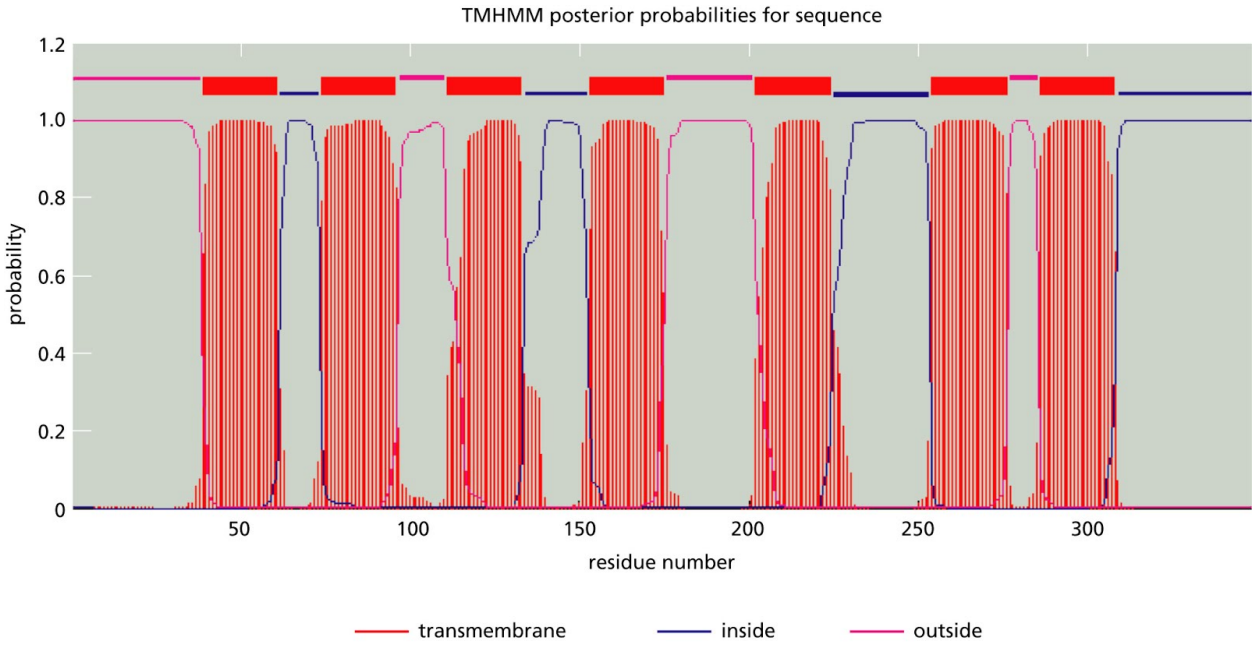
Intrinsic Sequence Analysis

Intrinsic Sequence Analysis is the evaluation of sequence properties without explicitly referring to other sequences using a derived formula, sequence or value. Examples include calculating percent G+C values (DNA), or comparing the sequence against itself for the presence of direct or inverted repeats. The self-alignment of the BRCA2 protein shown in the dot plot below is an example of intrinsic sequence analysis.



Understanding Bioinformatics Figure 4.3

The transmembrane prediction shown below is another example of intrinsic sequence analysis.



Sequence	outside	1	38
Sequence	TMhelix	39	61
Sequence	inside	62	73
Sequence	TMhelix	74	96
Sequence	outside	97	110
Sequence	TMhelix	111	133
Sequence	inside	134	152
Sequence	TMhelix	153	175
Sequence	outside	176	201
Sequence	TMhelix	202	224
Sequence	inside	225	253
Sequence	TMhelix	254	276
Sequence	outside	277	285
Sequence	TMhelix	286	308
Sequence	inside	309	348

Understanding Bioinformatics Figure 11.37

Extrinsic Sequence Analysis

Extrinsic sequence analysis is the evaluation of sequence properties by explicitly comparing them to other sequences or sets of sequences. Examples of extrinsic analysis include sequence comparisons using pairwise dot plots, pairwise dynamic (Needleman-Wunsch or Smith-Waterman), pairwise heuristic (BLAST, BLAT, FASTA), or multiple sequence alignments to detect homology, insertions or deletions, as well as statistical or phylogenetic analysis. A common mistake in sequence analysis is to use only one of these approaches.

Combining intrinsic and extrinsic computational approaches can give results neither alone can approach.

Dynamic Programming versus Heuristics

The two main categories of sequence alignment methods are exact methods, which use dynamic programming algorithms and approximate methods, which use heuristic algorithms. Dynamic programming algorithms guarantee a mathematically optimal result with a given scoring scheme. Dynamic programming algorithms include:

- Needleman-Wunsch (global)
- Smith-Waterman (local)

These algorithms are often used for pairwise alignments, but they are rarely used for multiple alignments because they become computationally intractable with increasing number and/or length of sequences.

Heuristic algorithms make some assumptions that hold true most, but not all of the time. Programs that use heuristic pairwise alignment algorithms include:

- BLAST
- FASTA

All widely used multiple alignment programs use heuristic algorithms, and they include:

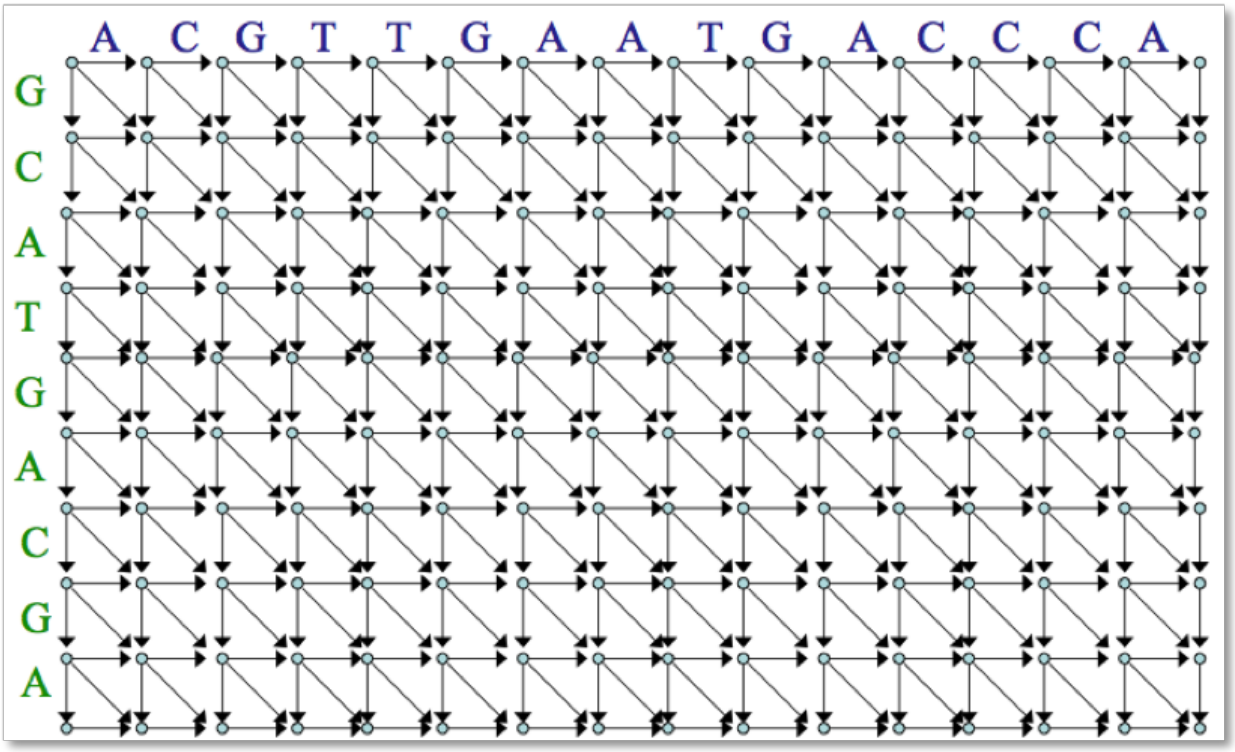
- ClustalW
- Tcoffee
- Muscle
- MAFFT

Optimal alignments in the mathematical sense provide the best or highest-scoring alignment for a given set of scoring functions. Optimal alignments in the biological sense provide an alignment in which each aligned sequence residue descended from the same ancestral residue, and each aligned sequence residue plays the same functional role for the two proteins. Finding mathematically optimal alignments is straightforward in concept (whether computationally feasible or not), but finding biologically optimal alignments (homology) is not.

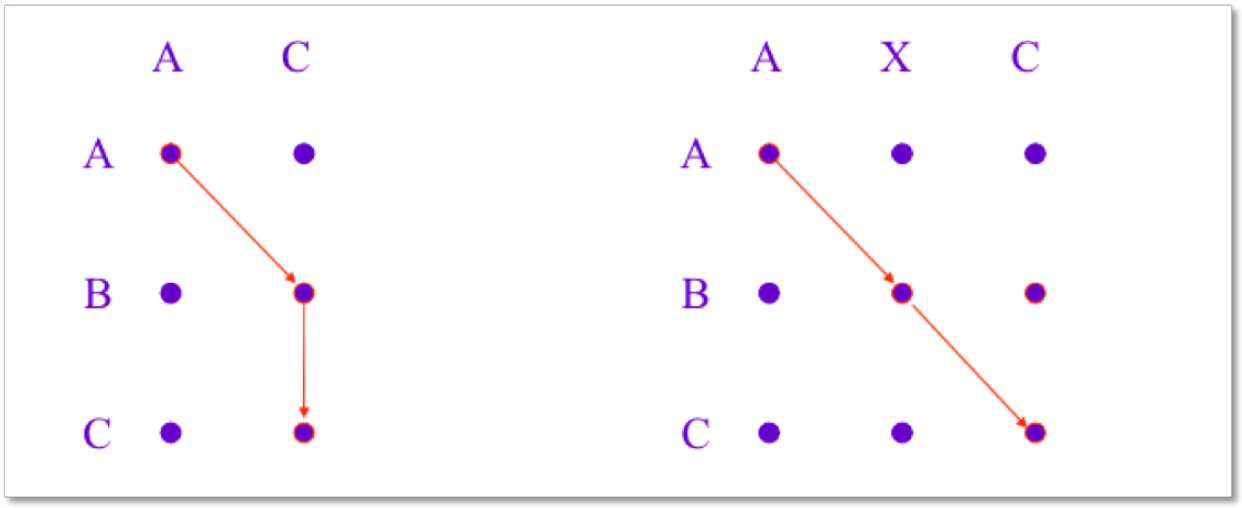
Graph Theory and Terminology

Sequence alignment methods are often described using graph terminology. A sequence graph represents residues as edges between nodes. Nodes are also called vertices. Such a graph can represent a single sequence, or a multi-dimensional comparison between two sequences. In a multi-dimensional graph, paths through the graph correspond to alignments of the sequences, with each edge on the path corresponding to a column of the alignment. The image below shows the alignment of two sequences displayed as a graph. Without a scoring mechanism you could just connect any combination of vertices and call it an

alignment. The challenge is to find the “best” path through this graph.



Diagonal edges correspond to two aligned residues. Horizontal and vertical edges correspond to a residue in one sequence and a gap in the other (indels). The image below shows indels (left) and substitutions (right) as graphs.



Edge Weights on Graphs

Edge weights correspond to scores for an aligned residue or gap. A simple version would be 1 for a match, 0 otherwise. The weight of a path is the sum of weights for each edge on the

path. The highest weight path corresponds to the highest scoring alignment for that scoring system. Weights are assigned using a substitution score matrix such as the BLOSUM62 matrix.

Over the course of evolution, some positions have undergone base or amino acid substitutions as well as insertion/deletion (indel) events. Any measurement of distance or similarity must therefore be done with respect to the best possible alignment between two sequences. Because indel events are rare compared to base substitutions, it makes sense to penalize gaps more heavily than mismatches when calculating scores.

Sequence Comparison Methods

To find the “best” alignment between a query sequence and a target sequence, we need a method for scoring alignments, and an algorithm for finding the alignment with the best score. The alignment score is calculated using a substitution matrix and gap penalties. The number of possible alignments may be astronomical. For example when two sequences 300 residues long each are compared there are more possible alignments (10^{88}) than the number of elementary particles in the universe (~10^{80})[1]. Fortunately there are computer algorithms for finding the optimal alignment between two sequences that do not require an exhaustive search of all the possibilities.

Dynamic Programming

Dynamic programming (DP) algorithms are a general class of algorithms typically applied to optimization problems. For DP to be applicable, an optimization problem must have two key ingredients:

- Optimal substructure - an optimal solution to the problem contains within it optimal solutions to subproblems
- Overlapping subproblems - the pieces of the larger problem have a sequential dependency

For a more detailed explanation of optimal substructure, read http://en.wikipedia.org/wiki/Optimal_substructure.

Two important dynamic programming algorithms are Needleman-Wunsch (NW), which is used for global alignments and Smith-Waterman (SW), which is used for local alignments. Global alignments attempt to align every residue in the sequences, and they are most useful when the sequences are similar in size. Local alignments find an alignment for parts of the two strings, and they are most useful for dissimilar sequences that share regions of similarity or contain similar motifs.

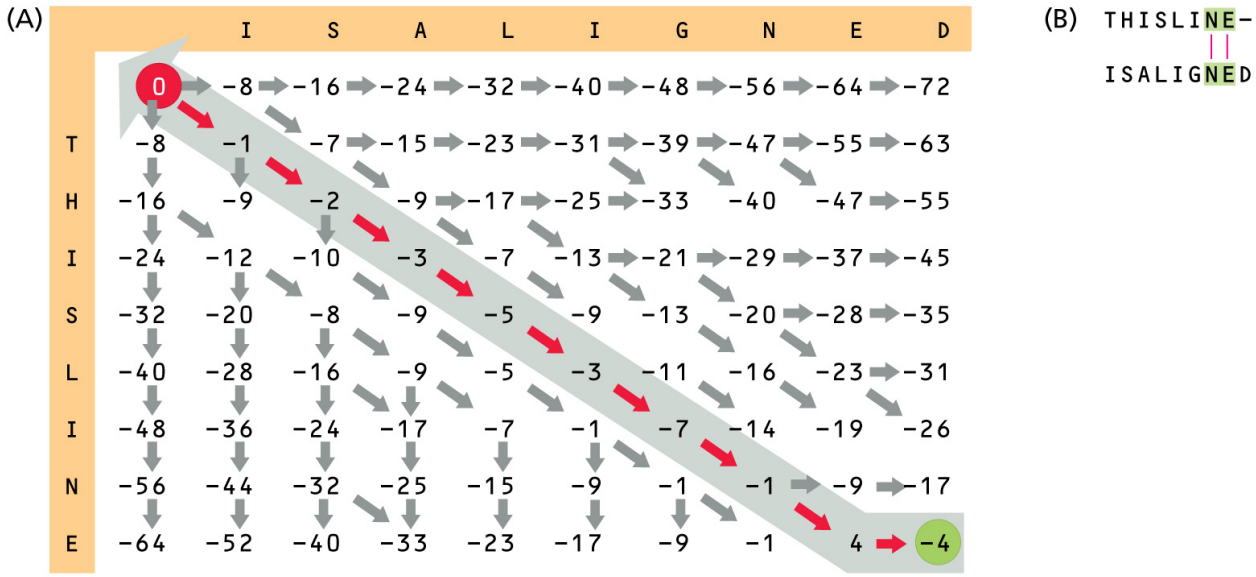
Needleman-Wunsch

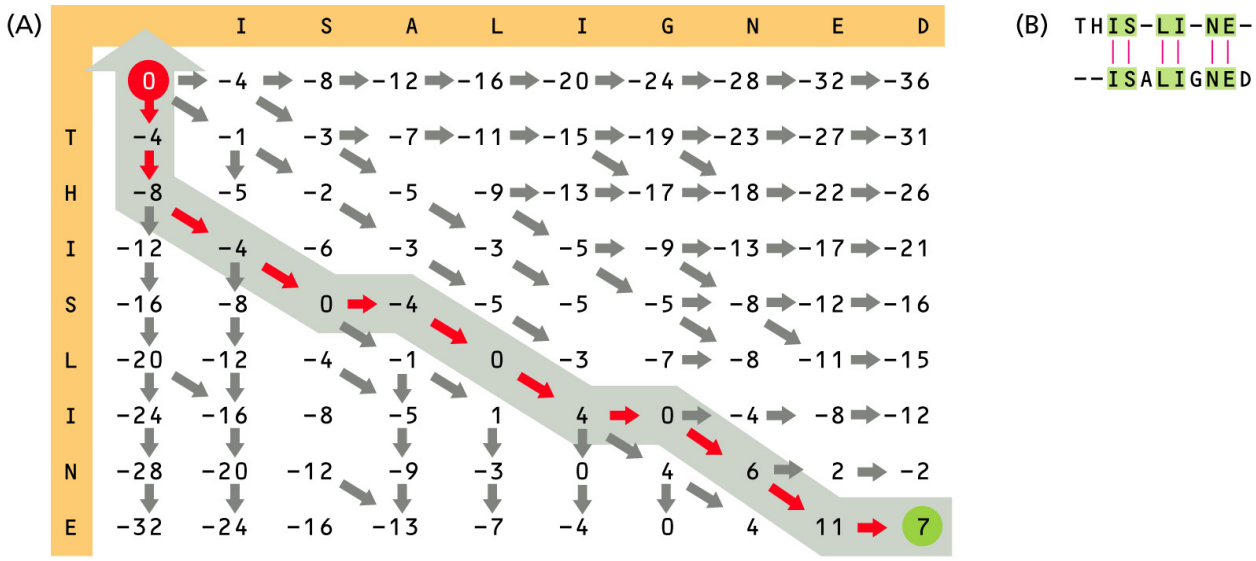
The basic steps for a Needleman-Wunsch dynamic programming alignment:

- Start at the top left corner on the graph

- Proceed across and down the graph, adding up match scores for every path and recording in a 2-D array
- Choose the path that gives the best score at the bottom row or right edge
- Trace back through the highest scoring path to obtain the alignment

The images below show all paths scored, as well as the trace-back through the highest-scoring path for two different scoring methods. Notice the relationship between horizontal and vertical arrows (indels) and the corresponding text alignment to the right of each graph. You should be able to write out the text alignment for a short sequence given a graph representation, and vice versa.





Understanding Bioinformatics Figures 5.09 and 5.11

Needleman-Wunsch Alignment Video

This video steps through a Needleman-Wunsch alignment.

Smith-Waterman

The idea behind the Smith-Waterman algorithm is to modify Needleman-Wunsch to ignore badly aligning regions. As with NW, Smith-Waterman starts with:

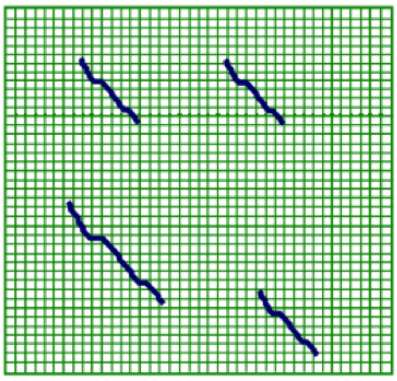
- 2-dimensional matrix with one sequence along the top and the other sequence down the left side
- All possible alignments are represented by the paths through the matrix
- A diagonal step is an alignment between the query and the subject sequences at that position
- A vertical step is a gap in the query sequence
- A horizontal step is a gap in the subject sequence

The difference is in the calculations used for initialization and scoring of the matrix:

Modifications to Needleman-Wunsch:

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration: $F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) + g \\ F(i, j-1) + g \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$



Which Method to Use

It's important to know which type of alignment – global or local – to use in a given situation. Use a global alignment if:

- You expect, based on some biological information, that your sequences will match over the entire length

- Your sequences are of similar length

Use local alignment if:
- You expect that only certain parts of two sequences will match, as in the case of conserved segments that can be found in many different proteins
- Your sequences are very different in length
- You want to search a sequence database

Dynamic Programming Advantages and Disadvantages

The advantage of dynamic programming is that it guarantees the optimal (very best or highest-scoring) alignment for a given set of scoring functions in a mathematical sense. The disadvantages are that dynamic programming is slow due to the very large number of computational steps and the computer memory requirement increases as the square of the sequence lengths.

Consider a search for best aligning sequences in huge datasets. Would a dynamic programming algorithm work? Let's go through some numbers to see. For two sequences of lengths M and N, the sequence alignment graph has M * N nodes and 3 * M * N edges. The search time complexity is of the order M * N. The search space complexity to reconstruct the highest-scoring alignment is of the order M * N.

Using a 1 Kb DNA sequence to search the GenBank nr database (~132 billion nucleotides in 2011) would require calculation of about 300 trillion edge weights (3×10^{14}). Even running on fast clusters, this is far too slow for general use.

Heuristic Algorithms

Heuristic algorithms solve a problem by using rules of thumb to reach a solution. The solution is not guaranteed to be an optimal solution, but it's generally arrived at far faster than using an optimal solution approach such as dynamic programming.

In sequence alignments, commonly used heuristic approaches include the FASTA algorithm, developed by William Pearson in 1988, the BLAST algorithm, developed by Stephen Altschul in 1990, and the BLAT algorithm, developed by Jim Kent in 2002.
The most widely known and used of these is the BLAST algorithm.

FASTA and BLAST

FASTA and BLAST are word-based sequence alignment methods. Both methods are fast enough to support searching for alignments of query sequences against entire databases. The target dataset is pre-indexed to indicate the positions in all dataset sequences that match each search word above some scoring threshold using a global score matrix such as BLOSUM62.

In dynamic programming a lot of time is spent calculating the whole matrix. In most cases this isn't necessary, because much of the matrix is far away from the main diagonal. Heuristic

methods only calculate diagonals where the score is above a certain minimum. Note that by not calculating the entire matrix, you risk missing an optimal alignment.

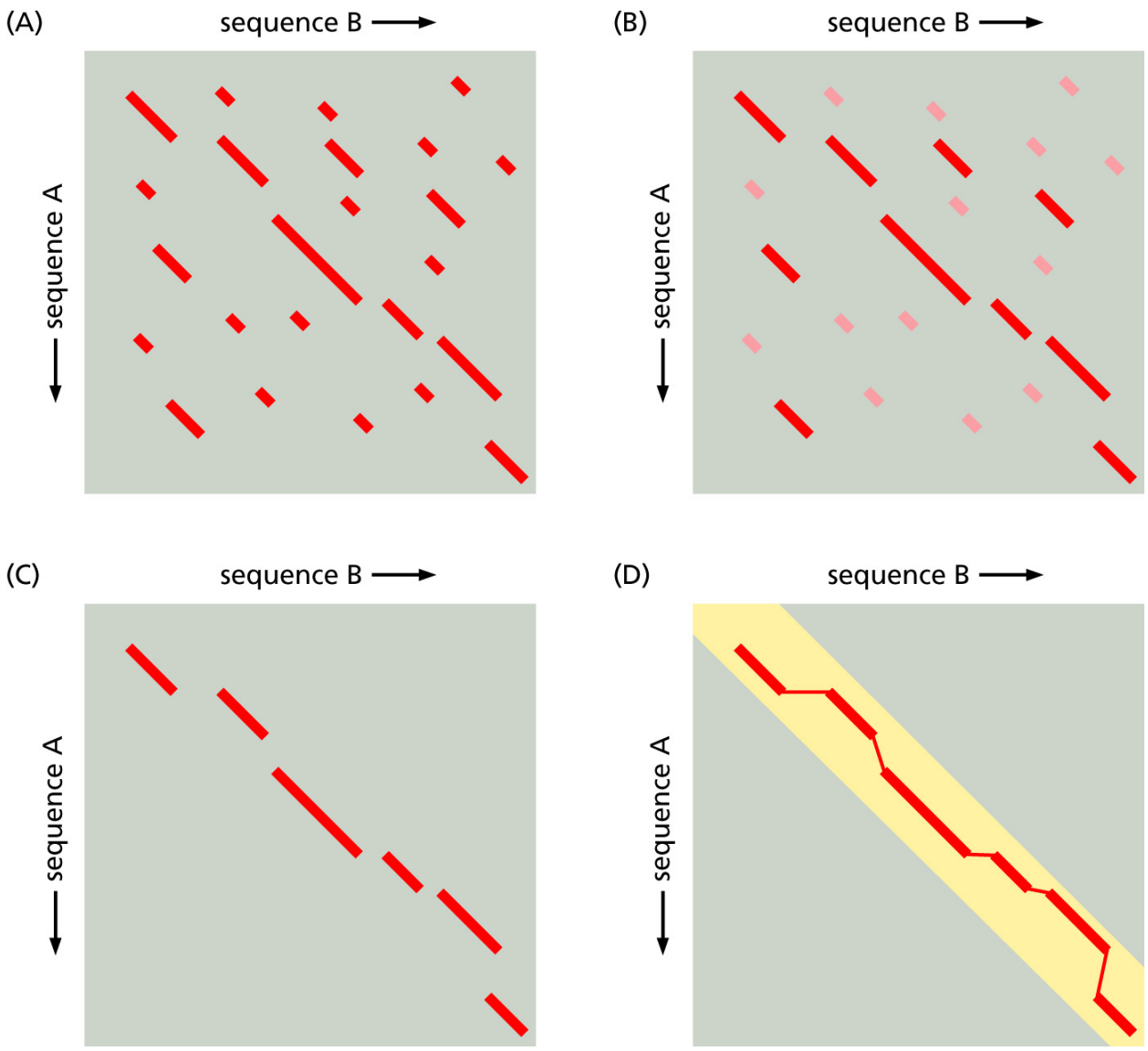
Short Exact Matches

Smith-Waterman is exact but slow. The best improvement on S-W involves a list of short sequences that match exactly between the query and subject. The improved process:

- Make a list of the positions of all k-mers or k-tuples in both sequences where k = length. Length k is set at 2 for protein and 6 for DNA in the FASTA program and 3 for protein, 11 for DNA in BLAST
- Find all matching k-mers between query and subject, and combine them to form regions of exact un-gapped matches

FASTA

There are four steps in the FASTA algorithm:
A. Find the best un-gapped perfect matching alignments
B. Rescore the highest-scoring alignments using the PAM250 Matrix
C. Join together some of these high-scoring ungapped alignments using some gaps
D. use the Smith-Waterman dynamic programming method to extend the alignment



Understanding Bioinformatics Figure 5.22

BLAST

Of all the sequence alignment algorithms, BLAST is the most frequently used, with about 44,000 citations. BLAST is typically used to compare one query nucleotide/protein sequence against a database of sequences to uncover similarities and sequence matches. The success and popularity of BLAST stems from combination of:

- Speed
- Sensitivity
- Statistical assessment of the results

BLAST is a heuristic method to find the high-scoring locally optimal alignments between a query sequence and a database. The algorithm and family of programs rely on the statistics of

gapped and ungapped sequence alignments. The statistics allow the probability of obtaining an alignment with a particular score to be estimated. BLAST is unlikely to be as sensitive as a full dynamic programming algorithm, but the underlying statistics provide a direct estimate of the significance of any match found.

There's no comparison in terms of speed when compared to dynamic programming. There are several BLAST programs, and the BLAST program to use in a given situation depends on the characteristics of the sequences being compared. The table below shows BLAST programs that are designed for specific types of query and subject sequences.

Program	**Query**	**Subject**
BLASTn	Nucleotide	Nucleotide
BLASTp	Protein	Protein
BLASTx	NT translated in all reading frames	Protein
tBLASTn	Protein	NT database dynamically translated in all reading frames
tBLASTx	NT translated 6 frames	NT translated 6 frames

The table below shows two specialized BLAST programs

Program	**Description**
BLAST2seq	Compares two protein or two nucleotide sequences
PSI-BLAST	Compares a protein sequence to a protein database and performs the comparison in an iterative fashion in order to detect homologs that are evolutionarily distant. PSI-BLAST Uses a dynamically calculated scoring matrix from the actual BLAST search

BLAST Steps

- Apply complexity filtering to mask segments of the query sequence with low compositional complexity. Filtering can eliminate statistically significant, but biologically uninteresting hits from the BLAST output.
- Find hits based upon a lookup table
- Extend those hits

Using FASTA and BLAST

Now that you have some understanding of how BLAST and FASTA work, you need to learn how to use them yourself. Both BLAST and FASTA have web-based versions. The web-based versions can be useful for occasional one-off searches, but for most bioinformatics work you'll need to be able to use the command-line interface to build your own alignment databases and do your own alignments.

BLAST and FASTA command-line applications are available for your use on the server.

Both FASTA and BLAST can also be installed on your computer. FASTA download and installation instruction are available at:

http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml

BLAST download and installation instructions are available at:
http://www.ncbi.nlm.nih.gov/books/NBK52640/ (Linux/Mac)
http://www.ncbi.nlm.nih.gov/books/NBK52637/ (Windows)

You aren't required to install these, this information is just provided for those who prefer to run it locally.

Using FASTA

To perform a command-line alignment using FASTA, you need two FASTA files – one containing the query sequences, and the other containing the subject sequences. For example to align a fasta file containing a collection of gene knockout oligonucleotides to all Drosophila chromosomes, the command would be:

```
fasta36 /scratch/Drosophila/oligos.fasta
/scratch/Drosophila/dmel-all-chromosome-r6.02.fasta
1>oligos_aligned.txt 2>oligos_aligned.err&
```

In this command,
`1>oligos_aligned.txt` means redirect STDOUT to oligos_aligned.txt
`2>oligos_aligned.err` means redirect STDERR to oligos_aligned.err
`&` means run the program in the background

You should use `1>`, `2>` and `&` like this for any program that will run for more than a few minutes. By running programs in this way, you can continue working from the command line while your program runs in the background. It will continue running even if you logout. You can see if the program is still running, and how much CPU and memory it's using with the `top` command. Type `q` to exit top.

The first 40 lines of output viewed with head would look like this:

```
head -n40 oligos_aligned.txt
# fasta36 /scratch/Drosophila/oligos.fasta
/scratch/Drosophila/dmel-all-chromosome-r6.02.fasta
FASTA searches a protein or DNA sequence data bank
 version 36.3.5e Nov, 2012(preload8)
Please cite:
 W.R. Pearson & D.J. Lipman PNAS (1988) 85:2444-2448

Query: /scratch/Drosophila/oligos.fasta
  1>>>CG18721shmi-1 - 21 nt
Library: /scratch/Drosophila/dmel-all-chromosome-r6.02.fasta
  143725995 residues in  1870 sequences

Statistics:  Expectation_n fit: rho(ln(x))= 4.5727+/-0.00044; mu= 10.8954+/- 0.038
 mean_var=18.7603+/- 5.947, 0's: 0 Z-trim(99.2): 1  B-trim: 0 in 0/30
 Lambda= 0.296111
```

```
 statistics sampled from 1899 (1900) to 1899 sequences
Algorithm: FASTA (3.7 Nov 2010) [optimized]
Parameters: +5/-4 matrix (5:-4), open/ext: -12/-4
 ktup: 2, E-join: 0.25 (0.93), E-opt: 0.05 (0.513), width:  16
 Scan time: 10.120

The best scores are:                                      opt bits E(2784)
3R type=golden_path_region; loc=3R:1..32079331 (32079331) [f]  105 40.3  0.0022

>>3R type=golden_path_region; loc=3R:1..32079331; ID=3R;  (32079331 nt)
 initn: 105 init1: 105 opt: 105  Z-score: 146.6  bits: 40.3 E(2784): 0.0022
banded Smith-Waterman score: 105; 100.0% identity (100.0% similar) in 21 nt overlap
(1-21:4644365-4644385)

                                          10        20
CG1872                                CTGAGCGACGACATTGATGTA
                                      :::::::::::::::::::::
3R     CCTGAAGAACTGCAGGATCATGACGAGGATCTGAGCGACGACATTGATGTAGACAGCGAT
      4644340   4644350   4644360   4644370   4644380   4644390

3R     ATTGAAGATACCCCGGAGCTCAGCGATTAGTTGAGGACCCATTTCATTTTATTTACAAAT
      4644400   4644410   4644420   4644430   4644440   4644450


  2>>>CG18721shmi-2 - 21 nt
Library: /scratch/Drosophila/dmel-all-chromosome-r6.02.fasta
  143725995 residues in  1870 sequences
```

To run the alignment and produce output in an easy-to-parse tabular format add the -m8 option:

```
fasta36 -m8 /scratch/Drosophila/oligos.fasta
/scratch/Drosophila/dmel-all-chromosome-r6.02.fasta 1>
oligos_aligned.tsv 2>oligos_aligned.err&
```

Try this and view oligos_aligned.tsv using the `less` command.

Using BLAST

To perform a command-line alignment using BLAST, you need a BLAST database containing subject sequences, and a FASTA file containing query sequences. Using the FASTA files from the previous example, the command to build a BLAST database from the FASTA file would be:

```
makeblastdb -in /scratch/Drosophila/dmel-all-chromosome-r6.02.fasta
-dbtype nucl -parse_seqids -title Drosophila -out Drosophila

Building a new DB, current time: 11/23/2014 06:40:25
New DB name:   Drosophila
New DB title:  Drosophila
Sequence type: Nucleotide
Keep Linkouts: T
Keep MBits: T
```

```
Maximum file size: 1000000000B
Adding sequences from FASTA; added 1870 sequences in 5.68065 seconds.
```

To see the files created by `makeblastdb`, run the `ls` command:
`ls -ltrh`

To run the alignment:
`blastn -task blastn -db Drosophila -query /scratch/Drosophila/oligos.fasta 1>oligo_blast.txt 2>oligo_blast.err&`

To view the first 40 lines of output:
`head -n40 oligo_blast.txt`

To run the alignment and produce tabular output add `-outfmt 6` and change the output filename to `oligo_blast.tsv`:
`blastn -task blastn -db Drosophila -query /scratch/Drosophila/oligos.fasta` **`-outfmt 6`** `1>`**`oligo_blast.tsv`** `2>oligo_blast.err&`

Use `top` to see when the program finishes, and use `less` to view the output file.

Parsing BLAST Output

There will be more about what all these columns mean in the next module, but for now you need to know how to parse them using Perl. The Perl program shown below runs BLAST and parses the output in a single step.

```
#!/usr/bin/perl
use warnings;
use strict;
#Author: C Roesel
#Creation Date: November 15 2013

#This program executes BLAST and parses the output in a single step.
#This can be useful when you will need to convert the BLAST output to
#another format. If for example you expect to produce 20G BLAST
#output, then reformat that output to another 20G file in a different
#format, you avoid having to use 20G disk space for a temporary file.

blastOligos();

sub blastOligos {
   #Put my BLAST command and all the params in an array. This could be created as
   #a single string, but an array makes it easier to see the individual
   #parameters.
   my @commandAndParams = (
       'blastn', '-task blastn',
       '-db Drosophila',
       '-query /scratch/Drosophila/oligos.fasta',
       '-outfmt 6'
   );
   #Print the BLAST command for debugging purposes.
   print "@commandAndParams\n";
   #Run the BLAST command and get the output as a filehandle named BLAST.
```

```
   open( BLAST, "@commandAndParams |" );
   #Process the BLAST output line-by-line using the filehandle BLAST.
   while (<BLAST>) {
       #Get rid of end-of-line characters.
       chomp;
       #Assign the line of output from the default variable $_ to the meaningfully
       #named variable blastOutputLine.
       my $blastOutputLine = $_;
       #If the output line isn't a comment line
       if ( $blastOutputLine !~ /^#/ ) {
           #Split the output line using the tab as separator.
           my @blastColumns = split("\t",$blastOutputLine);
           #Assign the column positions to meaningfully named variables.
           my $queryId      = $blastColumns[0];
           my $chrom        = $blastColumns[1];
           my $identity     = $blastColumns[2];
           my $length       = $blastColumns[3];
           my $mismatches   = $blastColumns[4];
           my $gaps         = $blastColumns[5];
           my $qStart       = $blastColumns[6];
           my $qEnd         = $blastColumns[7];
           my $sStart       = $blastColumns[8];
           my $sEnd         = $blastColumns[9];
           #Print one of the columns to make sure I'm parsing correctly.
           print $sStart, "\n";
           #TODO
           #Do something with the data here.
       }
   }
}
```

This video steps through the code line-by-line.

Converting BLAST Output to GFF3 Format

A popular data format for sequence alignments is GFF3. This format is used by several graphical genome viewers like JBrowse, Gbrowse, and the Integrated Genome Viewer. The program shown below runs BLAST and converts the output to GFF3 format in a single step. Part of your final project will be based on the conversion of BLAST output to GFF3 format.

```
#!/usr/bin/perl
use warnings;
use strict;

#Author: C Roesel
#Creation Date: November 15 2013

#This program executes BLAST and parses the output in a single step.
#This can be useful when you will need to convert the BLAST output to
#another format. If for example you expect to produce 20G BLAST
#output, then reformat that output to another 20G file in a different
#format, you avoid having to use 20G disk space for a temporary file.

unless ( open( GFF3, ">", 'oligos.gff3' ) ) {
   die $!;
}

blastOligos();
```

```
sub blastOligos {

 #Put my BLAST command and all the params in an array. This could be created as
 #a single string, but an array makes it easier to see the individual
 #parameters.
   my @commandAndParams = (
       'blastn', '-task blastn',
       '-db Drosophila',
       '-query /scratch/Drosophila/oligos.fasta',
       '-outfmt 6'
   );

   #Print the BLAST command for debugging purposes.
   print "@commandAndParams\n";

   #Run the BLAST command and get the output as a filehandle named BLAST.
   open( BLAST, "@commandAndParams |" );

   #Process the BLAST output line-by-line using the filehandle BLAST.
   while (<BLAST>) {

       #Get rid of end-of-line characters.
       chomp;

    #Assign the line of output from the default variable $_ to the meaningfully
    #named variable blastOutputLine.
       my $blastOutputLine = $_;
       processBlastOutputLine($blastOutputLine);
   }
}

sub processBlastOutputLine {
   my ($blastOutputLine) = @_;

   #If the output line isn't a comment line
   if ( $blastOutputLine !~ /^#/ ) {

       #Split the output line using the tab as separator.
       my @blastColumns = split( "\t", $blastOutputLine );

       #Assign the column positions to meaningfully named variables.
       my (
           $queryId, $chrom, $identity, $length, $mismatches,
           $gaps,    $qStar, $qEnd,     $start,  $end
       ) = @blastColumns;
       my $strand   = '+';
       my $gffStart = 0;
       my $gffEnd   = 0;
       if ( $start > $end ) {
           $strand   = '-';
           $gffStart = int $end;
           $gffEnd   = int $start;
       }
       else {
           $gffStart = int $start;
           $gffEnd   = int $end;
       }
       my @rowArray;
       @rowArray = (
           $chrom, ".", 'OLIGO', $gffStart, $gffEnd, ".", $strand, ".",
```

```
            "Name=$queryId;Note=Some info on this oligo"
        );
        local $, = "\t";
        print GFF3 @rowArray, "\n";
    }
}
```

This video steps through the code line-by-line.

Lab

The purpose of this lab is to build upon your previous program, adding some of the parts you'll need to complete your final project. You'll get more details on the specifics of the final project later, but the direction this is heading is to complete a subset of the programming involved in designing CRISPRs. CRISPRs provide a new way to edit specific locations within a genome.

1. Make a copy of your k-mer counting program from the previous lab. Call it `uniqueKmersEndingGG.pl`.
2. Open a filehandle for writing output to `uniqueKmersEndingGG.fasta`
3. Change the window length from 15 to 23.
4. Rather than printing out the k-mers and associated counts, go through your hash of k-mers and only print out the first 1000 that occur once and end with GG. The reason for limiting the output to the first 1000 is to keep the time required to run BLAST manageable.
5. Put a FASTA header before each k-mer, assigning a number to each based on their order in the hash. If, for example, you find 3 k-mers that only occur once, print them to a file like this:
   ```
   >1
   ATGCATGCATGATTCAGTCAAGG
   >2
   ATTCATGCATGATTCAGTCACGG
   >3
   ATGCATGCATGATTAAGTCATGG
   ```
6. Create a BLAST database using `makeblastdb` with `dmel-2L-chromosome-r5.54.fasta` as the input and `Drosophila2L` as the output and title. Use the `makeblastdb -help` command to determine the command and options to use.
7. BLAST `uniqueKmersEndingGG.fasta` against the BLAST DB `Drosophila2L` using a Perl script that converts the BLAST output lines with 100% identity to GFF3 format. Write the lines with less than 100% identity to `offTarget.txt`, without changing the format of the BLAST output for these lines. Write the GFF3 output to `crispr.gff3`.

---

[1] Gribskov, M. R., & Devereux, J. (1991). Sequence analysis primer. New York: New York : Stockton Press. p.