Report

Advanced algorithmics and programming

Project - Edit distance

- Hugo Thiollière
- Clément Colin
- Christopher Jeamme
- Dimitri Bruyère

Table of contents

- 1. Introduction
- 2. Algorithms
  - 2.1. Classic
    - 2.1.1 Functionnement
    - 2.1.2 Complexity
  - 2.2. Pure recursive
    - 2.2.1 Functionnement
    - 2.2.2 Complexity
  - 2.3. Branch and Bound
    - 2.3.1 Functionnement
    - 2.3.2 Complexity
  - 2.4. Divide and Conquer
    - 2.5.1 Functionnement
  - 2.5. Classic with approximation
    - 2.5.1 Functionnement
    - 2.5.2 Time and space complexity
  - 2.6. Greedy
    - 2.6.1 Functionnement
    - 2.6.2 Complexity
- 3. Evaluation protocol
  - 3.1 Random string generator
  - 3.2 Results
- 4. Organisation

1. Introduction

For this project, we had to implement algorithms to solve the edit distance problem. Specifically, we had to implement the following ones : * Classic * Recursive * Branch and bound * Divide and conquer * An approximation of the classic * Greedy

After that, we had to evaluate these algorithms, in order to compare their performance. This is possible thanks to the creation of random strings. We also have at our disposition a protein database to evalute our algorithms.

The application is available online here

(full address : https://dimitribruyere.github.io/X12B/X12B_pde/)

2. Algorithms

2.1. Classic

2.1.1 Functionnement

Basically, we fill a matrix with one string vertically and one horizontally.

The first line and column are set sequentially starting with 0, by steps of 1. Then, we fill the matrix one cell at a time, starting with the top-left one.

To fill a cell, if both characters are the same, we take the same value as the one on the top-left, else we take the minimum value of : - the one above plus one - the one on the left plus one - the upper-left one plus one.

In order to get the backtrace, we fill at the same time another matrix of the same size with the origin of the cell.

Finally, we have the cost of the edit distance by taking the bottom-right cell, and we compute the backtracking thanks to the other matrix.

2.1.2 Complexity

We use and fill a matrix of size of n*m, so the time and space complexity :

$$O(n * m)$$

2.2. Pure recursive

2.2.1 Functionnement

The functionnement of this algorithm is simple. On each recurrent call, if the first letter of both strings are the same, we call the algorithm with the first letter deleted in both string, else we add 1 to the edit distance and we add the minimum of the three calls with the suppression, the substitution and the addition. If one of the strings is empty, we return the size of the remainder of the second string in the edit distance.

2.2.2 Complexity

$$O(3^{max\{n,m\}})$$

2.3. Branch and Bound

2.3.1 Functionnement

The branch and bound algorithm is base on the same principle than the pure recursive algorithm. The difference is that when we first find a solution, we then use this knowledge to decide whether or not we should explore a branch. We explore it only if we have a chance of getting a better solution. For that, we use an heuristic. At each call, the heuristic is equal to the length difference between the 2 strings (because it will be the minimal edit distance if maximum of letters match). It is an optimistic heuristic. If the sum of the path we used to get the current node plus the heuristic is not smaller than the solution we found so far, we stop exploring the branch.

2.3.2 Complexity

The complexity of this algorithm depends on whether we found a good solution early or not. If the solution is found early and the heuristic permits to stop the exploration of many branches, the algorithm will be efficient. Else, we could need the entire exploration of the tree and so the complexity will be equals to the pure recursive one.

2.4. Divide and Conquer

2.5.1 Functionnement

The main idea behind this algorithm is to recursively cut the second string into two halves and to find where the first one should be cut to get smaller problems. When the problems are small enough we can call the classic approach to find the edit distance between two very small strings. When coming back from the recursive calls, we combine all the alignments into one.

2.5. Classic with approximation

2.5.1 Functionnement

This algorithm looks like the classic one, but it has an additional feature. In order to save time and space, we will only explore a diagonal set with the Bresenham algorithm. We can extend this diagonal by a factor K (above and below) if we want to. But this gain of space and time has a cost : the edit distance found is not always optimal, since we don't explore all the possible pathes for a small K.

2.5.2 Time and space complexity

The space complexity is equal to the size of the diagonal times *(2 k + 1)*. With a big K, the time and space complexity are the same as the classic algorithm.

$$O(n * m)$$

2.6. Greedy

2.6.1 Functionnement

A greedy algorithm is an algorithm that takes the best solution at each step. If the two characters are the same, we simply move forward one character on each string, else we substitute one character at a time until we meet the end of a string. Then, we add to the cost all remaining letters of the non-empty string.

2.6.2 Complexity

We only parse the smaller string, then we add the length difference between the two strings. So, the time complexity is :

$$O(min(n, m))$$

3. Evaluation protocol

3.1 Random string generator

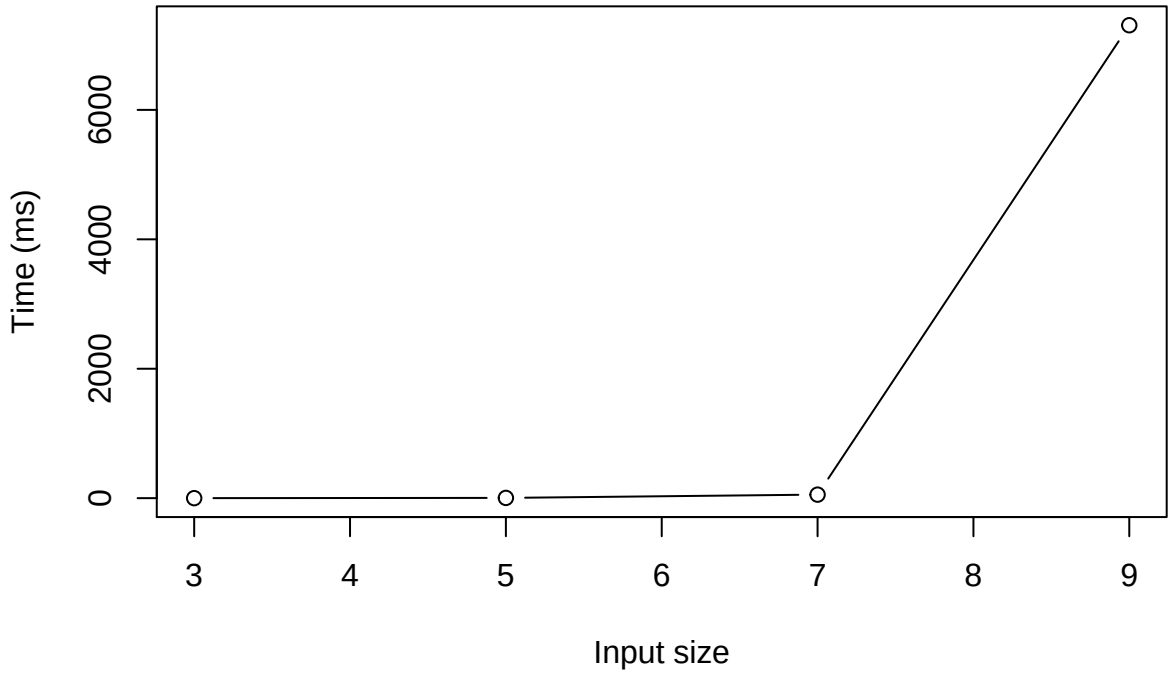
To test the different algorithms, we created a generator of random strings. The generator asks you the number of strings to generate, the length of the first string, and the percentage of similarity between the 2 strings.

3.2 Results

Recursive :

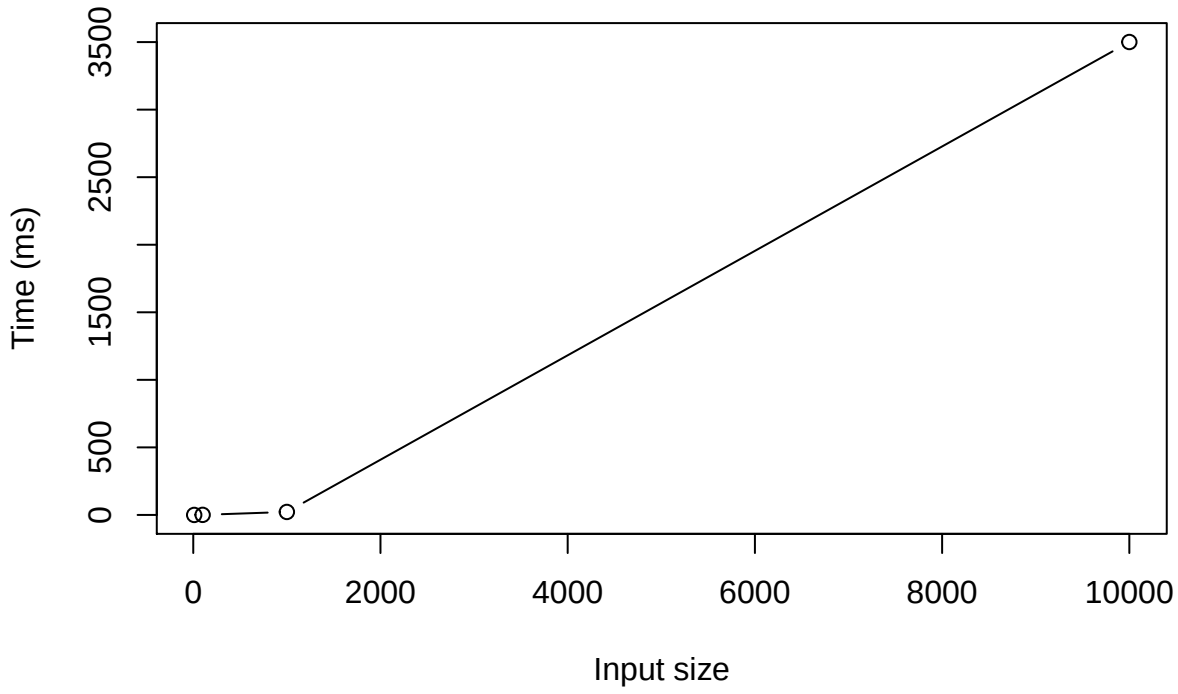
Similarity 20%

```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/recursiveTimes20.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



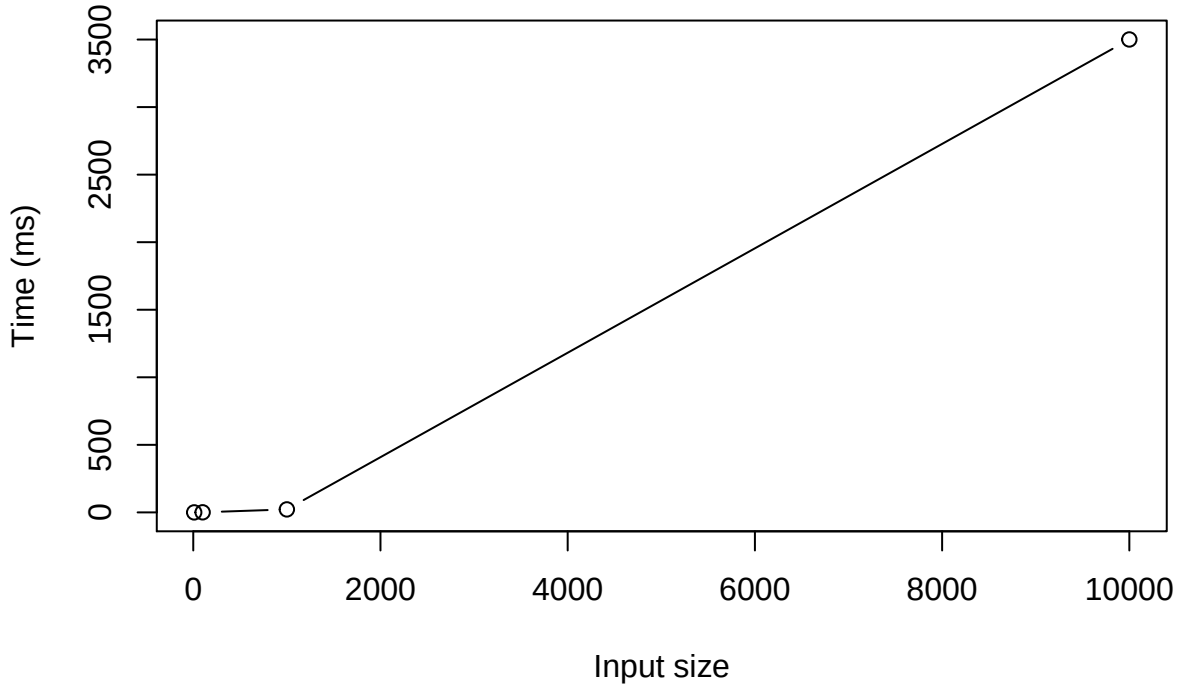
Similarity 80%

```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/recursiveTimes80.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



Classic

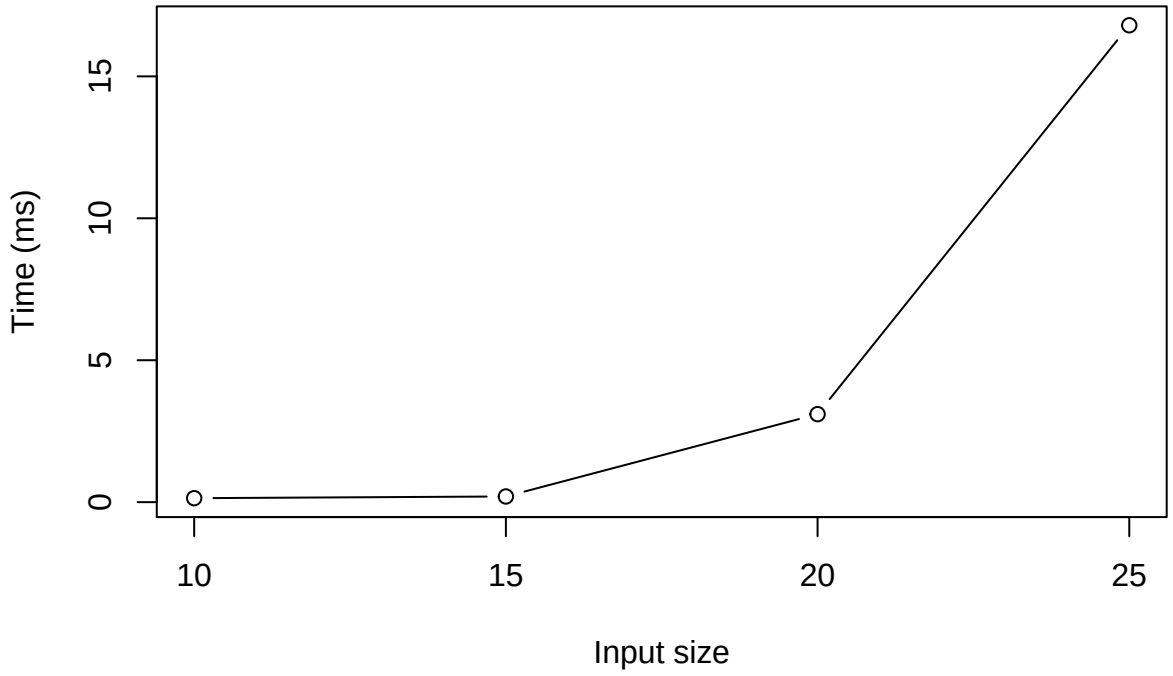
```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/classicTimes20.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



Branch and bound

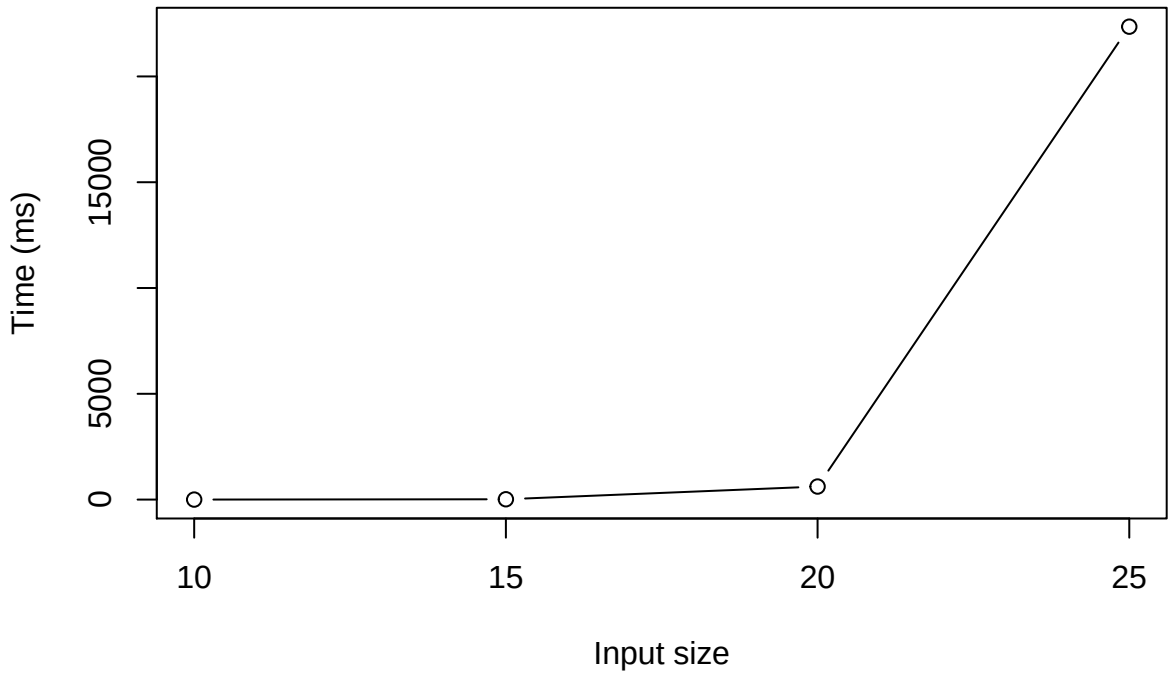
Similarity 10%

```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/branch_boundTimes10.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



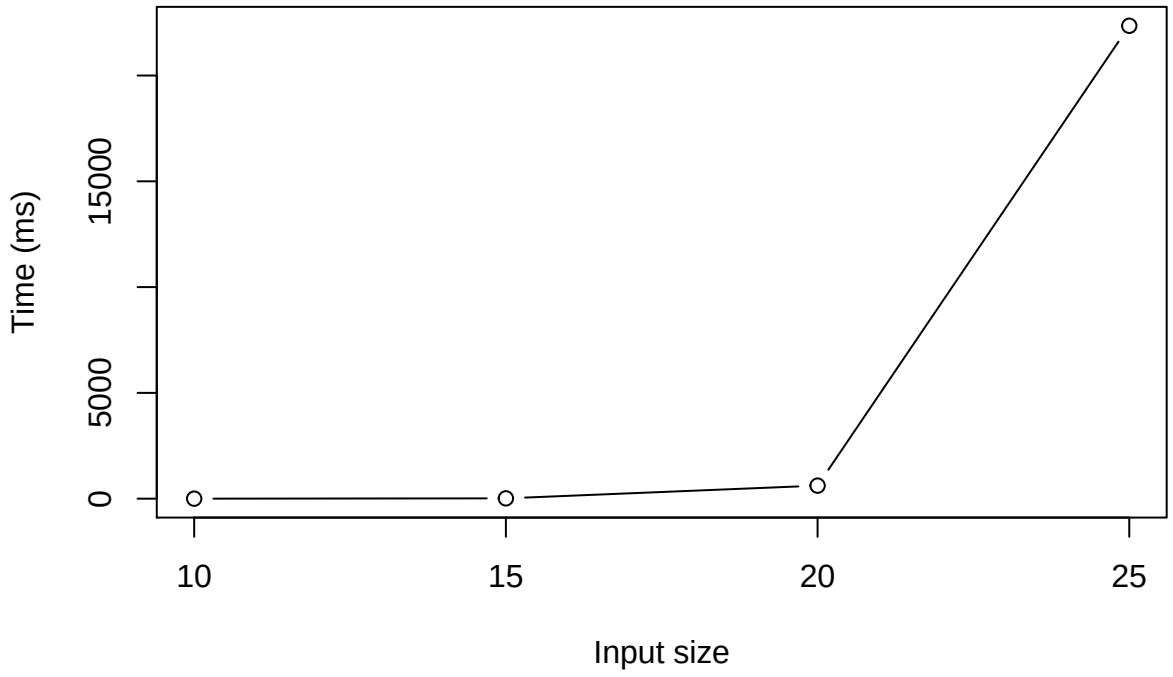
Similarity 20%

```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/branch_boundTimes20.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



Similarity 80%

```
dataTempsCalcul = read.csv("~/Fac/M1/Advanced_algo/Project/X12B/X12B_pde/branch_boundTimes80.csv")
plot(dataTempsCalcul$taille,dataTempsCalcul$temps, type="b", xlab = "Input size", ylab = "Time (ms)")
```



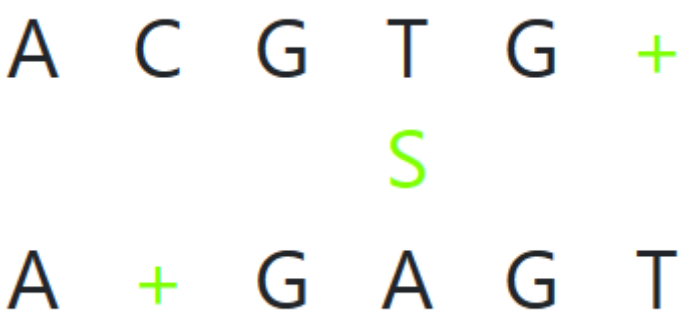


Figure 1: image1

4 User interface

Here is how we made the alignment

We also showed the matrix in the classical version

The diagonal is shown here in the approximative classical method

5. Organisation

Before each work session, we took 5 minutes to discuss the goals of the session. In the end, we realise we should have taken a more global approach and established a more precise planning.
We still managed to share the workload evenly between us. Also, we used `git` because it is a tool we cannot not use when we are working on a project.

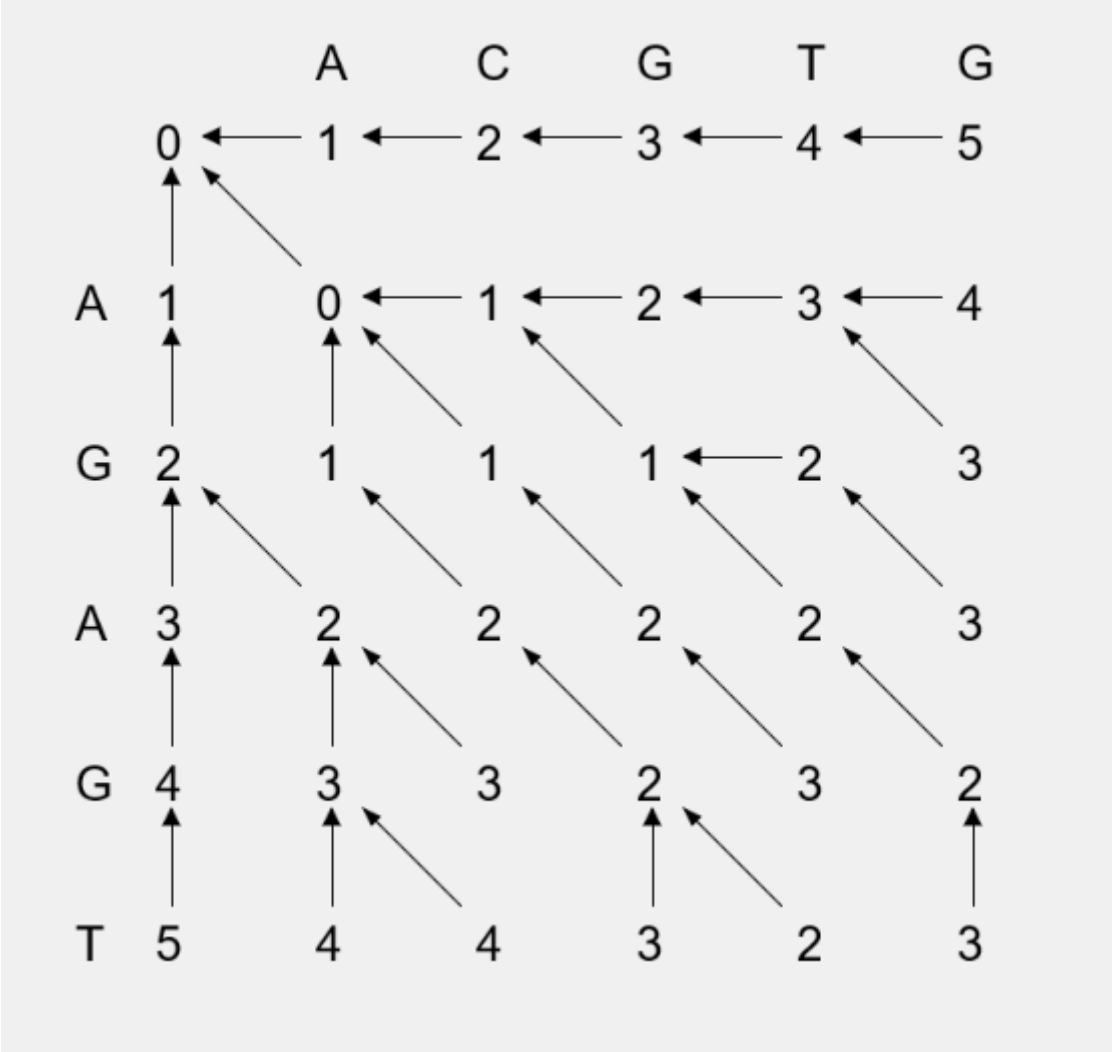


Figure 2: image2

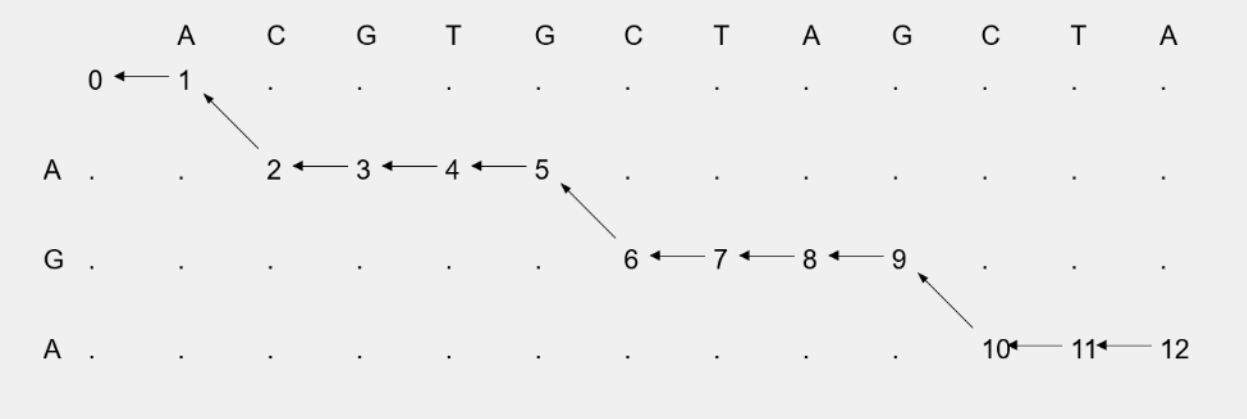


Figure 3: image3

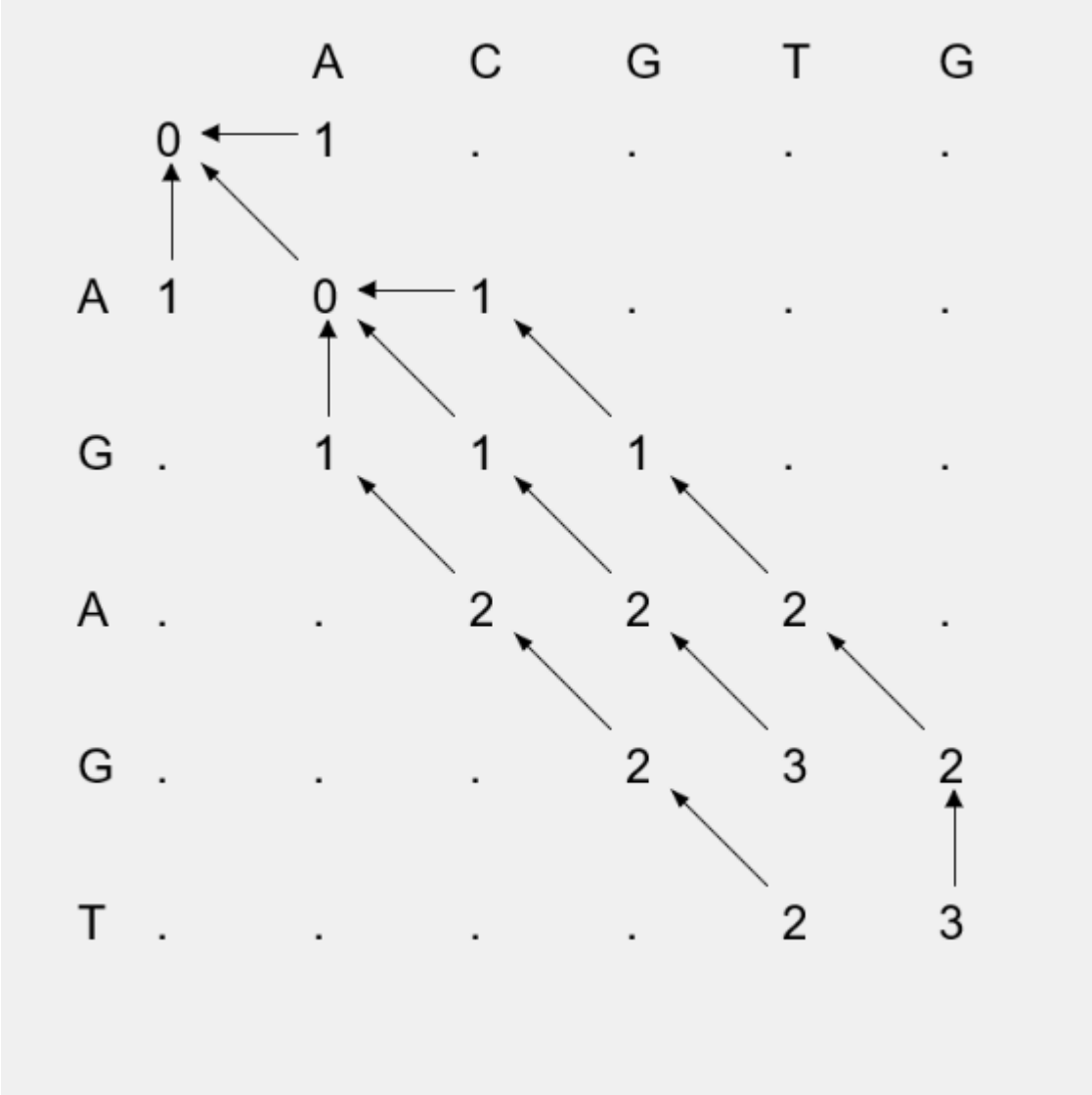


Figure 4: image4

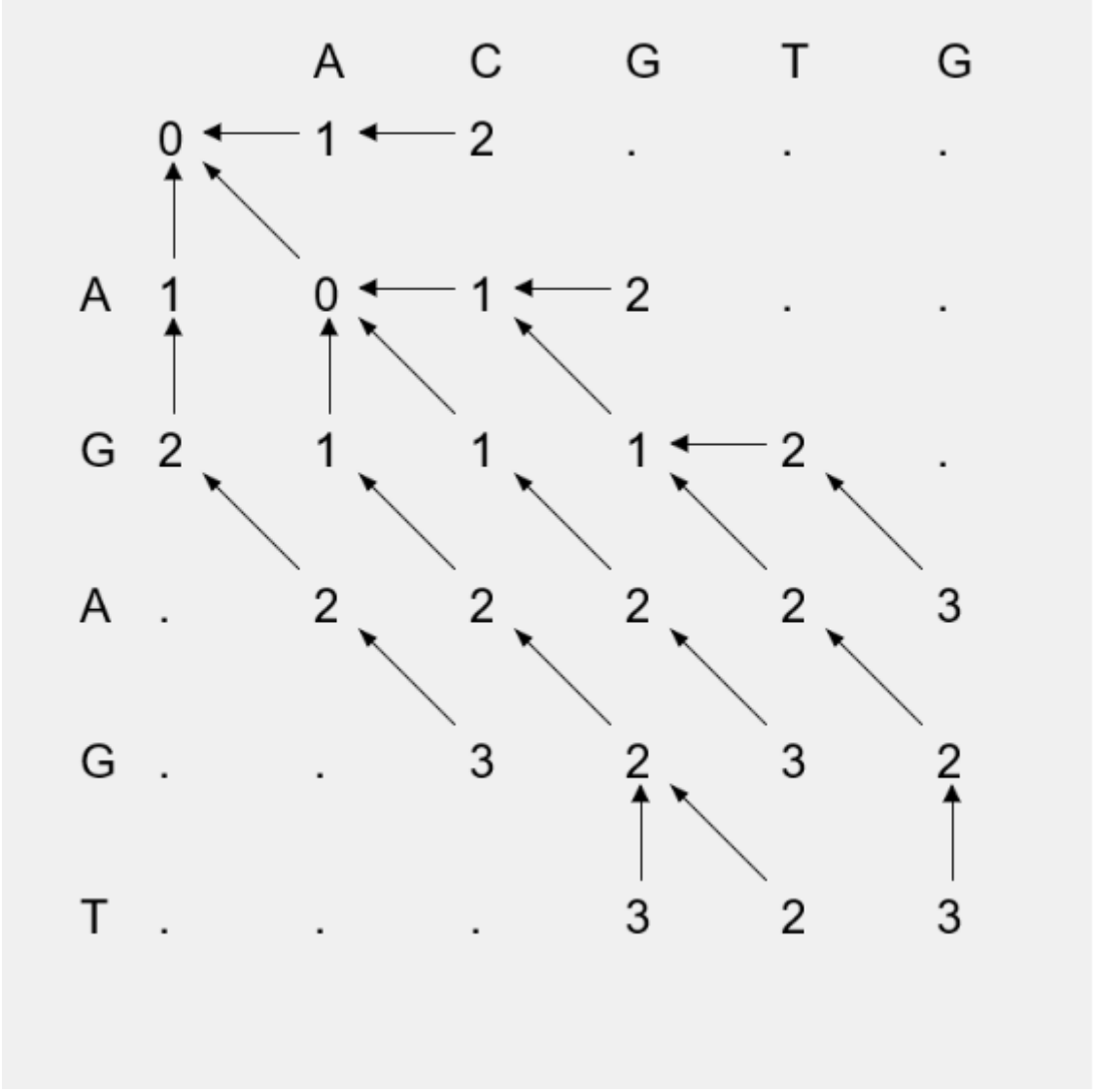


Figure 5: image5