Experiments with Algorithms for DNA Sequence Alignment

Matt Lehman
Computer Science
Simpson College
Indianola, Iowa 50125
mattlehman@netins.net

Abstract

A DNA sequence is a representation of the genetic code contained within an organism. Molecular biology researchers have great need to compare portions of DNA sequences. This paper discusses the computational problems inherent to this application, and shows how some solutions may be implemented.

Global and local alignment implementations are presented. Known methods of quantifying global and local alignments are discussed and demonstrated. Dynamic programming techniques (using memoization through tabular computation) are discussed and applied to the implementations.

As a background primer to these techniques, the concepts of 'string edit distance' and 'string edit transcripts' are discussed, and implementations to demonstrate them are presented.

Background information is provided on the basics of DNA, the composition of DNA sequences, the motivations for DNA sequence alignment, and computational problems related to DNA sequence alignment.

The relationship between edit distance and string alignment (with provisions for DNA sequence alignment) is demonstrated.

DNA Sequence Alignment

DNA sequence alignment is a representation of the similarity between two or more sections of genetic code. It is used to compare these sections in a quantitative way. Biologists use the comparisons to discover evolutionary divergence, the origins of disease, and ways to apply genetic codes from one organism into another.

DNA Basics

DNA is an acronym for the molecule deoxyribonucleic acid. DNA is contained in each living cell of an organism, and it is the carrier of that organism's genetic code.

The genetic code

The genetic code is a set of sequences which define what proteins to build within the organism. Since organisms must replicate and/or reproduce tissue for continued life, there must be some means of encoding the unique genetic code for the proteins used in making that tissue. The genetic code is information which will be needed for biological growth and reproductive inheritance.

The double helix

DNA is a double-stranded, helical molecule often called a "double helix". Each strand is composed of a sequence of nucleotides. The nucleotide sequence is what encodes genetic information.

Nucleotides

There are four nucleotide molecules, which are identical in all respects excepting a nitrogen base. The four nucleotides are thus named after these different bases: adenine, guanine, cytosine, and thymine. The nucleotides are often denoted by the letters A, G, C, and T.

A nucleotide will bond between the double-strand of DNA to another nucleotide, but the bond may only follow the pairing A-T or G-C. This pairing is called a base pair (bp).

Thus, the two strands composing the DNA molecule are exactly complementary.

Replication

To replicate a DNA molecule, the cell will first split it into two strands. Then, using a pool of free nucleotides within the cell, each strand will have a complementary strand built to fit it, yielding 2 new DNA molecules.

This is basically how cells divide to produce new cells.

DNA sequences

A DNA sequence is our representation of a string of nucleotides contained in a strand of DNA. For example:

ATGCGATACAAGTTGTGA

represents a string of the nucleotides A, G, C, and T.

Codons

Although the genetic code is composed of DNA sequences, proteins are not built directly from them. There are intermediary chemicals called amino acids which, when combined in a certain order, lead to proteins. The DNA sequence is split into triplets of nucleotides which code for these amino acids. The triplet is called a 'codon'.

There are approximately 20 amino acids used in proteins.

An amino acid is encoded by a sequence of three (3) nucleotides, called a codon.

For example, the amino acid methionine is represented by the codon ATG.

If we had other amino acids represented by the codons CGA, TAC, AAG, TTG, and TGA, then we could string them together to produce a sequence:

ATGCGATACAAGTTGTGA

which may represent the encoding for a particular protein comprised of 6 amino acids. (We have 18 nucleotides here, and each amino acid is encoded by 3 nucleotides.)

Since we have 4 possible nucleotides and a codon is composed of 3 nucleotides, there are 4^3 = 64 possible codon triplets. But since in the whole of life on Earth there are only 20 amino acids used to compose proteins, most amino acids are specified by more than one codon. See Shamir [1].

This means that redundancy is built into the coding scheme. In part, this is nature's way of ensuring that coding errors do not produce lethal mutations. It has been shown, for example, that only 30% coding identity is required to produce the same protein. See Shamir [2].

DNA sequence alignment

Motivations

Some of the most common uses for DNA sequence alignment are in determining the function of new sequences, and medical applications. Tompa [3]

Determine function of new sequences

As new sequences are discovered and catalogued, it becomes necessary to hypothesize their function. How can we do this?

One way is to look for matches in a database of sequences for which we already know the protein encoding. If we can find a match (or close match) in this database, we then have a clue as to the new sequence's function.

Medical applications

Multiple sclerosis is a disease in which the immune system T-cells attack the body's myelin sheath around nerves.

It was hypothesized that myelin sheath proteins are similar to viral or bacterial sheath proteins from an earlier infection. So, researchers:

1. sequenced myelin sheath proteins
2. searched a protein database for similar bacterial and viral sequences
3. performed lab tests to determine if T-cells attacked these same proteins

They discovered that the immune system was indeed confusing bacterial and viral proteins with the body's own myelin sheath proteins. This was a vital step in the progress toward treating multiple sclerosis. See Tompa[4]

Computational problems

Dot Plot

Regions of similarity between two DNA sequences can be plotted by hand. The technique is to create a table, putting one sequence on a vertical axis and the other sequence on a horizontal axis. The table is populated with dots which mark a match between nucleotides in the sequences.

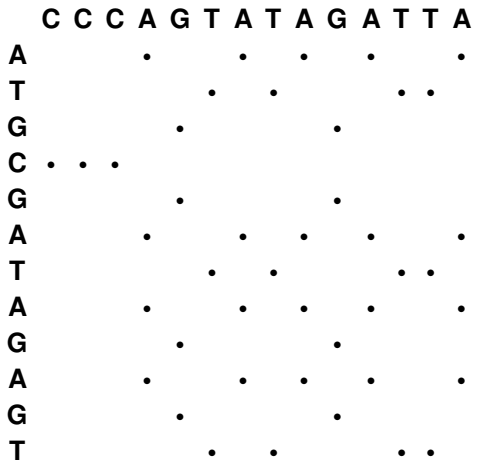

Figure 1: Dot Plot

In the dot plot of Figure 1, the sequence ATGCGATAGAGT is matched against the sequence CCCAGTATAGATTA. Regions of similarity occur where it is apparent that there is a string of diagonal dots in the dot plot.

The dot plot is fine for small sequences, but is not adequate for very long ones.

Complexity and scale

The human genome has approximately 3 billion base pairs (bp), and some simple amoebas have 200 times the DNA as humans. See Shamir [5]

An average protein has 200 amino acids, while a large one has 1000. Since an amino acid is coded by 3 nucleotides, this means that the sequence size for a protein can be from 600 to 3000 nucleotides.

If we are to search the human genome for a small protein, then the table to match the bases will occupy 1,800 billion cells of memory.

Needless to say, techniques must be developed to address this huge space, and also to perform calculations within this space more optimally than brute-force.

These are the basic problems of DNA sequence alignment.

Concepts from computer science

In order to understand the possible computable solutions to the problem of DNA sequence alignment, it will be helpful to review the concepts of string edit distance and dynamic programming.

Edit distance

Edit distance can be thought of as the "difference" between two strings. The difference between two strings is measured by counting the number of edit operations which must be performed, character by character, to transform one string into another. These edit operations are:

- R = replace
- I = insert
- D = delete
- M = match

For example, to transform the string "cat" to the string "chat" we can insert (I) the character 'h' between the 'c' and 'a' of "cat", yielding the string "chat".

There are many possible edit distances between "cat" and "chat", but the minimal edit distance is one (1) -- just one insertion.

Our objective is to discover a *minimal edit distance* which will tell us the minimum number of edit operations which may be used to transform one string into another. This number is the most interesting because we are usually searching for strings or portions of strings with the most similarity.

Edit Distance Definition

How can we derive or calculate a value for the edit distance between two strings? We must use a general description of the problem. To illustrate:

Say that we have two strings S and T, of length n and m, respectively.

$|S| = n, |T| = m$

We can define

D(i, j)

as the value of the minimal edit distance between strings

S[1]…S[i] and T[1]…T[j]

so that the *minimal edit distance between S and T* is

D(n,m)

In order to calculate D(i,j), we must:

establish a *base condition*, based on two cases:

D(i,0) = i

D(0,j) = j

(note: D(0,0) = 0)

and establish a *recurrence relation*, used in all other cases:

$$\text{D(i,j) = minimum} \begin{Bmatrix} D(i-1, j)+1, \\ D(i, j-1)+1, \\ D(i-1, j-1)+t(i, j) \end{Bmatrix}$$

where

t(i,j) = 1 if S[i] ≠ T[i], and

t(i,j) = 0 if S[i] = T[i]

See Gusfield [6].

Edit Distance in Plain English

When we refer to D(3,4) on the alignment between "cat" and "chat", this means that we are trying to find the minimal number of edit operations required to transform the first three (3) characters of "cat" into the first four (4) characters of "chat". In this case, this applies to the whole strings.

The base condition cases are easily derived from this way of thinking: in order to transform the first *i* characters of a string to *0* characters (the null string), it will require *i*

deletions. In our notation, D(*i*,*0*) = *i*. The same logic applies to transorming a null string into some string with *j* characters (using *j* insertions), yielding D(*0,j)* = *j*.

The recurrence relation operates upon the principle that the edit distance value of any D(i,j) is dependent upon the edit distances of alignments that come "before" it. Going from D(i-1,j) to D(i,j) represents an insertion. Going from D(i,j-1) to D(i,j) represents a deletion. Going from D(i-1,j-1) to D(i,j) represents a replacement if S[i] ≠ T[j], and it represents a match if S[i] = T[j].

Edit Distance Example

For example, consider the edit distance between the strings "cat" and "chat":

S = "cat", T = "chat"

n = |S| = 3, m = |T| = 4

Thus, the minimal edit distance between "cat" and "chat" is defined as:

D(n,m) = D(3,4)

Since there is only one edit operation required to transform "cat" to "chat" (inserting the 'h' character), the value of D(3,4) is 1.

Cognitively, we can think of this function as stating that the minimal effort required to transform the first three (3) characters of "cat" into the first four (4) characters of "chat" is one (1) edit operation.

Applying this generalization to other transformations, we can see, for example, that D(3,0) represents the minimal effort required to transform the string "cat" into the null string. (The edit operations are: delete 'c', delete 'a', and delete 't'). Thus, D(3,0) has value of 3.

Edit transcript

The series of edit operations is called an 'edit transcript', and is represented in the following way:

```
  I                 (the edit transcript)
c   a t             (original string)
c h a t             (transformed string)
```

(edit distance = 1)

The position of the edit operation 'I' shows where that operation is performed upon the original string.

Another example (from Gusfield):

```
R I M D M D M M I    (the edit transcript)
v   i n t n e r      (original string 'vintner')
w r i   t   e r s    (transformed string 'writers')
```

(edit distance = 5)

Note that there may be more than one edit transcript to achieve the same alignment:

```
  R I                (the edit transcript)
c a   t              (original string 'cat')
c h a t              (transformed string 'chat')
```

(edit distance = 2)

We can see that the edit distance is the sum of insertions, deletions, and replacements in an alignment of strings. The minimal edit distance is the distance found in an optimal alignment of strings.

Calculating Edit Distance

It is evident that in order to calculate D(n,m) we must be able to calculate D(i,j) for any i and j. This calculation can be done recursively, or through dynamic programming.

It will become clear that dynamic programming is the preferred method, but in order to use it (and understand why it is better) we must first understand the recursive method.

Our definition of edit distance is a recursive one. That is, our desired final value is defined by simpler (but unknown) values.

In implementation, a computer program would use a call stack which repeats from D(n,m) down to the known values of D(i,0) or D(0,j). Then, the values are passed back up the call stack to compute our desired final value.

The recursive method is fine for a definition, but poor in implementation because it is so inefficient. The complexity lies at about $O(2^n)$.

Dynamic Programming

Dynamic programming uses the concept of 'memoization' to eliminate calculating values more than once. Memoization is simply the process of storing a calculated value which we know will be used to calculate other values later in computation.

In the case of sequence alignment, we take a cue from the use of dot plots (See Figure 1). Similar to a dot plot, we create a table arranging one string on a vertical axis and another string on a horizontal axis. Unlike a dot plot, however, we populate the cells not with dots but rather with numerical values. The numerical values represent an edit distance (or some other score, as will be shown later in the case of DNA sequences.)

The cells in a dynamic programming table are filled from left to right, top to bottom. We do this because we can start with our *base condition*, which is the set of known values D(i,0) and D(0,j). The bottom-most, right-most value will be the minimal edit distance, D(n,m).

		c	**h**	**a**	**t**
	0	1	2	3	4
c	1	0	1	2	3
a	2	1	1	1	2
t	3	2	2	2	1

Figure 2: Dynamic programming table for cat:chat

In Figure 2, the bottom-most, right-most value represents D(3,4), which is the minimal edit distance between the strings "cat" and "chat". This value is 1.

		w	**r**	**i**	**t**	**e**	**r**	**s**
	0	1	2	3	4	5	6	7
v	1	1	2	3	4	5	6	7
i	2	2	2	2	3	4	5	6
n	3	3	3	3	3	4	5	6
t	4	4	4	4	3	4	5	6
n	5	5	5	5	4	4	5	6
e	6	6	6	6	5	4	5	6
r	7	7	6	7	6	5	4	5

Figure 3: Dynamic programming table for vintner:writers

In Figure 3, the bottom-most, right-most value represents D(7,7), which is the minimal edit distance between the strings "vintner" and "writers". This value is 5.

Using a dynamic programming algorithm, our time complexity is reduced to O(nm). See Gusfield [7]

Applications for DNA sequences

It may be evident now that the concepts of edit distance and edit transcript may apply to DNA sequences. There is nothing stopping us from viewing a DNA sequence as just a string, albeit with a limited alphabet comprised of the characters ‘A’, ‘T’, ‘G’, and ‘C’.

Alignment

Alignment is a representation of the similarity between strings. It does not show us the operations performed (as an edit transcript does) but rather shows us the how those strings would line up next to one another in the most favorable comparison. The most favorable comparison is the transformation with minimum edit distance.

Regarding biological applications to DNA sequences, Gusfield states that “Different evolutionary models are formalized via different permitted string operations, and yet these can result in the same alignment.”

Here is an optimal alignment for the strings “vintner” and “writers” (again from Gusfield):

```
v - i n t n e r -
    |   |   | |
w r i - t - e r s
```

The pipe character (‘|’) represents matching characters, and a dash character (‘-‘) represents a gap in the alignment caused by a substitution, deletion, or insertion.

Here is an alignment of DNA sequences:

```
Edit transcript:  R R R R M I M M M M M R M I

      String #1:  A T G C G - A T A G A G T -
                            |   | | | | |   |
      String #2:  C C C A G T A T A G A T T A
```

Comparing alignments

This visual alignment is all fine and well, and the minimal edit distance may be a good way to quantitatively compare strings, but what about DNA sequences? It turns out that gaps are important in DNA sequences because they represent significant biological events. A run of gaps must be accounted for, and it would be better to score insertions, deletions, and replacements differently. We must have some way of scoring our alignments.

Gaps are an important concept in biological applications, because a stream of gaps in a DNA sequence may represent a significant biological characteristic. Gaps usually incur a 'penalty' to the potential alignment score between two sequences, depending on the length of the gap.

The way to achieve a more useful score is to turn around our definition a bit. Instead of computing the *minimal edit distance* between strings or sequences, it will help to compute the *maximum similarity* between them. To do this, we need a *scoring function*.

Say that we have two strings S and T of differing lengths.

Alignment A maps S and T into strings S' and T', such that |S'| = |T'|.

If we remove the gaps from S' and T', we will have restored S and T.

The value of alignment A is defined as:

$$\sum_{i=1}^{l} \sigma(S'[i], T'[i]), \text{ where } l = |S'| = |T'|$$

and σ(x,y) is a *scoring function.*

An *optimal alignment* is an alignment that has maximum possible value for these two strings.

The scoring function produces a value dependent upon the matching qualities of two characters in our strings. For example, with two distinct characters 'A' and 'G', we can employ a scoring function defined as such:

σ(A,A) = +2,	// this is a match
σ(G,A) = -1,	// this is a substitution of 'A' for 'G'
σ(G,-) = -1, and	// this is a deletion of character 'G'
σ(-,G) = -1	// this is an insertion of character 'G'

(from Tompa)

Using a scoring function, we can apply different scoring tables to our algorithm to achieve results which are biologically interesting.

In moving from the idea of edit distance to the idea of alignment, we have become more interested in the quantity of maximum similarity, as opposed to the quantity of minimum distance. See Tompa [8]

Scoring a Global Alignment

Say that we have two strings S and T, of length n and m, respectively.

|S| = n, |T| = m

We can define

V(i, j)

as the value of the maximum score of the string alignments

S[1]…S[i] and T[1]…T[j]

so that the *maximum score of an alignment between S and T* is

V(n,m)

In order to calculate V(i,j), we must:

establish a *base condition*:

V(0,0) = 0

V(i,0) = V(i-1,0) + σ(S[i],-) for i > 0

V(0,j) = V(0,j-1) + σ(-,T[j]) for j > 0

and establish a *recurrence relation*:

$$\text{V(i,j)} = \text{maximum} \begin{Bmatrix} V(i-1,\, j) + \sigma(S[i], -), \\ V(i,\, j-1) + \sigma(-, T[\,j]), \\ V(i-1,\, j-1) + \sigma(S[i], T[\,j]) \end{Bmatrix}$$

for $0 < i \geq n$ and $0 < j \geq m$

See Tompa [9]

Algorithms Implemented

To enhance my understanding of the principles of DNA sequence alignment, I have written four programs in C++, two of which use the dynamic programming techniques I have detailed above.

DotPlot

The first program, called “dotplot”, will take as input two strings or sequences and produce a table populated with the matching dots between the strings. Output is configurable and the user can choose HTML or plain text output. This program is useful as a teaching tool to generate and demonstrate the early beginnings of sequence alignment.

Minimal Edit Distance

The second program, called “ed”, calculates the minimum edit distance between two strings. The strings are supplied by the user, either on the command line or through a plain text file. The program uses an in-memory dynamic programming table to arrive at the minimum edit distance score. It will also calculate the total number of alignments which produce this edit distance. The program displays an alignment of the two strings, along with an edit transcript which shows the operations used to achieve that alignment.

Global Alignment

The third program, called “global”, calculates the score of an optimal global alignment between strings or sequences. The scoring function is configurable by the user. This program can use either an in-memory dynamic programming table (if an alignment is desired) or be configured to discard tabular values along the way (thus saving memory).

Local Alignment

The fourth program, called “local”, calculates the score of optimal local alignments between two strings or sequences. These alignments show regions of similarity between strings or sequences. This program only calculates the score and does not present an alignment or transcript. The local alignment is a special case of global alignment which is useful to researchers. Although local alignment was not discussed in this paper, it is an often used concept in biological research which I understand but have not written about. The implementation is based upon “global” and otherwise has the characteristics of that program.

Acknowledgements

This work has been accomplished at Simpson College as independent study under the supervision of Dr. Lydia Sinapova.

References

[1] Shamir, Ron, *Algorithms for Molecular Biology*, Lecture 1, Fall 2001, Section 1.1.4
<url: http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html>

[2] Shamir, Section 1.1.5

[3] Tompa, Martin, *Lecture Notes on Biological Sequence Analysis,* Technical Report #2000-06-01, Winter 2000, Section 3.2.1
<url: http://www.cs.uml.edu/bioinformatics/resources/lectures/tompa00lecture.pdf >

[4] Tompa, Section 3.2.2

[5] Shamir, Section 1.5

[6] Gusfield, Dan, *Algorithms on Strings, Trees, and Sequences,* pp. 217-218

[7] Gusfield, p. 218

[8] Tompa, Section 3.3

[9] Tompa, Section 4