EXTENSIONS TO THE BASIC ALGORITHMS 3.3

The algorithms presented in Section 3.2 are generally adequate for most applications. Sometimes, however, we have a special situation and we need a better algorithm. In this section we study a few techniques that can be employed in some of these cases.

One kind of improvement is related to the problem's computational complexity. We show that it is possible to reduce the space requirements of the algorithms from quadratic (mn) to linear $(m+n)$, at a cost of roughly doubling computation time. On the other hand, we show a way of reducing also the time complexity, but this only works for similar sequences and for a certain family of scoring parameters.

Another improvement has to do with the biological interpretation of alignments. From the biological point of view, it is more realistic to consider a series of consecutive spaces instead of individual spaces. We study variants of the basic algorithms adapted to this point of view.

3.3.1 SAVING SPACE

The quadratic complexity of the basic algorithms makes them unattractive in some applications involving very long sequences or repeated comparison of several sequences. No algorithm is known that uses asymptotically less time and has the same generality, although faster algorithms exist if we restrict ourselves to particular choices of the parameters.

With respect to space, however, it is possible to improve complexity from quadratic to linear and keep the same generality. The price to pay is an increase in processing time, which will roughly double. Nevertheless, the asymptotic time complexity is still the same, and in many cases space and not time is the limiting factor, so this improvement is of great practical value. In this section we describe this elegant space-saving technique.

We begin by noticing that computing $\text{sim}(s, t)$ can be easily done in linear space. Each row of the matrix depends only on the preceding one, and it is possible to perform the calculations keeping only one vector in memory, which will hold partly the new row being computed and partly the previous row. This is done in the code shown in Figure 3.5. Obviously, the same is valid for columns, and if $m < n$, using this trick with the columns uses less space. Notice that at the end of each iteration in the loop on i in Figure 3.5 the vector a contains the similarities between $s[1..i]$ and all prefixes of t. This fact will be used later.

The hard part is to get an optimal alignment in linear space. The algorithm we saw earlier depends on the whole matrix to do its job. To remove this difficulty, we use a *divide and conquer* strategy, that is, we divide the problem into two smaller subproblems and later combine their solutions to obtain a solution for the whole problem.

The key idea is the following. Fix an optimal alignment and a position i in s, and consider what can possibly be matched with $s[i]$ in this alignment. There are only two possibilities:

```
Algorithm BestScore
    input: sequences $s$ and $t$
    output: vector $a$
    $m \leftarrow |s|$
    $n \leftarrow |t|$
    for $j \leftarrow 0$ to $n$ do
        $a[j] \leftarrow j \times g$
    for $i \leftarrow 1$ to $m$ do
        $old \leftarrow a[0]$
        $a[0] \leftarrow i \times g$
        for $j \leftarrow 1$ to $n$ do
            $temp \leftarrow a[j]$
            $a[j] \leftarrow \max(a[j] + g,$
                            $old + p(i, j),$
                            $a[j-1] + g)$
            $old \leftarrow temp$
```

FIGURE 3.5

Algorithm for similarity in linear space. In the end, $a[n]$ contains $\text{sim}(s, t)$.

1. The symbol $t[j]$ will match $s[i]$, for some j in $1..n$.
2. A space between $t[j]$ and $t[j+1]$ will match $s[i]$, for some j in $0..n$.

In the second case the index j varies between 0 and n because there is always one more position for spaces than for symbols in a sequence. We also abused notation when $j = 0$ or $j = n$. What we mean in these cases is that the space will be before $t[1]$ or after $t[n]$, respectively.

Let

$$Optimal\begin{pmatrix} x \\ y \end{pmatrix}$$

denote an optimal alignment between x and y. Every alignment between s and t, optimal or not, satisfies (1) or (2). In particular, our fixed optimal alignment must satisfy one of these as well. If it satisfies (1), to obtain all of it we must concatenate

$$Optimal\begin{pmatrix} s[1..i-1] \\ t[1..j-1] \end{pmatrix} + \begin{matrix} s[i] \\ t[j] \end{matrix} + Optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n] \end{pmatrix}, \tag{3.6}$$

while in case (2) we must concatenate

$$Optimal\begin{pmatrix} s[1..i-1] \\ t[1..j] \end{pmatrix} + \begin{matrix} s[i] \\ - \end{matrix} + Optimal\begin{pmatrix} s[i+1..m] \\ t[j+1..n]) \end{pmatrix}. \tag{3.7}$$

These considerations give us a recursive method to compute an optimal alignment, as long as we can determine, for a given i, which one of the cases (1) or (2) occurs and what is the corresponding value of j.

This can be done as follows. According to Equations (3.6) and (3.7) we need, for fixed i, the similarities between $s[1..i-1]$ and an arbitrary prefix of t, and also the similarities between $s[i+1..m]$ and an arbitrary suffix of t. If we had these values, we could explicitly compute the scores of the j alignments represented in (3.6) and of the $j+1$ alignments represented in (3.7). By choosing the best among these, we will have the information necessary to proceed in the recursion.

As we saw earlier, it is possible to compute in linear space the best scores between a given prefix of s and all prefixes of t (see Figure 3.5). A similar algorithm exists for suffixes. Hence, our problem is almost solved. The only thing left is to decide which value of i to use in each recursive call. The best choice is to pick i as close as possible to the middle of the sequence. The complete code appears in Figure 3.6. In this code, the call

$$\mathit{BestScore}(s[a..i-1], t[c..d], \mathit{pref\text{-}sim})$$

returns in *pref-sim* the similarities between $s[a..i-1]$ and $t[c..j]$ for all j in $c-1..d$. Analogously, the call

$$\mathit{BestScoreRev}(s[i+1..b], t[c..d], \mathit{suff\text{-}sim})$$

returns in *suff-sim* the similarities between $s[i+1..b]$ and $t[j+1..d]$ for all j in $c-1..d$. The call

$$\mathit{Align}(1, m, 1, n, 1, \mathit{len})$$

will return an optimal alignment in the global variables *align-s* and *align-t*, and the size of this alignment in *len*.

One last concern remains: Can the processing time go up too much with these additional calculations? Not really. In fact, the time roughly doubles, as we show below.

Let $T(m,n)$ be the number of times a maximum is computed in the internal loop of *BestScore* or *BestScoreRev* as a result of a call *Align*$(a, b, c, d, \mathit{start}, \mathit{end})$ where $m = b-a+1$ and $n = c-d+1$. It is easy to see that the total processing time will be proportional to $T(m,n)$ plus linear terms due to control and initializations. We claim that $T(m,n) \le 2mn$.

A proof can be developed by induction on m. For $m = 1$ no maximum computations will occur, so obviously $T(1,n) \le 2n$. For $m > 1$ we will have a call to *BestScore* with at most $mn/2$ maximum computations, another such amount for *BestScoreRev*, and two recursive calls to *Align*, producing at most $T(m/2, j)$ and $T(m/2, n-j)$ maximum computations. Adding this all up, we have

$$\begin{aligned} T(m,n) &\le \frac{mn}{2} + \frac{mn}{2} + T(m/2, j) + T(m/2, n-j) \\ &\le mn + mj + m(n-j) \\ &= 2mn, \end{aligned}$$

proving the claim.

3.3.2 GENERAL GAP PENALTY FUNCTIONS

Let us define a **gap** as being a consecutive number $k > 1$ of spaces. It is generally accepted that, when mutations are involved, the occurrence of a gap with k spaces is more probable than the occurrence of k isolated spaces. This is because a gap may be due to a

```
**Algorithm** *Align*
    **input:** sequences $s$ and $t$, indices $a, b, c, d$, start position *start*
    **output:** optimal alignment between $s[a..b]$ and $t[c..d]$ placed in vectors *align-s*
        and *align-t* beginning at position *start* and ending at *end*
    **if** $s[a..b]$ empty **or** $t[c..d]$ empty **then**
        // Base case: $s[a..b]$ empty or $t[c..d]$ empty
        Align the nonempty sequence with spaces
        $end \leftarrow start + \max(|s|, |t|)$
    **else**
        // General case
        $i \leftarrow \lfloor (a+b)/2 \rfloor$
        $BestScore(s[a..(i-1)], t[c..d], pref\text{-}sim)$
        $BestScoreRev(s[(i+1)..b], t[c..d], suff\text{-}sim)$

        $posmax \leftarrow c - 1$
        $typemax \leftarrow$ SPACE
        $vmax \leftarrow pref\text{-}sim[c-1] + g + suff\text{-}sim[c-1]$
        **for** $j \leftarrow c$ **to** $d$ **do**
            **if** $pref\text{-}sim[j-1] + p(i,j) + suff\text{-}sim[j] > vmax$ **then**
                $posmax \leftarrow j$
                $typemax \leftarrow$ SYMBOL
                $vmax \leftarrow pref\text{-}sim[j-1] + p(i,j) + suff\text{-}sim[j]$

            **if** $pref\text{-}sim[j] + g + suff\text{-}sim[j] > vmax$ **then**
                $posmax \leftarrow j$
                $typemax \leftarrow$ SPACE
                $vmax \leftarrow pref\text{-}sim[j] + g + suff\text{-}sim[j]$

        **if** $typemax =$ SPACE **then**
            $Align(a, i-1, c, posmax, start, middle)$
            $align\text{-}s[middle] \leftarrow s[i]$
            $align\text{-}t[middle] \leftarrow$ SPACE
            $Align(i+1, b, posmax+1, d, middle+1, end)$
        **else** // $typemax =$ SYMBOL
            $Align(a, i-1, c, posmax-1, start, middle)$
            $align\text{-}s[middle] \leftarrow s[i]$
            $align\text{-}t[middle] \leftarrow t[posmax]$
            $Align(i+1, b, posmax+1, d, middle+1, end)$
```

FIGURE 3.6

An optimal alignment algorithm that uses linear space.

single mutational event that removed a whole stretch of residues, while separated spaces are most probably due to distinct events, and the occurrence of one event is more common than the occurrence of several events.

Up to now, we have not made any distinction between clustered or isolated spaces. This means that a gap is penalized through a linear function. Denoting by $w(k)$, for $k \geq 1$, the penalty associated with a gap with k spaces, we have