IQS2	Spring 2012

Programming Project 1: Smith-Waterman Global Sequence Comparison

Assigned: Tuesday, January 10 *Due: Monday, January 30, 5:00 pm*

1 Introduction

In this project, you will write a Java application that uses the global Smith-Waterman algorithm (you may have called it the Needleman-Wunsch algorithm last semester) to determine the optimal global alignment of two nucleotide sequences. Though you are familiar with the algorithm, I have provided a description of it as an Appendix in this writeup.

What one considers the optimal alignment depends on both the algorithm and the values of the various algorithm parameters. For Smith-Waterman, these parameters are the two sequences, A and B, a gap penalty, denoted by g, and a weight matrix that lists scores assigned to nucleotide pairs, denoted here by $s(a_i, b_j)$, where a_i and b_j are respectively the ith nucleotide in sequece A and the jth nucleotide in sequence B.

Since these parameters are required and variable, your program must somehow acquire this information. Once acquired, the information should be used by your program to compute both the optimal similarity score and the optimal alignment. Unlike some of the previous assignments in IQS, there is no graphical component to the project: the alignments should be printed to the terminal (standard output) aligned as described in the algorithm description section below.

2 Minimum Required Functionality

Your application must provide at least the following minimum criteria:

- Your application must be configurable and flexible. Specifically,
  - It should be able to read input sequences from a special sequence input file. Files may contain several sequences, so there needs to be a means to determine which two sequences should be used. This will be discussed in greater detail below.

  - It should be able to read the weight matrix from a special matrix input file.

- Because of the need for several configuration parameters, your program must accept a single command line argument that gives the name of a configuration file specified by the user. All other user input is obtained from this configuration file.

- The minimum configuration options handled by your code must include:

  - The name of the weight matrix input file.
  - The value of the gap penalty.
  - The name of the sequence input file.
  - The value of a variable called `sequenceA`, which specifies the sequence number of the sequence in the input file that should be used as sequence A.
  - The value of a variable called `sequenceB`, which specifies the sequence number of the sequence in the input file that should be used as sequence B.
  - The user should have the option of specifying the maximum number of columns to appear on a single line of output. Default value should be set to 80.
  - The user **must not** be required to specify the lengths of the input sequences. That is, this must not be a configuration option. Rather, your code should determine these lengths simply by reading the appropriate sequences from the specified input file (see related comment on the specification of input file formats below). There should be no explicit limit on the lengths of the input sequences.

3 File Formats

Input files must adhere to the following format rules. Any file that does not (for example if there are inappropriate characters or there are more sequence end characters than there are sequences) should trigger an error message followed by immediate exit from the program.

3.1 Configuration Files

Configuration files should consist of multiple lines of the form

```
parameter = value
```

with only a single parameter value on each line. Lines consisting entirely of white space should be ignored, as should lines beginning with the pound sign(#) which should be considered comments.

Legal parameter values must include (exactly as listed here, and these are case sensitive)

- `seqenceInputFile`
- `sequenceA` The sequence number of the first sequence.
- `sequenceB` The sequence number of the second sequence.
- `weightMatrixFile`
- `gapPenaltyValue`
- `maxColumns` The maximum number of columns per line of output.

3.2 Weight Matrix Input Files

The weight matrix is four by four, and to allow for greater flexibility, you should *not* assume that it is symmetric. Each line of a weight matrix input file should represent a row in the matrix, and thus should consist of four doubles separated by one or more spaces. You need to consider as well that some of these values may be negative. You should assume that the order of nucleotides corresponding to both the rows and columns of the matrix is A, C, G, T. The is, the entry in row 0, column 0 is $s(A, A)$, the entry in row 0, column 1 is $s(A, C)$, the entry in row 1, column 0 is $s(C, A)$, etc. Lines consisting entirely of white space should be ignored, as should lines beginning with the pound sign(#) which should be considered comments.

3.3 The Gap Penalty

The gap penalty should always be negative, since it is, after all, a penalty. Some texts, when discussing the gap penalty, always specify it as a positive value, with the understanding that a gap will cause this value to be *subtracted* from scores while they are being computed. Others texts give it as a negative value, with the understanding that the gap penalty should be *added* to scores while they are being computed. To remove this ambiguity, for this project, the value given in the configuation file should be positive, so that during computation, this value will be subtracted from a similarity score when a gap occurs.

3.4 Sequence Input Files

These should consist of multiple lines of characters from the alphabet $\Sigma = \{A, C, G, T\}$. Each sequence begins with the string `sequence:`, followed by an integer, followed by another colon, followed by the nucleotide sequence itself. For example

```
sequence:1:ACCGTTCTGAGTCGATCX
```

Whitespace should be ignored. The uppercase `X` character marks the end of a sequence. Lines consisting entirely of white space should be ignored, as should lines beginning with the pound sign(#) which should be considered comments. The integer between the colons specifies the sequence number, and this is what the user must specify in the configuraton file.

3.5 Representing the Weight Matrix

One can represent a matrix in Jave using a two dimensional array. If, however, one finds it necessary to perform any kind of matrix arithmetic (such as inverting a matrix, multiplying two matrices, or adding two matrices), it is much easier to use a Matrix class. Though this project does not require such matrix manipulations, future projects might. For this reason, I would like you to import the JAMA Matrix package (implemented by the National Institute of Standards and Technology – see `http://math.nist.gov/javanumerics/jama/`). The package comes packed in a jar file, `Jama-1.0.2.jar`, which you should download from the IQS2 CS website, and install in BlueJ following the steps described for installing squint. I have added a link to the JAMA documentation in order to help you in using the `Matrix` class. Please note that you should add the line

```
import Jama.*;
```

to any file that uses any of the JAMA classes.

3.6 The SequenceComparisonArrayElement class

The primary result of your code is to print the optimal sequence alignment, as opposed to simply printing the similarity score. With the global Smith-Waterman algorithm, this means you need to "backtrack" once you have finished computed the dynamic programming matrix.

The need to backtrack introduces some issues in terms of how one chooses to represent the dynamic programming matrix. For example, considering that we are using the JAMA package, you might be tempted to use a `Matrix` object to represent the dynamic programming matrix. This is possible. But if you choose to do this, you will essentially have to compute the entire dynamic programmming matrix twice: once the first time to fill the values, and then once when backtracking in order to determine the path that gives the alignment.

In order to avoid this, one can instead create an object, lets call it a `SequenceComparisonArrayElement`, that will act like an array element, but carry two pieces of information: the double value that belongs in a specific space in the dynamic programming matrix, along with the "direction" from which this value was derived. That is, you know from the previous semester that computing the value of the i, j-element of the matrix, $S_{i,j}$, involves computing a max:

$$S_{i,j} = \max\{S_{i-1,j} - g, S_{i-1,j-1} + s(u_i, v_j), S_{i,j-1} - g\},$$

where here g represents the gap penalty (which in this formulation should be positive). The value of $S_{i,j}$ will thus be derived from either $S_{i-1,j}$ ("up"), $S_{i-1,j-1}$ ("diagonally"), or $S_{i,j-1}$ ("back"). In order to compute the optimal alignment, you need to know which of these gave the max (that is, from which of these is $S_{i,j}$ derived). So a `SequenceComparisonArrayElement` has two data fields, the first a double which holds the value of the score, and the second a char, which holds either a "u", "d", or "b", depending on which "direction" gave the max.

Your dynamic programming matrix in this case is represented using a two dimensional array of `SequenceComparisonArrayElement` objects. This is how you are required to represent the dynamic programming matrix in this project.

4 The Project in Two Parts

This is a large project that is probably more involved than anything you coded during the first semester. For this reason, I require that students complete the project in two parts. No only does this help partition the code, but it also forces you to have completed roughly half of the assignment within about ten days of when the project was assigned. For this semester, that means that **I expect to receive your working Part I by 5pm on Friday, January 20!**

Part I of the project deals with reading information from the configuratiom, matrix, and sequence input files. Once correctly read, your code will use an object of class `SequenceComparisonEngine` to perform the sequence alignment. I will provide you with this class file for use in Part I, as well as the API for using a `SequenceComparisonEngine`.

Part II of the project is simply to implement the `SequenceComparisonEngine` class!

4.1 The `SequenceComparisonEngine` Class

The `SequenceComparisonEngine class` has a single constructor. This constructor takes five input parameters in the following order:

- A `Matrix` object that represent the weight matrix.
- A `double` that represents the gap penalty.
- A `StringBuffer` object that represents the first of the two sequences that are being compared.
- A `StringBuffer` object that represents the second of the two sequences that are being compared.
- An `int` that represents the maximum number of columns of the alignment that should be displayed on a single line.

Once a `SequenceComparisonEngine` object has been created, one only has to call its `executeSmithWaterman()` method (which takes no parameters) to run the Smith-Waterman comparison.

4.2 Implementing the `SequenceComparisonEngine` Class

In order to complete the `SequenceComparisonEngine` class, you should implement four methods:

- The constructor. This must take five input parameters. The order of the parameters, along with what they represent, is listed above. This information is also given to you in the starter code for this class.

- The `computeDynamicArray()` method. This method computes the values that should go in the Smith-Waterman dynamic programming array.
- The `generateAlignment()` method. This method backtracks through the dynamic programming array to determine the optimal alignment. You will probably want to place the alignment in two `StringBuffer` objects, one for the "top row" of the alignment, and one for the "bottom row".
- The `printAlignment()` method. This does just what it's name implies. It should print out the alignment in ***the exact format that my*** `SequenceComparisonEngine` ***prints out alignments!***

5 A Complication

In doing some deep thinking about implementing the Smith-Waterman algorithm, it occurs to me that it might be good to inform you about a small complication that arises when building the Smith-Waterman dynamic programming array. The issue arises because of the extra column and extra row that are part of the array. Consider, for example, the array shown below:

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	3	4	4	4	4	4
G	0	1	2	2	3	3	3	4	4	5	5	5
A	0	1	2	3	3	3	3	4	5	5	5	6

This is a Smith-Waterman dynamic array corresponding to the situation where the gap penalty is zero (though there is nothing special about that except that it makes both the first row and the first column all zeros). Notice that the first column corresponds to having no symbol from the `GAAT...` sequence in the alignment, while the first row corresponds to having no symbol from the `GGAT...` sequence in the

alignment. In terms of indexes, the left most column of the array is column zero, while the top row is row zero. But because of the need for the extra row and extra column, the **zero** element of the sequence `GAATTCAGTTA`, which is the nucleotide `G`, corresponds to column **one** of the array. Similarly, the **zero** element of sequence `GGATCGA` corresponds to row **one** of the array. In general, then, if one is filling in the row i, column j position of the array, one is dealing with the index i - 1 nucleotide of the sequence on the left, and the index j - 1 nucleotide of the sequence at the top.

One can account for this in one of two ways:

1. Keep careful track of indices in your code and always subtract one from indices when you are accessing elements of the nucleotide sequences. OR
2. Switch the indices of your `StringBuffer` objects at the beginning of your `SequenceComparisonEngine` code so that the indices match those of the array. This means simply shifting the nucleotides right one index. The easiest way to do this is to create a `new StringBuffer` that starts with the lone character `X`. (Any character will do—you'll never access it so it doesn't matter.) Then append your original `StringBuffer` to this new one. Once done, the indices of your sequences will align with the indices of the dynamic programming array.

In my code I used option number 1 (for efficiency reasons). You can do it either way, but option two makes for easier coding.

6 Perhaps of some help

- As you know, the Scanner class is nice for reading and parsing text files.
- In the final version of your project, your output will not be guaranteed to be the same as mine unless you "break ties" in the same manner that my code did. That is, when building the dynamic programming matrix, it can happen that at times more than one of the three options in the "max" expression will give the same score. In order to get the same answer that my code gives, you need to break those ties in a manner consistent with what I did. Specifically, in the event of ties, going "diagonally" has highest priority, going "up" has second priority, and going "back" has lowest priority.
- I will provide you with valid configuration, matrix, and sequence input files, so that you can test your implementation. I will also provide you with my class (but not source) file in order for you to compare your results to mine.

7 Deliverables

You should provide me with a zipped BlueJ project called `XXXSequenceComparison.zip`, where `XXX` represents the first three letters of your last name. The name of the source file containing the `main` method must be `SequenceComparison.java`. (Note that there is no `XXX` part of the class name. That is only required as part of the project file name).

Your project should be submitted to me by dropping it in my netfiles inbox (`d/dszajda`).

A The Smith-Waterman Algorithm

A thorough treatment of sequence comparison techniques would (and does) fill several texts. This section gives a brief description of a dynamic programming alignment technique developed by Smith and Waterman. The sequences that biologists study consist of either nucleotide bases (occurring in DNA fragments) or amino acids (the building blocks of proteins). We consider only DNA sequences, for which the underlying alphabet, Σ, consists of the set $\{A, C, T, G\}$ representing the nucleic acids adenine, cytosine, thymine, and guanine.

Let $U = u_0 u_1 \ldots u_{n-1}$ be a sequence over Σ. Sequences evolve primarily in three ways. Either an element of a sequence is removed (a *deletion*), an element is inserted (an *insertion*), or an existing element is transformed into a different element (a *substitution*). Biologists track evolutionary changes by writing the original sequence alongside the new sequence with appropriate positions aligned. For example, if U = CTGTTA, and u_1 undergoes a transformation from T to A, this would be written

$$\begin{array}{ll} U: & \text{CTGTTA} \\ V: & \text{CAGTTA} \end{array}$$

If instead u_3 is deleted from U, this is written

$$\begin{array}{ll} U: & \text{CTGTTA} \\ V: & \text{CTG–TA} \end{array}$$

where the '–' symbol acts as a placeholder, allowing the other symbols to remain aligned. Positions in

a sequence with the '–' symbol are called *gaps*. If the element G is inserted in position 1 of U, this is represented by

$$\begin{array}{ll} U: & \text{C}-\text{TGTTA} \\ V: & \text{CGTGTTA} \end{array}$$

After several such mutations, U may have evolved significantly. We can represent this evolution with an alignment such as the following.

$$\begin{array}{ll} U: & \text{C}-\text{TGT}--\text{TA}-- \\ V: & \text{CTA}-\text{TGCT}-\text{CG} \end{array}$$

In the example above, we assume that V evolves from U. In general, however, when given an alignment of two sequences, there is no implied origin — it is impossible to tell whether a particular gap is caused by a deletion or an insertion. Because of this symmetry, insertions and deletions are considered the same event, an *indel*.

Note also that two sequences can be aligned in several ways, and that aligned sequences need not have the same length. For example, in addition to the alignment above, U = CTGTTA and V = CTATGCTCG also have the following alignment.

$$\begin{array}{ll} U: & \text{CT}-\text{GT}--\text{T}-\text{A} \\ V: & \text{CTA}-\text{TGCTCG} \end{array}$$

It is stated (though not proved) in some of Waterman's work that the number of alignments of two sequences of length n is asymptotically equal to $(2^{5/4}/\sqrt{\pi})(1+\sqrt{2})^{2n+1}(1/\sqrt{n})$. Thus, for example, two sequences of length 1000 have approximately 7.03×10^{763} distinct alignments.

Goodness of an alignment is measured using a scoring function s defined on pairs of symbols in Σ, and typically having the form

$$s(u,v) = \begin{cases} c & \text{if } u = v \\ -d & \text{if } u \neq v, \end{cases}$$

for nonnegative integers c and d with c equal to or slightly larger than d. This function can also be described using a matrix, called the *weight matrix*. Gaps are scored using a *gap function* (or *gap penalty*) g. Gap

functions usually impose either a constant penalty per gap, or have an *affine* form, with

$$g(k) = \alpha + \beta(k-1),$$

where k is the length of the gap, $\alpha > 0$ is the penalty of the initial indel in a multiple column gap, and $\beta > 0$ is the penalty for each subsequent indel in the gap. The score of an alignment is defined as the sum of the scores of each individual column, minus the gap penalties. By carefully choosing the scoring function and gap penalty, goodness of fit can be made to correspond to intuitive notions such as the probability that the sequences evolved from a common ancestor.

The similarity $S(U, V)$ of sequences U and V is defined to be the maximum score over all alignments between the two sequences. Although the number of alignments is huge, a dynamic programming algorithm developed by Needleman and Wunsch, and later augmented by Smith and Waterman, allows the similarity of sequences of length n to be determined in $O(n^2)$ time.

Details of the Needleman-Wunsch (e.g., global Smith-Waterman) algorithm follow. We assume here, for simplicity of notation, that length n sequences begin at index 1 and end at index n. Recall that S denotes the similarity score of a sequence pair, s denotes the similarity function for symbols, and g denotes the gap penalty.

Theorem 1. *If $U = u_1 u_2 \ldots u_n$ and $V = v_1 v_2 \ldots v_m$, define*

$$S_{i,j} = S(u_1 u_2 \ldots u_i, v_1 v_2 \ldots v_j).$$

Also, set

$$S_{0,0} = 0, \quad S_{0,j} = \sum_{k=1}^{j} g(v_k), \text{ and } S_{i,0} = \sum_{k=1}^{i} g(u_k).$$

Then

$$S_{i,j} = \max\{S_{i-1,j} + g(u_i), S_{i-1,j-1} + s(u_i, v_j), S_{i,j-1} + g(v_j)\}.$$

Proving the validity of the dynamic programming approach in this context is straightforward. One need only observe that an alignment ending with indices i and j must end with one of the choices below.

$$\begin{array}{ccc} \cdots u_i & \cdots u_i & \cdots - \\ \cdots - & \cdots v_j & \cdots v_j \end{array}$$

Thus the best alignment ending with indices i and j must be the best alignment ending with indices i and $j-1$ plus the gap penalty, or the best alignment ending with indices $i-1$ and $j-1$ plus $s(u_i, v_j)$, or the best alignment ending with indices $i-1$ and j plus the gap penalty.

The alignments discussed above are *global* alignments because they include every element of both sequences. More common (and more compute intensive) are the so-called *local* alignments, in which case one seeks the best matching substrings of the two sequences. Specifically, the problem is to find

$$H(U,V) = \max\{S(u_i u_{i+1} \ldots u_{j-1} u_j, v_k v_{k+1} \ldots v_{l-1} v_l) : 0 \leq i \leq j \leq n-1,\ 0 \leq k \leq l \leq m-1\}.$$

The amount of compute required to consider all possibilities is considerable, as there are $\binom{n}{2}\binom{m}{2}$ different sequence alignments required here. Waterman notes that computing H for two length n sequences using the above algorithm for computing each S take $O(n^6)$ time. Fortunately, a dynamic programming attack similar to the global solution reduces this time to $O(n^3)$.