31

Dynamic Programming – Evolutionary Distance

Norbert Blum and Matthias Kretschmer

Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, Germany

150 years ago parts of the skeleton of the so-called Neanderthal man were found near Düsseldorf (Germany) for the first time. Since then we've wanted to know how Homo sapiens, our ancestor, and the Neanderthal man are related. Today we know that Homo neanderthalensis is not an ancestor of Homo sapiens, and vice versa. Differences in the genotype of Homo sapiens and Homo neanderthalensis proved that. New technologies allow us to extract genotypes from bones that are more than 30,000 years old. The genotypes are extracted in the form of DNA sequences. These DNA sequences are like construction plans of animals and humans. In the course of time, DNA sequences change through mutations. Given the DNA sequences of different species, one can calculate their similarity with the help of a computer. We measure the similarity of two sequences by specifying a distance between them. A small distance of two DNA sequences indicates a high similarity of both sequences. How do we calculate the distance of two DNA sequences? We will show how to develop an algorithm for solving this problem. To do this, we need a mathematical model of DNA sequences, mutations and the distance between two DNA sequences.

Mathematical Modeling

A DNA sequence consists of bases. A sequence of three bases encodes an amino acid. There exist four different bases which are represented by the letters A, G, C and T. Hence, a DNA sequence may be represented by a string over the alphabet $\Sigma = \{A, G, C, T\}$. For example,

$$CAGCGGAAGGTCACGGCCGGGCCTAGCGCCTCAGGGGTG$$

is a part of the DNA sequence of the chicken.

In nature DNA sequences change through mutations. A *mutation* can be considered as a mapping from a DNA sequence x to a DNA sequence y. We assume that all mutations are modeled using the following three types of basic mutations:

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_31,

1. deletion of a character,
2. insertion of a character, and
3. substitution of a character with another character.

For example, let $x = AGCT$ be a DNA sequence. Then the mutation substitute G by C would mutate x to the sequence $y = ACCT$. We use the notation $a \to b$ to represent the substitution of a by b. $a \to$ represents the deletion of character a and $\to b$ is the insertion of character b. The position where the mutation has to be performed is explicitly given by the algorithm.

To measure the distance of two DNA sequences, we give every basic mutation a specific cost. The mutation s has cost $c(s)$. The cost of a mutation corresponds to the probability of that mutation. The more likely a mutation is the lower the cost. We use the following costs for the three basic mutations:

- deletion: 2
- insertion: 2
- substitution: 3

To compare two DNA sequences x and y, we require in most cases more than one basic mutation to transform x to y. For example, if we want to compare $x = AG$ and $y = T$, then one basic mutation would not be sufficient. But we could perform the transformation using the sequence of mutations $S = A \to, G \to T$ (delete A and substitute G by T). The cost $c(S)$ of a sequence of mutations $S = s_1, \ldots, s_t$ is the sum of the costs of its basic mutations, i.e.,

$$c(S) := c(s_1) + \cdots + c(s_t).$$

The distance of two DNA sequences is defined by the cost of a specific sequence of mutations which transform one to the other. The problem is that there might be many different sequences of mutations that transform one DNA sequence to another. For example, we could use the following mutation sequences to transform the DNA sequence $x = AG$ to $y = T$:

- $S_1 = A \to, G \to T$; $c(S_1) = c(A \to) + c(G \to T) = 2 + 3 = 5$
- $S_2 = A \to T, G \to$; $c(S_2) = c(A \to T) + c(G \to) = 3 + 2 = 5$
- $S_3 = A \to, G \to, \to T$; $c(S_3) = c(A \to) + c(G \to) + c(\to T) = 2 + 2 + 2 = 6$
- $S_4 = A \to C, G \to, C \to, \to T$;
  $c(S_4) = c(A \to C) + c(G \to) + c(C \to) + c(\to T) = 3 + 2 + 2 + 2 = 9$

There exist many other sequences that transform x to y, but none of them have lower cost than the sequences S_1 and S_2. To define the distance of two DNA sequences, we use the sequence of mutations with the lowest cost. Given a cost function c, the distance $d_c(x, y)$ of the DNA sequences x and y is defined by

$$d_c(x, y) := \min\{c(S) \mid S \text{ transforms } x \text{ to } y\}.$$

For example, the sequences S_1 and S_2 are the sequences of mutations with the lowest cost that transform $x = AG$ to $y = T$. Hence, the evolutionary distance $d_c(x, y)$ of x and y is five.

Calculation of the Evolutionary Distance

How to calculate the evolutionary distance $d_c(x, y)$? We only allow the basic mutations deletion, insertion and substitution to transform x to y. The definition of the basic mutations and of their costs are in such a way that multiple mutations at the same position can be replaced by a single mutation with lower cost. For example, the deletion of the character A and the insertion of the character B can be replaced by the substitution of A by B which has lower cost (cost 3 for the substitution and $2 + 2 = 4$ for the deletion and insertion). Operations at different positions do not depend upon each other. Hence, we can perform the mutations in any order. Assume that the last mutation will always be performed at the last position of both DNA sequences. Let $x = a_1a_2 \dots a_m$ and $y = b_1b_2 \dots b_n$ two DNA sequences; i.e., x consists of m and y of n characters. By the definition of our three mutations, we have the following three possibilities for the last mutation:

1. *Deletion*: Transform $a_1a_2 \dots a_{m-1}$ to $b_1b_2 \dots b_n$ and then delete a_m.
2. *Insertion*: Transform $a_1a_2 \dots a_m$ to $b_1b_2 \dots b_{n-1}$ and then insert b_n.
3. *Substitution*: Transform $a_1a_2 \dots a_{m-1}$ to $b_1b_2 \dots b_{n-1}$ and then substitute a_m by b_n.

For the calculation of the evolutionary distance, we only need to consider the last mutation with the lowest cost.

We develop our algorithm by using the scheme above. Let $x[i]$ be the sequence consisting of the first i characters of x. This sequence is called the *prefix of length* i of x. The prefix of length 0 of x is the empty sequence $x[0]$. The sequence of length m of x is x itself (x consists of m characters). Analogously, $y[j]$ denotes the prefix of length j of y. Now we can reformulate the three possible last mutations:

1. Transform $x[m-1]$ to y and then delete a_m.
2. Transform x to $y[n-1]$ and then insert b_n.
3. Transform $x[m-1]$ to $y[n-1]$ and then substitute a_m by b_n.

It remains to solve the following problem: How to transform $x[m-1]$ to y, x to $y[n-1]$ and $x[m-1]$ to $y[n-1]$? For these three transformations, we can just apply the same scheme. To calculate the evolutionary distance $d_c(x[i], y[j])$ of the prefix of length i of x and the prefix of length j of y, we use the following scheme:

$$d_c(x[i], y[j]) := \min \begin{cases} d_c(x[i-1], y[j]) + c(a_i \rightarrow) & \text{(deletion)}, \\ d_c(x[i], y[j-1]) + c(\rightarrow b_j) & \text{(insertion)}, \\ d_c(x[i-1], y[j-1]) + c(a_i \rightarrow b_j) & \text{(substitution)}. \end{cases}$$

Hence, we require the distances $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ and $d_c(x[i-1], y[j-1])$ to calculate the distance $d_c(x[i], y[j])$.

If one of the prefixes has length zero then not all of the three mutations can be performed. We cannot delete a character from or substitute a character into the empty string. To transform $x[i]$ to the empty string $y[0]$ with minimum cost, we will never insert or substitute a character. Each character that is inserted or substituted has to be deleted, thus we can omit the insert and substitute mutations and get a sequence of mutations with lower cost. The lowest cost transformation from $x[i]$ to $y[0]$ is thus the sequence of basic mutations which consists only of deletions. Similarly, in the case of the transformation from $x[0]$ to $y[j]$, the lowest cost transformation is to insert the characters of the string $y[j]$. Hence, for $i = 0$ and $j > 0$ we perform only insertions and for $i > 0$ and $j = 0$ we perform only deletions. The cost of the sequences of mutations in the case of $i = 0$ and $j > 0$ is $2 \cdot j$ and in the case of $i > 0$ and $j = 0$ is $2 \cdot i$. If both i and j are zero then we have to transform the empty string to the empty string. Of course, we do not need any mutation for this transformation. Hence, the cost $d_c(x[0], y[0])$ is zero.

For example, consider the two DNA sequences $x = AGT$ and $y = CAT$. Assume that we know the distances $d_c(x[1], y[2]) = d_c(A, CA)$, $d_c(x[2], y[1]) = d_c(AG, C)$ and $d_c(x[1], y[1]) = d_c(A, C)$. Then we can use the scheme above to get the evolutionary distance $d_c(x[2], y[2]) = d_c(AG, CA)$ by calculating the minimum of

- $d_c(x[1], y[2]) + c(a_2 \rightarrow) = d_c(A, CA) + c(G \rightarrow) = d_c(A, CA) + 2$,
- $d_c(x[2], y[1]) + c(\rightarrow b_2) = d_c(AG, C) + c(\rightarrow A) = d_c(AG, C) + 2$, and
- $d_c(x[1], y[1]) + c(a_2 \rightarrow b_2) = d_c(A, C) + c(G \rightarrow A) = d_c(A, C) + 3$.

The minimum of these three cases is the evolutionary distance $d_c(AG, CA)$.

The Algorithm

How to get an algorithm from this scheme? The distances of most prefixes of x and y are required multiple times. For example, the distance $d_c(x[i-1], y[j-1])$ is required to calculate $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ and $d_c(x[i], y[j])$. To save time, we only want to calculate $d_c(x[i-1], y[j-1])$ once. Hence, we have to keep this value to use it multiple times without recalculation. To do this, we use a table in which we store all previous calculated evolutionary distances of prefixes of x and y. We store in the cell (i, j) (row i and column j) of the table the value of the evolutionary distance $d_c(x[i], y[j])$. The advantage of this method is that we only need to calculate the distances of prefixes once and can use a simple table lookup operation to get the value again. We require the evolutionary distance for all $0 \leq i \leq m$ and $0 \leq j \leq n$ to calculate the distance of the DNA sequences x and y. Thus the table consists of $m+1$ rows and $n+1$ columns.

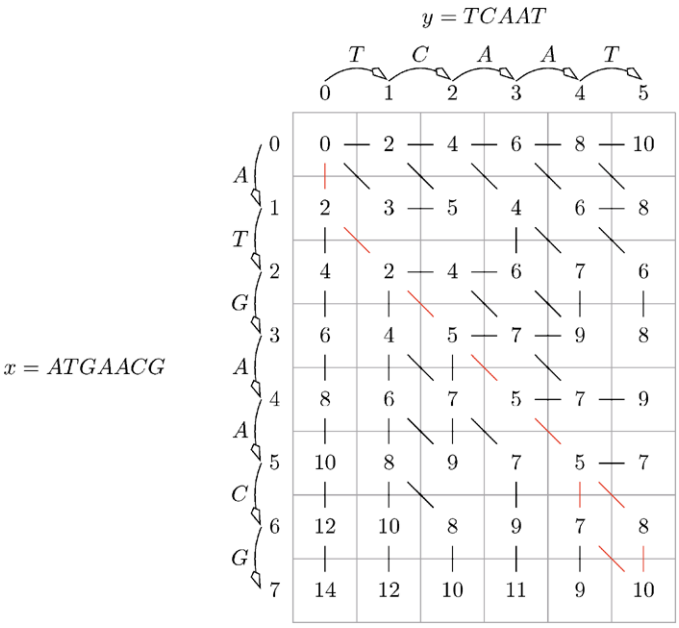

Fig. 31.1. Table for the calculation of the evolutionary distance of $x = ATGAACG$ and $y = TCAAT$

For example, let $x = ATGAACG$ and $y = TCAAT$. The corresponding table for the calculation of the evolutionary distance is given in Fig. 31.1.

We start by calculating the distance of $d_c(x[0], y[0])$ and storing the value in cell $(0, 0)$. As mentioned above, we know that this distance is always zero. We already know the values of the entries in Column 0 and in Row 0. In Column 0 we only perform deletions and in Row 0 only insertions. Hence, we store the values $2, 4, 6, \ldots$ in this column and this row.

We use the developed scheme to calculate the values of the other cells. For example, consider the cell $(2, 1)$. The cell represents the evolutionary distance of $x[2] = AT$ and $y[1] = T$. The deletion of A is intuitively the only mutation of minimum cost to transform $x[2]$ to $y[1]$. The algorithm has to calculate the same distance. It chooses one of the following possible mutations:

1. $d_c(x[1], y[1]) + c(a_2 \rightarrow) = d_c(A, T) + c(a_2 \rightarrow) = 3 + 2 = 5$ (deletion of T),
2. $d_c(x[2], y[0]) + c(\rightarrow b_1) = d_c(AT, y[0]) + c(\rightarrow b_1) = 4 + 2 = 6$ (insertion of T) and
3. $d_c(x[1], y[0]) + c(a_2 \rightarrow b_1) = d_c(A, y[0]) + c(a_2 \rightarrow b_1) = 2 + 0 = 2$ (substitution of T by T).

The substitution of T by T is no mutation. We use this to keep the notation simple. In reality we do not substitute T by T. Hence, the cost for this operation is zero. The deletion of A is not explicitly given in this step of the algorithm. This mutation is performed during the transformation of $x[1] = A$

to $y[0]$. Since the algorithm will choose the third possible mutation in the last step, it will generate the solution which we have intuitively generated.

The lines in the table that form a path from cell $(0,0)$ to a cell (i,j), represent the possible transformations from $x[i]$ to $y[j]$. In the case of $(2,1)$, we used the path over $(1,0)$ by first deleting A and then substituting T by T. The table shows that there might be multiple paths to a cell (i,j). Hence, there might exist multiple different optimal sequences of mutations to transform x to y. The lines can be calculated by the algorithm, as these just represent the case that lead to a minimum distance in a single step. The red colored lines constitute the path of minimum cost for the transformation of x to y. Hence, these represent the sequences of mutations that transform x to y with minimum cost and which can be used to get the evolutionary distance.

We do not know which cells we require for the calculation of a path of minimum cost to transform x to y. Thus, we have to calculate the values of all cells. To calculate the value $d_c(x[i], y[j])$ of cell (i,j), we need the values of the cells $(i-1,j)$, $(i,j-1)$ and $(i-1,j-1)$. To make sure that we have already calculated these values, we generate the table row by row or column by column. If we do it row by row, we need to go through the rows from the left to the right. Similarly, if we generate the table column by column, we need to go through the columns from the top to the bottom. This ensures that all required values are stored in the table, when we calculate the distance $d_c(x[i], y[j])$. At the end, the evolutionary distance $d_c(x[m], y[n]) = d_c(x, y)$ is stored in the cell (m,n).

Conclusion

Starting with the smallest subproblem, the calculation of $d_c(x[0], y[0])$, we have solved larger and larger subproblems. In each step we have increased the lengths of the prefixes of x and y and calculated their evolutionary distances. For the calculation of the distance $d_c(x[i], y[j])$ we have used the distances $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ and $d_c(x[i-1], y[j-1])$. Hence, we have used the optimal solution of these smaller subproblems to solve the larger subproblem.

Given a problem, the calculation of a solution of minimum cost is called an *optimization problem*. The calculation of the evolutionary distance of two DNA sequences is such an optimization problem. We have used a specific technique to create an algorithm for our optimization problem. This generic technique may be applied to other but not all optimization problems. Implicitly, we have used the following property of our optimization problem:

- Every subsolution of an optimal solution which is a solution for a subproblem is an optimal solution for that subproblem.

Many optimization problems have this property. The technique we have used for solving the problem of finding the evolutionary distance may be applied

to any problem with this property. This technique is called *dynamic programming*. In the case of dynamic programming, we split the problem into subproblems. We solve the smallest subproblems directly. In our case, this is the calculation of $d_c(x[0], y[0])$. The solution for this subproblem is zero. From the optimal solutions of small subproblems we calculate the optimal solutions of larger subproblems. We repeat this until we have calculated the optimal solution of the original problem. Dynamic programming is an important generic technique that is often used for the development of algorithms.

The algorithm for calculating the minimum evolutionary distance of two DNA sequences can be used for other purposes than calculating the similarity of two species. For example, one can use it to measure the similarity of two words. This can be useful for spellchecking software. The correct spelling of a word has most probably a very small distance to the incorrectly spelled word given by the user. So the software may show the user all words within a given maximum distance as possible correct spellings of the given word.

References

The references below provide a generic introduction to dynamic programming. They present the theoretical background and also examples of its application.

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. Chapter 3.
2. Jon Kleinberg, Éva Tardos: *Algorithm Design*. Addison-Wesley, 2005. Chapter 6.