Bioinformatics Algorithms

Lecture Notes

a.a. 2020/2021

Alberto Mosconi
Politecnico di Milano

Contents

1 Definitions

1.1 Alphabet

Let Σ be a finite set of symbols (alternatively called characters), called the alphabet. No assumption is made about the nature of the symbols.

1.2 Strings

A string (or word) over Σ is any finite sequence of symbols from Σ. For example if $\Sigma = \{0, 1\}$ then 01011 is a string over Σ.

The **length of a string *S* is the number of symbols** in s (the length of the sequence) and can be any non-negative integer. It is often denoted as $|S|$.

The **empty string** is the unique string over Σ of length 0, and is denoted as ε.

The **set of all strings over Σ of length *n*** is denoted $\mathbf{\Sigma^n}$. For example , if $\Sigma = 0, 1$, then $\Sigma^2 = \{00, 01, 10, 11\}$. Note that $\Sigma^0 = \{\varepsilon\}$ for any alphabet Σ.

The **set of all strings of any length over Σ** is the *Kleene closure* of Σ and is denoted as $\mathbf{\Sigma^*}$. Also:

$$\Sigma^* = \bigcup_{n \in N \cup \{0\}} \Sigma^n$$

The **set of all non-empty strings over Σ** is denoted by $\mathbf{\Sigma^+}$

1.3 Subsequences

A **subsequence** of S is a string that can be **obtained by deleting any number (from 0 to n) of non-consecutive characters** from S, including S and the empty string too.

1.3.1 Let us count: subsequences

A subsequence of S can be described by a binary string B of n elements, telling us for each position i if the corresponding character is kept in the subsequence (1) or deleted (0).

- S = apple, B = 11010 $\rightarrow$ subsequence = apl

Thus, the **number of subsequences** corresponds to the number of binary strings of length n, hence

$$subsequences = 2^n$$

Notice that if two subsequences produced by different choices are **identical, they will be counted twice**.

1.4 Substrings

A string s is said to be a **substring** (or *factor*) of t if there exist (possibly empty) strings u and v such that $t = usv$.

Given a string t, **suffixes** and **prefixes** are special substrings of t.

1.4.1 Let us count: substrings

Given a string of length n:

- We have 1 substring of length n (starting at 1)
- We have 2 substrings of length *n-1* (starting at 1 and 2)
- We have 3 substrings of length *n-2* (starting at 1,2 and 3)
- ...
- We have $n - i + 1$ substrings of length i (starting at all positions from 1 to $n - i + 1$)
- We have n substrings of length 1 (starting at every position)

Plus, 1 substring of length 0.
This brings the **total number of substrings** of a string of length n to

$$\textit{substrings} = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} + 1$$

1.5 Prefixes and suffixes

A string s is said to be a **prefix** of t if there exists a string u such that $t = su$. If u is nonempty, s is said to be a *proper* prefix of t.
Simmetrically, a string s is said to be a **suffix** of t if there exists a string u such that $t = us$. If u is nonempty, s is said to be a *proper* suffix of t.

1.5.1 Let us count: prefixes and suffixes

The number of all posssible prefixes/suffixes of a string of length n is

$$\textit{prefixes/suffixes} = n + 1$$

n are non-empty, +1 because we include also **the empty string** as prefix/suffix.

1.6 Reverse and palindrome

The **reverse** of a string is a string with the same symbols but in reverse order. For example, if $s = abc$ (where a, b, and c are symbols of the alphabet), then the reverse of s is cba.
A string that is the reverse of itself is called a **palindrome**, which also includes the empty string and all strings of length 1.

1.7 Rotations and permutations

A string $s = uv$ is said to be a **rotation** of t if $t = vu$. For example, if $\Sigma = \{0, 1\}$ the string *0011001* is a rotation of *0100110*, where $u =$ *00110* and $v =$ *01*.

1.7.1 Let us count: rotations

Given a string S, with $|S| = n$, we denote **S=AB**, where **A is a prefix** substring, and **B a suffix** substring.
For every possible pair A, B the string **R=BA is a rotation of S**, including S itself, that is, the case where $A = \varepsilon$ and $B = S$.
Since we have n different non empty suffixes B, the number of rotations is

$$rotations = n$$

It is not *n + 1* because the two suffixes $B = S$ and $B = \varepsilon$ produce exactly the same rotation, given by S itself.

1.7.2 Let us count: permutations

Given a string S, with $|S| = n$, the number of permutations of the characters in S is

$$permutations = n!$$

Notice that the formula does not consider the case of indentical permutations: that is, a string of n identical characters, will have n identical permutations.

2 String comparison

2.1 Hamming distance

The **Hamming distance** between **two strings of equal length** is the **number of positions at which the corresponding symbols are different**.

- "ka*rol*in" and "ka*thr*in" → 3
- 10*1*1*1*01 and 10*0*1*0*01 → 2

With Hamming distance we can formalize *substitutions* in biological sequences - or simply sequencing errors in which the wrong base pair is identified.

2.2 Edit distance

The **edit distance** is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the **minimum number of operations required to transform on string into the other**.
In the *Levenshtein distance* (the most common), edit operations are: **removal**, **insertion**, and **substitution**.
The edit distance between "kitten" and "sitting" is 3. A minimal edit script that transforms the former into the latter is:

- **k**itten → **s**itten (substitute s for k)
- sitt**e**n → sitt**i**n (substitute i for e)
- sittin → sittin**g** (insert g at the end)

The number of solutions (sequences of operations) is infinite.

2.2.1 Computing the edit distance

We denote as $E(S, T)$ the edit distance between two strings S and T.
For every string S, $E(S, \varepsilon) = |S|$, that is:

- the edit distance between a string S and the empty string ε equals the length of S.
- also, $E(\varepsilon, \varepsilon) = 0$: the edit distance between two empty strings is zero.

Let $S = s_1 s_2 ... s_n$ and $T = t_1 t_2 ... t_m$ be two strings:

- we denote as $S(i)$ the prefix of length i of $S \rightarrow S(i) = s_1 s_2 ... s_i$
- likewise, as $T(j)$ the prefix of length j of $T \rightarrow T(j) = t_1 t_2 ... t_j$

Now, to compute the edit distance $E(S,T)$ between two strings S and T, with $|S| = n$ characters, and $|T| = m$ characters, a technique of dynamic programming is used.

We can easily calculate the edit distance of two substrings of S and T, with length i and j, if we have the edit distance of the two "previous" substrings, that is, the substrings of length $i-1$ and $j-1$.

To do so we prepare a matrix with $|S|+1$ rows and $|T|+1$ columns, numbered starting from 0.

The cell(i, j) will contain the value of the edit distance between prefixes S(i) and T(j).

We can fill the first row and first column right away since we know that $\forall i, E(S(i), \varepsilon) = i$ and that $\forall j, E(T(j), \varepsilon) = j$.

	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1				
O	2				
U	3				
S	4				
E	5				

Now, the value of each cell can be calculated with this formula.

$$E(S(i),T(j)) = \min \begin{cases} E(S(i),T(j-1)) + 1 \\ E(S(i-1),T(j)) + 1 \\ E(S(i-1),T(j-1)) \quad \text{if} \quad s_i = t_j \\ E(S(i-1),T(j-1)) + 1 \quad \text{if} \quad s_i \neq t_j \end{cases}$$

That is, **the edit distance between S(i) and T(j) depends on the edit distances computed for their prefixes**.

	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	2
E	5	4	3	3	2

Every cell M(i, j) contains the edit distance between prefixes S(i) and T(j). **Cell M(n, m) contains the edit distance between S(n) and T(m), that is E(S, T)**.

2.2.2 Computational complexity of edit distance

The table has $(n+1)x(m+1)$ cells to be filled, hence

$$\textbf{space complexity} = \mathbf{\Theta(nm)}$$

For every cell, the time required for the computation is constant (i.e. does not depend on the size of input strings), hence, the algorithm performs $(n+1)x(m+1)$ steps of constant time. Thus

$$\textbf{time complexity} = \mathbf{\Theta(nm)}$$

2.2.3 Traceback

Once the table has been built, and the edit distance between the two strings has been computed, one may want to know the exact sequence of operations (insertions, deletions and substitutions) that transform the first string into the second.

To allow this, during the creation of the matrix, every time each cell is filled we **not only save the score value, but also which of the 3 previous cells had the minimum value**.

	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	2
E	5	4	3	3	2

Now, if we **start from the final cell**, we can follow the pointers until we reach the top-left corner and deduce the sequence of edit operations from the type of moves that form the path.

- **Diagonal move**: when the path makes a diagonal move there are two possibilities, depending on whether the characters that identify the start cell match or not:
  - if they are **the same**, **no operation** is applied.
  - if they **don't match**, a **substitution** takes place.
- **Vertical move**: during a vertical move an **insertion** takes place: the current character of the second string is inserted into the first string.
- **Horizontal move**: when a horizontal move occurs a character is **deleted** from the first string.

In the case of the example above the sequence of moves and the corresponding operations are:

1. **E, E $\rightarrow$ M, S:** diagonal move, E is equal to E so **no operation**.

2. **M, S $\rightarrow$ M, U:** vertical move, M and S don't match so **S is inserted in the first string**.
3. **M, U $\rightarrow$ O, O:** diagonal move, M and U don't match so **M is substituted with U**.
4. **O, O $\rightarrow$ H, H:** diagonal move, O is equal to O, so **no operation**.
5. **H, H $\rightarrow$ -, -:** diagonal move, H is equal to H, so **no operation**.

We can shorten this representation like so:

```
S - - U N D A Y
S A T U R D A Y
```

2.2.4 Weighted edit distance

2.3 Longest Common Subsequence

3 Alignment

3.1 Global alignment

3.2 Local alignment

4 Pattern matching

4.1 Exact pattern matching

4.1.1 The naive approach

4.1.2 Knuth-Morris-Pratt algorithm