Dynamic Programming in Haskell

Christian Neukirchen*

March 2006

The purpose of this Literate Haskell program is to implement a function that does global sequence alignment using Needleman/Wunsch techniques.[1]

The algorithm is based on two steps: first, filling a matrix with the maximal alignment scores for each element and then tracing a path connecting the top-left and the bottom-right cell. Note that the matrix is $O(n \cdot m)$ memory-wise and therefore pretty inefficient, you don't want to use this on bigger sequences.

In Haskell, a good way to implement Dynamic Programming like this is an array that will memoize a lazy stream of scores per cell. This allows $O(1)$-lookup of formerly calculated values without losing referential transparency and (to an extent) lazy evaluation.

```
import Array
```

align is the function that wraps all the functions below and calls them in correct order. It takes two strings and returns them aligned:

```
align          :: String → String → [String]
align da db = format $ reverse $ traceback lena lenb
  where
    lena = length da
    lenb = length db
```

The algorithm is easier to express when the sequences to align are one-indexed, since the borders of the matrix are used as special values. An easy way to achieve this is prepending a space:

```
    a = '␣' : da
    b = '␣' : db
```

*The author can be reached at `http://chneukirchen.org`.

[1] More about these techniques, graphics helpful for understanding, and a codeless step-by-step explanation can be found at `http://www.sbc.su.se/~pjk/molbioinfo2001/dynprog/dynamic.html`.

memscore is the array that contains the actual matrix. It is filled using a lazy stream of scores for each element.

$$\begin{aligned} &\mathit{memscore} = \mathit{listArray}\ ((0,0),(\mathit{lena},\mathit{lenb})) \\ &\qquad\qquad [\mathit{score}\ x\ y \mid x \leftarrow [0\,..\,\mathit{lena}], y \leftarrow [0\,..\,\mathit{lenb}]] \end{aligned}$$

The scoring function looks very confusing since Haskell's array access operator is not very elegant. I'll introduce an infix operator i @@ j that corresponds to $M_{i,j}$:

$$\begin{aligned} &\textbf{infix}\ 5\ @@ \\ &(@@)\ i\ j = \mathit{memscore}\,!\,(i,j) \end{aligned}$$

The score $M_{i,j}$ of each element is determined in below code as follows, the borders of the matrix with $i = 0$ and $j = 0$ are initialized to zero. (More complex scoring algorithms could be added easily.)

$$M_{i,j} = \text{maximum of} \begin{cases} M_{i-1,j-1} + S_{i,j} \\ M_{i,j-1} + w \\ M_{i-1,j} + w \end{cases}$$

The gap penalty w is zero here for reasons of simplicity.

$$\begin{aligned} &\mathit{score}\ 0\ \_ = 0 \\ &\mathit{score}\ \_\ 0 = 0 \\ &\mathit{score}\ x\ y = \mathit{maximum}\ [(x-1\ @@\ y-1) + \mathit{difference}\ x\ y, \\ &\qquad\qquad\qquad\qquad\quad x-1\ @@\ y, \\ &\qquad\qquad\qquad\qquad\quad x\ \ \ \ \ @@\ y-1] \end{aligned}$$

$S_{i,j}$ is a mismatch penalty defined here like this:

$$S_{i,j} = \begin{cases} 0 & \text{if the symbols at position } i \text{ and position } j \text{ match} \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\textbf{where}\ \mathit{difference}\ x\ y \mid a\,!!\,x \equiv b\,!!\,y = 1 \\ &\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise}\ \ \ \ \ \ = 0 \end{aligned}$$

traceback now finds the path connecting both corners of the matrix and collects the appropriate symbols (or spaces for gaps).

$$\begin{aligned} &\mathit{traceback} :: \mathit{Int} \to \mathit{Int} \to [(\mathit{Char},\mathit{Char})] \\ &\mathit{traceback}\ 0\ 0 = [\,] \\ &\mathit{traceback}\ x\ y \mid x \equiv 0 = (\text{'␣'}\ \ , b\,!!\,y) : \mathit{traceback}\ 0\ (y-1) \\ &\qquad\qquad\quad \mid y \equiv 0 = (a\,!!\,x, \text{'␣'}\ \ ) : \mathit{traceback}\ (x-1)\ 0 \\ &\qquad\qquad\quad \mid x\ @@\ y \equiv x\ @@\ y-1 = (\text{'␣'}\ \ , b\,!!\,y) : \mathit{traceback}\ x\ (y-1) \\ &\qquad\qquad\quad \mid x\ @@\ y \equiv x-1\ @@\ y = (a\,!!\,x, \text{'␣'}\ \ ) : \mathit{traceback}\ (x-1)\ y \\ &\qquad\qquad\quad \mid \mathit{otherwise} = (a\,!!\,x, b\,!!\,y) : \mathit{traceback}\ (x-1)\ (y-1) \end{aligned}$$

The resulting list of tuples like $[(\texttt{'a'}, \texttt{'d'}), (\texttt{'b'}, \texttt{'e'}), (\texttt{'c'}, \texttt{'f'})]$ gets converted by *format* into $[\texttt{"abc"}, \texttt{"def"}]$.

$$format\ l = [map\ fst\ l, map\ snd\ l]$$

Finally, a small main program to test the algorithm:

$$dna1 = \texttt{"GAATTCAGTTA"}$$
$$dna2 = \texttt{"GGATCGA"}$$
$$main = mapM\_\ putStrLn\ \$\ align\ dna1\ dna2$$

Expected output:

```
G␣AATTCAGTTA
GGA␣T␣C␣G␣␣A
```

As you can see, corresponding symbols are aligned, with appropriate gaps in between. Implementing more complex rules for alignment is left as an exercise for the reader.

A run where bigger gaps are needed:

$$dna1 = \texttt{"ATGGCTTCTACC"}$$
$$dna2 = \texttt{"TATCAAAAGCCG"}$$

```
␣ATGGCTTCTA␣␣␣␣CC␣
TAT␣␣C␣␣␣␣AAAAGCCG
```