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

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

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

$$prefixes/suffixes = n + 1$$

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 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" $\rightarrow 3$
- 10*111*01 and 10*010*01 $\rightarrow 2$

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 $\rightarrow$ **s**itten (substitute s for k)
- sitt**e**n $\rightarrow$ sitt**i**n (substitute i for e)
- sittin $\rightarrow$ 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

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

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 $(n+1) \times (m+1)$ steps of constant time. Thus

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

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 → M, S:** diagonal move, E is equal to E so **no operation**.

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

We can shorten this representation like so:

```
H O M - E
H O U S E
```

2.2.4 Computational complexity of backtracking

The number of "backtracking steps" will equal the number of columns of the final alignment. **Each step takes constant time** (just follow a pointer and read the letters needed), so the worst-case number of steps corresponds to the maximum number of columns that one alignment of strings of length $|S| = n$ and $|T| = m$ can have. This means:

$$\textbf{space complexity} = \boldsymbol{\Theta}(\mathbf{n} + \mathbf{m})$$

$$\textbf{time complexity} = \boldsymbol{\Theta}(\mathbf{n} + \mathbf{m})$$

2.2.5 Weighted edit distance

The edit distance algorithm can be generalized by giving different "weights" to the various operations. By doing so a more general formula is obtained:

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

- $\mathbf{w_{id}}$ represents the weight of an insertion or deletion of a character.
- $\mathbf{w(s_i, t_j)}$ represents the weight of substitution of the character s_i with t_j. This means that we can prepare a "weight matrix" with the weight for every pair of characters of the alphabet Σ we use.

2.3 Longest Common Subsequence

We have seen the edit distance as a way to measure the dissimilarity of two strings S and T, now we want to evaluate their similarity. One solution is to find the **longest common subsequence (LCS)** between S and T. The longest this common subsequence is, the more similar the two strings are.

The algorithm to solve this problem, similarly to the edit distance, makes use of dynamic programming principles.

2.3.1 Computing the LCS

We denote *LCS(S, T)* as the length of the longest common subsequence between two strings S and T, having $|S| = n$ and $|T| = m$, and *S(i)* the prefix of length i of S and *T(j)* the prefix of length j of T.

Also, given the empty string ε, for every string S, we have $\mathbf{LCS}(\boldsymbol{\varepsilon}, \mathbf{S}) = \mathbf{0}$.

We can now build a similar table to that in the edit distance algorithm, but changing the formula to calculate the values in each cell.

$$E(S(i), T(j)) = \mathbf{max} \begin{cases} E(S(i), T(j-1)) + \mathbf{0} \\ E(S(i-1), T(j)) + \mathbf{0} \\ E(S(i-1), T(j-1)) + \mathbf{1} & \text{if} \quad s_i = t_j \\ E(S(i-1), T(j-1)) + \mathbf{0} & \text{if} \quad s_i \neq t_j \end{cases}$$

Note that "min" has become "max": we are looking for the LCS of **maximum** length, instead of the **minimum** set of edit operations.

As before, we also keep track of the path as the table is built.

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

In order to reconstruct the actual LCS we again start from the bottom right and move back following the pointers.

We add a letter to the LCS only when a **diagonal move** is made **and the letters match**. In the above example:

- **E, E → M, S:** diagonal move, letters match so LCS is now "E".
- **M, S → M, U:** vertical move, do nothing.
- **M, U → O, O:** diagonal move, letters don't match, do nothing.
- **O, O → H, H:** diagonal move, letters match, LCS becomes "OE".
- **H, H → -, -:** diagonal move, letters match, LCS is now "HOE".

3 Alignment

We have seen two symmetrical approaches to the problem of "comparing sequences": the edit distance measures how different two strings are, and the longest common subsequence, that measures how similar they are. **What if we combine the two approaches into a unique one?**

Idea: given two strings S and T, define a similarity measure where

- *differences* have a negative effect: define **negative scores** for the weight of **edit operations** (insertions, deletions, substitutions).
- *conserved letters* have a positive effect: define **positive scores** for the weight of **matches** between letters of the two strings.

The goal is now to **find the alignment that maximises the score** resulting by the weights just defined.

3.1 Global alignment

The goal of **global alignment** is to **try to align every character in every sequence**. We can find a solution for this problem once again using dynamic programming, in a similar way to the previous algorithms.

The rule for calculating the cell values is a combination of what we have seen so far:

$$E(S(i),T(j)) = \mathbf{max}\begin{cases} E(S(i),T(j-1)) + \mathbf{w_d} \\ E(S(i-1),T(j)) + \mathbf{w_d} \\ E(S(i-1),T(j-1)) + \mathbf{w_m} & \text{if} \quad s_i = t_j \\ E(S(i-1),T(j-1)) + \mathbf{w_s} & \text{if} \quad s_i \neq t_j \end{cases}$$

We are still looking for the "max": we're trying to find the alignment with the maximum score. As before, we can use pointers to keep track of the choice made for each cell.

For the example we set $\mathbf{w_d} = -\mathbf{2}, \mathbf{w_s} = -\mathbf{1}, \mathbf{w_m} = \mathbf{1}$.

	-	**H**	**O**	**M**	**E**
-	0	-2	-4	-6	-8
H	-2	1	-1	-3	-5
O	-4	-1	2	0	-2
U	-6	-3	0	1	-1
S	-8	-5	-2	-1	0
E	-10	-7	-4	-3	0

Backtracking as before we can build the alignment:

```
H O M - E
H O U S E
```

3.1.1 Needleman-Wunsch algorithm

The Needleman-Wunsch is the most famous global alignment algorithm and is exactly the same as above but formalized in a different way.

$$M(i,j) = \mathbf{max} \begin{cases} M(i, j-1) + \mathbf{g} \\ M(i-1, j) + \mathbf{g} \\ M(i-1, j-1) + \boldsymbol{\sigma}(\mathbf{s_i}, \mathbf{t_j}) \end{cases}$$

We no longer distinguish matches/mismatches: the score of aligning s_i and t_j **is already included in the substitution matrix $\boldsymbol{\sigma(s_i, t_j)}$**

Given an alphabet $\boldsymbol{\Sigma}$ **the substitution matrix** is a $|\Sigma| \times |\Sigma|$ matrix that for every pair of characters $a_i, a_j \in \Sigma$ gives us the **"weight" in the alignment of substituting $\mathbf{a_i}$ with $\mathbf{a_j}$**. It has the following properties:

- the matrix is symmetrical, i.e. $\sigma(a_i, a_j) = \sigma(a_j, a_i)$.
- given a_i, the values $\sigma(a_i, a_j)$ for all the characters a_j can be different in the matrix.
- values for "matches" $\sigma(a_i, a_i)$ can be different for every letter a_i.

The simplest substitution matrices are the ones used for DNA/RNA sequences. The scores for matches (positive) and substitutions (negative) do not change with nucleotides. For example:

	A	**C**	**G**	**T**
A	+2	-1	-1	-1
C	-1	+2	-1	-1
G	-1	-1	+2	-1
T	-1	-1	-1	+2

3.2 Local alignment

DNA and the related molecules (RNA, proteins) evolve at different speeds, so in a genome, some regions tend to be more conserved by evolution than others. This means that, in several cases, an **evolutionary relationship can be found only for a part of the sequences** compared, but not for them globally.

This is the problem of local alignment: **find the two substrings, one from S and the other from T, that produce the alignment with the maximum score**.

Notice that: since **negative scores are possible**, if there is no pair of substrings with an alignment with score > 0, then the best solution is trivially given by two empty strings with score 0.

Notice also that if we had only positive or null scores, then the best solution would be the alignment of the two whole strings.

3.2.1 Smith Waterman algorithm

This algorithm was designed explicitely for the comparison of biological sequences and **guarantees to find the best (optimal) solution**, and is based again on the dynamic programming matrix.

We start by building a table with $(n+1) \times (m+1)$ cells as in the global alignment.

Cell (i, j) will contain the score of the alignment of a substring ending at i in S and a substring ending at j in T. From this we can already fill the first row and the first column with zeros since **the best possible score between an empty string and any other string is 0**.

The formula for filling each cell is similar to all the others.

$$M(i,j) = \max \begin{cases} \mathbf{0} \\ M(i,j-1) + g \\ M(i-1,j) + g \\ M(i-1,j-1) + \sigma(s_i, t_j) \end{cases}$$

The main different is that when computing the cell value, **if all the possible alignment extensions lead to a negative score, it's better to "reset" everything and start from 0 again**.

Also, **pointers are not saved for cells with a value of 0**.

In the following example we set $g = -2$ (insertions, deletions), $\sigma(a_i, a_j) = -1$ for every i, j (mismatch), $\sigma(a_k, a_k) = 1$ for every i (match).

	-	**H**	**O**	**M**	**E**
-	0	0	0	0	0
H	0	1	0	0	0
O	0	0	2	0	0
U	0	0	0	1	0
S	0	0	0	0	0
E	0	0	0	0	1

The **traceback** process **doesn't start from the bottom right cell**, but instead it **starts from the cell with the maximum score**, and follows the paths back until a cell with a score of 0, then it stops.

For the above example, the two substrings that produce the highest alignment score are

```
H O
H O
```

Once we have found, traced back, and aligned the best pair of strings, we can iterate the process finding the "second best" score in cells that haven't been used previously. And then the third, fourth, and so on (remember that the algorithm can be applied to sequences of hundreds or thousands of nucleotides/aminoacids).

4 Pattern matching

4.1 Exact pattern matching

4.1.1 The naive approach

4.1.2 Knuth-Morris-Pratt algorithm