CS51 Final Report
BioStat51: A Bioinformatics Command-Line Tool

N. Katz & R. Burgess
May 2, 2014

Overview

This is a final report out for **BioStat51**, bioinformatics tool designed to align, compare, and convert sequences of DNA, RNA, and amino acids. This project was inspired by industry-standard bioinformatics tools like the BLAST (Basic Local Alignment Search Tool) on PubMed, which among other things, is used by biologists to compare sequences isolated in laboratories to a known database of sequences.

Our aim was to produce an easy-to-use tool on a smaller scale that, given a query sequence, can run a comparison against a database in a local environment using some historically effective algorithms. This is useful for determining whether two or more sequences evolved from the same sequence.

As a "cool extension," we also added functionality wherein the tool can convert DNA sequences to complementary DNA sequences, DNA to RNA sequences, and RNA to amino acid sequences (which effectively form proteins in biological systems.)

This tool has two primary modes: sequence alignment, and sequence conversion. Our video will cover the algorithms used in the sequence alignment mode, the parser, and the output modules. View this bonus video to see a demonstration of how RNA can be translated into an amino acid sequence using mode 2.

Mode 1: DNA, RNA, and Protein Alignment

To perform an alignment means the following:

1) line up a pair of DNA, RNA, or amino acid sequences to identify similar and/or dissimilar regions or subunits; and

2) assign a score to the two sequences that indicates their degree of relatedness.

Matrices & Algorithms

The reason why aligning DNA, RNA, and protein sequences is more challenging than simply aligning two strings is due to mutation. Homologous sequences, defined as those that over generations have arisen from an ancestral sequence, usually have accumulated a number of mutations, which are insertions, deletions, inversions, or substitutions in their sequence of subunits. Because of this, an algorithm needs to determine where the starting point of one sequence may line up with the starting point of its homolog, as well as how similarities can be detected in spite of the insertions, deletions, or substitutions in one or both sequences.

Matrices

Because there are bound to be mismatched bases resulting from mutation, these alignments function by assigning scores to every possible match and mismatch. For each type of molecule (DNA, RNA, and protein), these scores are mapped onto a 2-dimensional array called **substitution matrix.**

Because DNA and RNA differ by only one base – (T)hymine for DNA vs. (U)racil in RNA – we can use the same matrix for those. And because there are only four possible DNA or RNA bases but 20 amino acids, the matrix for protein sequence alignment is quite a bit larger. To compare, our DNA/RNA matrix is below:

	A	C	G	T
A	2	-1	-1	-1
C	-1	2	-1	-1
G	-1	-1	2	-1
T	-1	-1	-1	2

For our more robust amino acid substitution matrix, we borrowed the Blosum50 matrix, which is commonly used for comparing amino acid sequences. Match-ups between similar amino acids have a higher score than those between dissimilar ones. Each letter corresponds to a particular type of amino acid (ex. V = Valine, F = Phenylalinine, etc.)

Because one or the other matrix could be used with each algorithm, functors came in very handy. We had two matrix modules (one for DNA/RNA, and one for Blosum50), each with their specific indexing functions, which we then would pass into whatever algorithm module we were using. As a result, each algorithm can handle DNA, RNA, and amino acid sequences.

Algorithms

This tool uses three sequence alignment algorithms that accomplish the following:

1) aligning two entire sequences (the Needleman-Wunsch Algorithm),
2) finding the most likely occurrence (if at all) of a sequence within a larger sequence (the Smith-Waterman Algorithm),
3) finding repeats of a sequence within a larger sequence.

Each sequence alignment algorithm has several things in common:

-It references a substitution matrix to determine scoring for a given match or mismatch between subunits.

-It uses a gap penalty that it applies to scores for mismatched subunits. (We used a penalty of -8.)

-It stores the scores for each potential subunit match-up in a programming matrix.

-The score in each box of the matrix “inherits” part of its score from another box (usually left, above-left, or above.)

-Each box in a matrix has both a score and a path value that points to another box in the matrix on which it bases its score (see below). We used a tuple of ints to store the score – path values.

-As a result of the path value, each algorithm has a method of traversing this matrix while keeping track of its path and updating a score along the way.

-Once the final score has been determined, the algorithm uses the path values to trace back the path taken to its start to determine which subunits were aligned with which. Essentially it uses a linked list across the matrix to determine the alignment.

Sequence Alignment Algorithm Types

Global Alignment: The Needleman-Wunsch algorithm constructs the best alignment between two entire sequences. As such, it works best for sequences of similar length. Each box represents a match possibility between two subunits, and scores for each box are added up recursively. For a given box in the matrix, the final score is the max of three values:

- the current matching score plus the score from the top left
- the final score from the top minus a gap penalty
- the final score form the left minus a gap penalty.

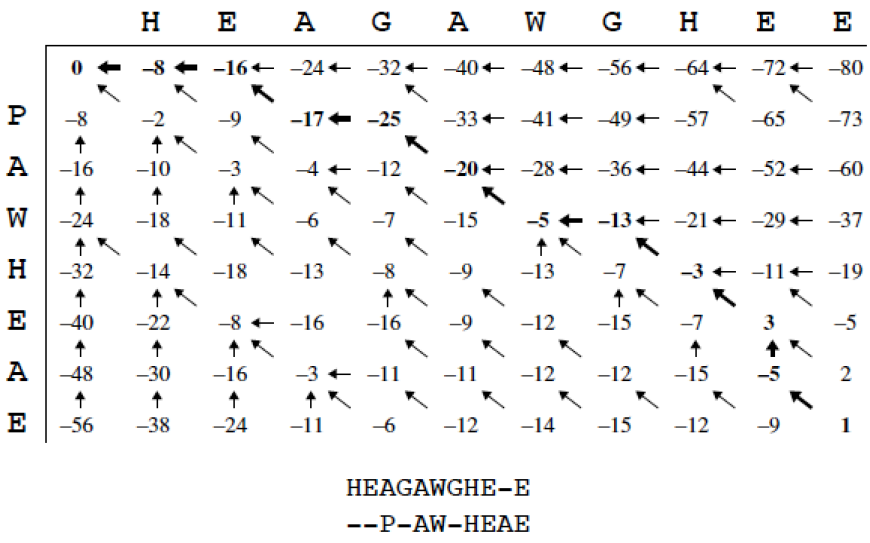

Figure 1. A global alignment programming matrix resulting from the Needleman-Wunsch algorithm, showing scores and traceback pointers. Our implementation of this algorithm produced the same aligned sequences from the two sequences shown. *(*Image from *Biological Sequence Analysis,* Durbin et al, p. 21.)

Local Alignment: The Smith-Waterman algorithm constructs the best alignment between one sequence and part of a larger sequence. The main difference is rather than taking the final score from the lower right-hand corner, it simply looks for the highest score in the matrix and starts its traceback from there. It stops its traceback when it reaches a value of zero. The other difference is that a box's score will become zero and point nowhere if all its scores are negative. (We used integers to represent paths in the path part of our tuple: -1 was for left, 1 was for above, 0 was for above-left, -2 was for a null pointer.)

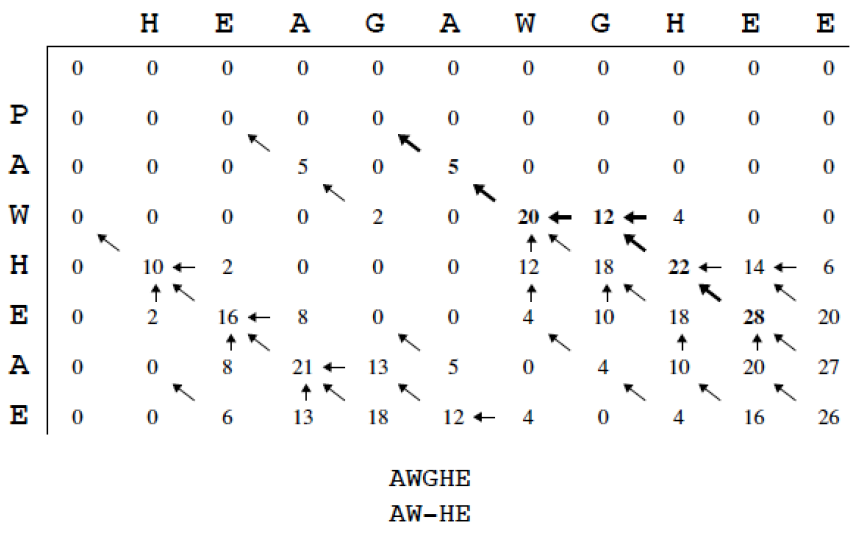

Figure 2. A local alignment programming matrix resulting from the Smith-Waterman algorithm, showing scores and traceback pointers. Our implementation produced the same aligned sequences from the two shown. *(*Image from *Biological Sequence Analysis,* Durbin et al, p. 23.)

Because the local alignment algorithm is only looking for a *region* of the larger sequence that matches up with the smaller sequence, it does not have to traverse the entire matrix, only the part with the highest scoring region. In the figure above, the traceback starts at the box with 28, and then ends at the A-A match and before the P-G mismatch.

Alignment with Repeats: This algorithm looks for repeated occurrences of one sequence in a larger sequence.

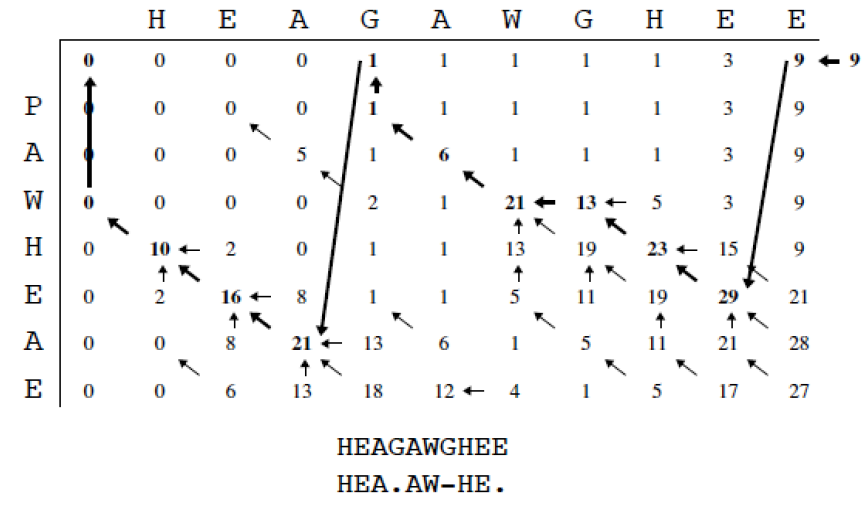

Figure 3. A repeat alignment programming matrix showing scores and traceback pointers. Our implementation produced the same result from aligning the two sequences shown. *(*Image from *Biological Sequence Analysis,* Durbin et al, p. 23.)

The repeats algorithm was the toughest one to implement because the scoring system was more complex than the others. Each box that is not in the top row takes a score that is the max of four values:

- the current matching score plus the score from the top left
- the final score from the top minus a gap penalty
- the final score form the left minus a gap penalty.
- The score of the top row in its column.

In addition, the boxes in the top row have a score that is the max of the previous box, or that of the highest scoring box in the previous column minus a threshold value. Adjusting the threshold value will adjust the result. We kept this at a constant 20 in our algorithm.

Whereas we traversed the local and global sub matrices horizontally throw rows, we traverse our repeat matrix vertically throw columns since we need to know the max of the values in the prior column before handling the top box. For the traceback, there is an extra cell in the upper right hand corner that is the max of the values. You'll notice there is a lot of jumping around in this matrix, which is due to reverting back to an earlier subunit in the smaller sequence as it aligns subunits to the larger sequence.

Mode 2: Conversion

DNA is used as a template for making RNA, which is in turn used as instructions for building proteins out of amino acids. Because DNA, RNA, and protein sequences are so closely related, converting from one to another is useful. For instance, if you have a DNA sequence and compare its protein product with that of another protein, this can then be done in a few steps.

This tool can accomplish the following conversion steps.

1) Conversion of DNA to complementary DNA. Here, all T's are converted to A's and vice-versa. All C's are converted to G's and vice-versa

2) Conversion of DNA to mRNA. All thymine bases, symbolized by T, are converted to uracil bases, symbolized by U.

3) Conversion of RNA to protein(s). During translation from RNA to protein, three bases make a set of three-letter "words" called a codon. Each codon codes for either a specific amino acid, a start signal, or a stop signal.

This third conversion process essentially required a complete algorithm and a 3-dimensional array to store the codon chart that maps out what each codon codes for. While the 3D matrix was straight forward, the algorithm was more challenging to construct than we had foreseen. We compartmentalized the process into two parts: a scan step wherein it scans each base until it hits the "start" codon, AUG – and a translate step wherein it reads in each 3-letter codon and populates a string with the corresponding amino acid symbols. If it reaches a stop codon, it will stop reading and start scanning base-by-base until it reaches another start (AUG) codon, at which point it will start translating again. Because there is the possibility of a sequence fragment being obtained that does not have a start

codon, there is an option for starting the translation right away without having to wait for the AUG start codon step.

Testing Algorithm Functionality. As far as testing our code, all our final tests ran with expected behavior. Below you will find links to each major test of our sequence alignment algorithm. For each test, we used the corresponding query sequence included in our code directory (dna_query1.seq, protein_query1.seq, or rna_query1.seq) and ran the tests, using a max of 10 at all times, a threshold of 0 for local and repeats, and a threshold of -200 for global. The results of our nine tests are linked to below.

	Alignment Type		
DNA:	Local	Global	Repeat
RNA:	Local	Global	Repeat
Protein:	Local	Global	Repeat

Testing Relatedness of Authentic Sequences. First, some quick biology background: according to one classification scheme known as the three domains, life can be classified into bacteria (traditional microbes), archaea (microbes that inhabit extreme environments), and eukarya (all other living things.) In this scheme, classification is based on evolutionary relatedness as indicated by select DNA and RNA sequences. 16S ribosomal RNA sequences are reliable indicators of evolutionary relatedness – similar sequences between organisms imply a recent common ancestor, while dissimilar sequences indicate a distant common ancestor or lack thereof.

Mode 1 Testing. Our initial goal had been to use this tool to learn about similarities in 16S ribosomal RNA in the two main classes of microbes, bacteria and archaea, and we later decided to include three living things from eukarya. Using this resource on rDNA sequences written by Dr. Stephanie Dellis (College of Charleston) as a reference, we translated the listed rDNA sequences into their RNA counterparts for testing. We added these nine rRNA sequences to our RNA database, and then ran nine tests. In each test, we used one of the rRNA sequences as a query and ran it against the database to examine its evolutionary relatedness to the sequences from the eight other organisms. Links to the results of these tests are below.

Eukarya	**Bacteria**	**Archaea**
Human	E. Coli	Methanococcus vannielii
Yeast	Anacystis nidulans	Thermococcus celer
Corn	Thermatoga maratima	Sulfolobus sulfotaricus

Predictably, the rRNA sequences with the highest scores belonged to organisms that were from the same domain as the query sequence's parent organism. For instance, umans were more similar to yeast and corn than to organisms from bacteria and archaea.

Mode 2 Testing. Starting on the right "letter" is very important when RNA is translated into amino acid sequences. Because each codon is a three letter "word" that is translated into an amino acid, starting one or two letters off will throw off the whole process. We demonstrated this in our mode 2 testing when we used our sequence converter module to translate the following sequence into proteins. Here, start codons are in green and stop codons are in red.

GGGGG**AUG**CCCCCCAAAAAAUUUUUU**UAG****AUG**CCCAAAUUU**UAG****AUG**AAACCC**UAG****AUG**ACACACACA**UAG**

We tried two options: 1) going base-by-base and waiting for the start (AUG) codon to translate (the "wait" results), and 2) translating right off the bat (the "start" results).

The "wait" results yielded four short sequences because of the start and stop codons interspersed through the whole sequence. Because of the AUG start codon, the "correct" reading frame was used.

CCC | CCC | AAA | AAA | UUU | UUU → PPKKFF (two prolines, two lysines, two phenylalinines)

CCC | AAA | UUU → PKF (proline, lysine, phenylalanine)

AAA | CCC → KP (lysine, proline)

ACA | CAC | ACA → THT (Threonine, Histidine, Threonine)

The first sequence yielded by the "start" results was long and completely different from the above sequences (GGCPPKNFFRCPNFR), since the number of bases before the first stop codon (GGGGG) is not a multiple of three. As a result, there was a shift in the program's reading frame that kept the correct start and stop codons, as well as the correct amino acids, from being read in. The codon UGA eventually stopped the production of this sequence, but it was not meant to be a stop codon.

Below are the codons that were read in for this sequence and their corresponding amino acids, symbolized by 1- and 3-letter abbreviations. (see this chart of amino acid abbreviations for further info)

GGG | GGA | UGC | CCC | CCA | AAA | AAU | UUU | UUU | AGA | UGC | CCA | AAU | UUU | AGA | **UGA**

Gly	Gly	Cys	Pro	Pro	Lys	Asn	Phe	Phe	Arg	Cys	Pro	Asn	Phe	Arg
G	G	C	P	P	K	N	F	F	R	C	P	N	F	R

After the stop codon, nothing was translated until the next start codon was reached. This is why the last protein fragment was the same.

AUG | ACA | CAC | ACA | **UAG** → (Thre, His, Thre)

Feature List. As we moved from concept to final spec and then coding the final product, we added and dropped features. Features that were dropped due to prioritizing given our time constraints included cosmetic features, such as a web-based GUI, that our TF suggested we should abandon. Features that were added included better client validation and in-program prompts which reduced the number of command-line arguments, as well as a debugging function for printing a matrix, to make sure the algorithms were working correctly.

Dropping the GUI piece also freed us up to explore the algorithm piece in more depth, and as a result, two additional sequence alignment algorithms were added. In addition, we added the second conversion mode, and while DNA to cDNA and DNA to RNA processes were fairly trivial, one fairly complex RNA-to-Protein algorithm was developed. We initially envisioned following a straightforward waterfall methodology but because of the more iterative approach, we ended up with a more Agile-like Software Design Lifecycle.

Below is the table of features planned or coded at each stage of the project.

Feature	Draft Spec	Final Spec	Final Submit Code
Entered Sequence Parser-stdin		x	x
Entered Sequence Parser-filename	X	x	x
Entered Sequence Parser-multi-filename			x
Entered Sequence Parser-Debug Features		x	x
Command Line Parsing & Client Validation			x
Integration to Biocaml	X		
Input data thru Browser	X		
Needleman-Wunsch (global) algorithm	X	x	x
Smith-Waterman (local) algorithm		x	x
Repeats algorithm			
thresholds	X	x	x
output-meta info	X	x	x
output-degree of similarity	X	x	x
Identify start and stop codons	X	x	x
return # of codons	X	x	x
return # of amino acids	X	x	x
Convert a DNA seq to RNA	X	x	x
Convert a DNA seq to complem DNA seq	X	x	x
Convert a RNA seq to amino-acid seq	X	x	x
Return # of bases in DNA seq	X	x	x
Output Data to an html page	X	x	x
Store internal data in a list	X	x	x
Store internal data in a tree	X		

Planning. In our first plan, we mapped out our tasks with end dates. (Here is our draft specification and final specification.) As new features got added and dropped we had to modify. We did, however, meet our deliverables as the project outlined. Our original planning for the most part came to fruition, as. most of the features we planned made it into the final code.

Collaboration. Our group consisted or 2 members. The below table outlines each person's contributions to the project:

Nevin
- Documentation
- Coded all Sequence and Conversion Algorithms
- Controlled git repository
- Created Makefile
- Added improvements to Parser
- Added improvements to Output HTML
- Created Half of the Output Video Presentation
- Tester

Rich
- Documentation
- Wrote Initial Parser
- Wrote Initial output to html
- Half of the Output Video
- Tester

Systems. We had originally planned to integrate our parser, algorithms, and output with the Biocaml library. However, as we look back we realize that Biocaml integration was more than a 4 week task, as we needed more time to learn the inner workings of Biocaml for integration. While Biocaml code is available via github, the documentation to integrate needs improvement – and we found it more productive to simply start with our own code and build our own system of modules from the ground up.

Design. With the course experience fresh in our minds, there was a continual process of looking at a set of similar functions and figuring out how to "abstract out" their common ground and create a more universal function that could serve the same set of purposes. This way of thinking helped us out repeatedly. While some unique functions with singular purposes were used, there were many instances where a single functions of functor was able to serve a range of objectives and thus extend our functionality in ways that surprised us. For instance, it was a surprise that with the right algorithm module, moving from comparing DNA sequences to comparing proteins was so easy. Other examples include the parser, output, and query handler functions that were also designed in general enough ways to serve multiple purposes.

Interfaces. Initially, constructing the interfaces and encapsulating our code in their related modules seemed like extra work, but now we cannot envision this project without them. Creating an interface for

each task and separating each one into a distinct file was an excellent way to compartmentalize the work and isolate specific problems during testing.

Language. Ocaml turned out to be a great language for parsing these sequence types. The use of recursion and pattern matching in particular came in handy on a consistent basis. We also became very familiar with the string and array libraries, which were pleasantly reminiscent of Javascript and C programming. And because of our familiarity with it, the project at one point became far less about syntax and far more about the logic of the algorithms.

Perhaps one of the greatest improvements I feel we made was making the most of Emacs in conjunction with Gedit to test our code. Any time we needed to write small helper functions, we would test them out in Emacs to make sure they worked. This worked out great, because then when we transplanted them into our module, we could be sure they were functioning properly. With larger blocks of code and whole modules, gedit proved instrumental because the error messages were more specific and we could see how the whole system functioned and could make sure everything compiled.

Takeaways. During this project we learned how to plan, code, and test a small sized yet potentially useful piece of software. We also broke down the projects into distinct modules that each of us could code independently. We also came to appreciate the importance of revising our approach mid-stream. At first we had plans that were more visually-oriented and depended on an existing libraries, but as our work progressed we shifted our focus from front-end design to back-end functionality and decided to focus on making our own code as finely tuned as possible, rather than integrating outside libraries.

The most important thing we learned from the project is that a good algorithm can provide a focus from which the rest of the project can grow, and a comprehensive plan for module organization can set one up for success. If we were to undertake a similar project, I think we would have a more singular focus on the algorithms at the outset, as once we had one in place, we were able to develop the matrix, parser, output, responder, and UI modules around it and then adapt these to subsequent algorithms. We would also plan out modules in more detail at the initial brainstorming stage, as the module structure proved to be instrumental in organizing our project.

Future Improvements:

If we were to continue with this project, we would explore the possibility of integrating Biocaml so that FASTA files (industry-standard files for storing DNA) and other file types could be used for sequence alignment. We would also like to add a user-friendly GUI with a web interface.

Were we to continue with adding more file type options, we may end up using a more complex "container" type that can store more meta properties on a sequence. At the time, this was not useful to us, since we were mainly focused on using the algorithms for the sequences themselves.

Within DNA sequences there are also symbols for bases that are ambiguous. Were we to continue we would want to include a more complex DNA/RNA scoring matrix that uses characters for ambiguous bases so that sequences with less clear aspects to them can be aligned.

We would like to add more authentic sequences to the database and organize databases by relatedness. It might also be cool to explore the possibility of using a graph data structure to store our sequences, as I have noticed that the structure of a graph could be used to represent close or distant similarity between known sequences.

We would like to write some compound processes that combine the alignment and conversion steps - for instance, writing a function for first translating RNA to Proteins and then adding the result to our protein sequence alignment database.

Finally, we would like to allow biologists to be able to add their own sequences to this database – so adding logic that sniffs out files in each directory in order to populate its databases instead of using hard-coded string lists would be a terrific new feature.

Advice to Future CS51 Students:

Stick with the OCaml language and trust your own programming expertise. Build the code from the ground up and test it in small chunks to make sure everything works as you move along. Be ambitious about your project but prepared for the complexity that new features bring. Writing self-contained modules with expandable, multipurpose functions will help you to adapt your code as that complexity grows.

References

Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids. Richard Durbin, Sean R. Eddy, Anders Krogh, Graeme Mitchison. Cambridge University Press, 1998.

16S rDNA Sequencing to Identify Unknown Microorganisms. The Virtual Lab Book. Dr. Stephanie Dellis, Biology Dept, College of Charleston.