Dynamic Programming Exercises. BME 205 Homework #7

Fall 2013
Due Mon 2 Dec 2013
(Last Update: 08:37 PST 15 November 2013)

There is a nice tutorial on dynamic programming at http://www.sbc.su.se/~per/molbioinfo2001/seqali-dyn.html with pointers to an example at http://www.sbc.su.se/~per/molbioinfo2001/dynprog/dynamic.html. It uses the more expensive alignment algorithm that allows arbitrary gap costs for different length gaps, though only an affine gap cost is used. If you want a textbook presentation, one of the best I know is in Chapter 2 of Durbin, Eddy, Krogh, and Mitchison's book *Biological Sequence Analysis*. The book *An Introduction to Bioinformatics Algorithms* by Neil C. Jones and Pavel A. Pevzner is more tutorial, but doesn't cover affine gap costs as well. Chapter 6 is the relevant chapter there.

The version of dynamic programming with affine gap costs that we presented in class used only nearest neighbors, but needed 3 matrices:

$$M_{i,j} = \max \begin{cases} Ic_{i-1,j-1} + \text{subst}(A_i, B_j) \\ M_{i-1,j-1} + \text{subst}(A_i, B_j) \\ Ir_{i-1,j-1} + \text{subst}(A_i, B_j) \end{cases}$$

$$Ic_{i,j} = \max \begin{cases} Ic_{i,j-1} - \text{extend} \\ M_{i,j-1} - \text{open} \\ Ir_{i,j-1} - \text{double\_gap} \end{cases}$$

$$Ir_{i,j} = \max \begin{cases} M_{i-1,j} - \text{open} \\ Ir_{i-1,j} - \text{extend} \\ Ic_{i-1,j} - \text{double\_gap} \end{cases}$$

In Python, one can handle this just as well with one array, containing triples of values—for example, (M,Ir,Ic). Some students find handling one subscripting and a triple of information easier.

Be careful of the boundary conditions. For global alignment, you should have "start"-"start" aligned with 0 cost in the MATCH state and –infinity in Ir and Ic, so that a gap opening penalty is incurred if you start with an insertion or deletion. Before start, all rows and columns are –infinity, and the rest of the start row and column can be inferred from the recurrence relation. **Make sure that your initial conditions are consistent with the recurrence relations—people sometimes get the indices of their matrices reversed, or confuse the Ir and Ic matrices.**

For local alignment, the recurrence relations and the boundary conditions are different:

$$M_{i,j} = \text{subst}(A_i, B_j) + \max \begin{cases} 0 \\ Ic_{i-1,j-1} \\ M_{i-1,j-1} \\ Ir_{i-1,j-1} \end{cases}$$

$$Ic_{i,j} = \max \begin{cases} Ic_{i,j-1} - \text{extend} \\ M_{i,j-1} - \text{open} \\ Ir_{i,j-1} - \text{double\_gap} \end{cases}$$

$$Ir_{i,j} = \max \begin{cases} M_{i-1,j} - \text{open} \\ Ir_{i-1,j} - \text{extend} \\ Ic_{i-1,j} - \text{double\_gap} \end{cases}$$

Since local alignment is allowed to start at any match state (with the 0 choice for previous alignment cost), all rows and columns before the first letters of the sequences can be minus infinity.

You'll need a substitution matrix for this assignment. I've provided BLOSUM62 in the format provided by matblas and used by BLAST.

Q1 (hand computation):

Compute the score for the following **global** alignment using the BLOSUM62 scoring matrix with a gap-opening cost of 12 and a gap-extension cost of 1 and a double-open cost of 2 (all are in 1/2 bit units to be compatible with the BLOSUM62 matrix). Show what you added up to get the score, since the final number alone could easily be incorrect due to a transcription or addition error.

```
--CTNIDDSA--DK-NR-
VLDAM-EEIGEGDDIKMV
```

Note: this is neither the global nor the local optimal alignment.

Q2 (align module):

Create a module (align.py) that contains two classes: `local_aligner` and `global_aligner`. Both classes should have the same methods:

```
aligner= local_aligner(subst,open=,extend=,double=)
aligner= global_aligner(subst,open=,extend=,double=)
```

(the `__init__` method) records the substitution matrix and gap costs for the alignments to follow. I found it useful to store the substitution matrix as a dict, mapping strings of 2 letters to the substitution matrix value for that pair. The `__init__` method should assign reasonable default values for the three gap costs.

```
score_a2m(self,s1,s2)
```

Given two strings that represent an alignment in A2M format, score the alignment with the parameters used when the aligner was created. Use this method to check that the alignments you output score what the dynamic programming algorithm thinks that they should have scored.

```
score=align(row_seq,col_seq)
```

Creates and fills in alignment matrices and stores row_seq and col_seq.

I found it convenient (though slow) to use dict objects for the matrices, mapping (i,j) 2-tuples to the value of the matrix at that coordinate pair. Note that this allows negative indices. You can make things faster by using numpy arrays with normal 0-based indexing of arrays, but you have to be careful about the bounds.

`seq=traceback_col_seq()`
: Treating the letters of the (cached) row_seq as alignment columns, return the col_seq sequence with appropriate lower-case and "-" characters so that it would correctly provide the alignment in an A2M file (col_seq residues that do not align should be lower-case, and there should be a "-" every time a character of row_seq is skipped).

You may implement the recursive definition of the computation of the alignment matrix in any way that works, as long as subalignments are not recomputed. Although dynamic programming is the standard implementation (being fast and fairly simple), some people find it easier to do memoized computation. For example, you might have three methods `m(i,j)`, `ir(i,j)`, and `ic(i,j)` that look up the value in the matrices if it is already there, and otherwise do recursion (or base cases) necessary to compute it. There are many ways to implement the recursive definition, and memoization is not the fastest of them.

Write a program `test_align`

Use your aligner module to write this master-slave alignment program. You should have options

`--subst_matrix BLOSUM62`
: Giving the file or URL to read the substitution matrix from (integer values in BLOSUM format).

`--align local` and `--align local`
: Specify either local or global alignment.

`--open 10`
: Gap opening penalty.

`--extend 2`
: Gap extension penalty.

`--double_gap 3`
: Penalty for moving from gap in one sequence to gap in the other.

The program should read a fasta file from stdin with multiple sequences and output an alignment as an A2M file to stdout with the same sequences in the same order, and with the same names and comments on the id lines.

The fasta files may have *alignment* characters in them, like spaces, periods, and hyphens. These are **not** part of the sequences and should be removed or ignored. The alphabet for the fasta input is taken to be the characters used as indices in the substitution matrix. Whether letters in the fasta file are upper- or lower-case should also be ignored. This means that you should be able to read an A2M file and recover the unaligned sequences in a uniform case. As a sanity check, if you feed the output of your program back into the program with the same alignment options, the output should be identical to the input.

The first sequence of the fasta file is taken to be the "master" sequence and is echoed to stdout. All subsequent sequences are aligned to the first sequence and output to stdout. The score for each alignment should be output to stderr.

The gap penalties should (by default) be open=12, extend=1, double_gap=3. (The open and extend penalties are reasonable, but I'm not so sure about the double_gap penalty.)

The complete output of the program should be a legal A2M file.

```
>test1
CCCCCCCCCCCCCCCADEFGCCCCCCCCCCCCCCCC
>test2
```

```
eeeeeeeeee---------------ADEFG----------------eeeeeeeeeeeeeeee
```

or

```
>test1
CCCCCCCCCCCCCCCADEFGCCCCCCCCCCCCCCCC
>test2
---------------eeeeeeeeeeADEFGeeeeeeeeeeeeeeeee----------------
```

See http://compbio.soe.ucsc.edu/a2m-desc.html for more details of a2m format. You can check that the file is formatted ok, by running it through the SAM program `checkseq`, which unfortunately does not use stdin. To check the file x.a2m, run

```
checkseq foo -d x.a2m
```

You can also use the SAM program prettyalign to convert the a2m into a more visually appealing format. The SAM programs are installed in /projects/compbio/bin/i686 and /projects/compbio/bin/x86_64

Note: Python is **not** the language of choice for computation-intensive applications like sequence alignment. It may be better than Perl, but it is still not very efficient. My memoized implementation on 1001 alignments to a 302-long sequence (input from 1bqcA.fa.gz) which requires about 98.2 million cell updates took 1380 seconds on my laptop, or about 71,000 cell-updates-per-second (CUPS). A reasonable software implementation in an efficient language should do about 30 million CUPS on a somewhat newer machine, and a highly optimized implementation taking advantage of all the hardware on fast modern desktops may manage 46 billion CUPS (http://www.clcbio.com/wp-content/uploads/2012/09/Cell_A413.pdf). Python is required for this assignment only for pedagogic purposes, and my implementation was chosen for simplicity, not speed. Even in Python, one could go much faster (I might try to do a slightly more efficient implementation this year, since the memoized one is ludicrously slow).

Name your program "test_align", so that I can test everyone's program with a script that does `test_align --align=local < test.fasta` and `test_align --align=global < test.fasta` . Turn in your program after testing it on the following fasta files:

- align-1.fasta This example is taken from the textbook, and there are 2 local alignments with the same optimal score.
- align-2.fasta
- align-3.fasta
- align-4.fasta
- align-5.fasta
- align-6.fasta

(I may use other alignment test cases, if I can come up with some good ones for teasing out bugs in traceback.)

Don't waste paper turning in printouts of the alignments, as I'll be recomputing them with your program anyway. I'll be grading based on whether your alignment scores the same as an optimal alignment that my program finds, so it is essential that your output be a legal A2M file that correctly represents the alignment, but it need not be the same optimal alignment that I find.

To help you with your debugging, I provide possible output for local alignment of align-3, with g=12, e=1, d=2:

```
>fat
```

```
ISAYGARFIELDISAFATCAT
>cat
----GARFIELDwasthelASTFATCAT
>cat2
----GARFIELDISAFATCAT
>cat3
----GARFIELDISTheFATCAT
>cat4
----GARFIELDISAFAsTCAT
>cat4
----GARFIELDISTHefaSTCAT
>cat5
----GARFIELDIS-FAT---
>say
-SAYFAt---------------
```

with scores 58, 85, 68, 73, 57, 51, and 16. Possible global alignments are

```
>fat
ISAYGARFIELDISAFATCAT
>cat
----GARFIELDwasthelASTFATCAT
>cat2
----GARFIELDISAFATCAT
>cat3
----GARFIELDISTheFATCAT
>cat4
----GARFIELDISAFAsTCAT
>cat4
----GARFIELDISTHefaSTCAT
>cat5
----GARFIELDIS----FAT
>say
-SAY--------------FAT
```

with scores 43, 70, 53, 58, 42, 25, and -15.

Bonus point—linear gap costs

> Implement linear gap costs and compare running times for affine and linear algorithms. You may need to use longer test sequences to get repeatable timing results, as the expected 3-fold difference in the inner loop may be buried in overhead.

Things learned for next year

SoE home

Biomolecular Engineering Department

BME 205 home page

UCSC Bioinformatics research

Kevin Karplus's
home page

Questions about page content should be directed to Kevin Karplus
Biomolecular Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
USA
karplus@soe.ucsc.edu
1-831-459-4250
318 Physical Sciences Building