CS 594: Representations in Algorithm Design **Spring 2022**

Lecture 18: March 10, 2022

Lecturer: Xiaorui Sun *Scribe: Tanmay Kelkar*

1 Last time and today

Previously:

- Solutions for special cases of graph representations can be generalized by approximation, decomposition and sparsification.

Starting today:

- Computing edit distance for strings using exact algorithms (eg: DP) and approximation algorithms.

2 Computing Edit Distance for strings

Let us assume that we have two strings X and Y, where X, Y consist of characters from the alphabet. The edit distance is the minimum number of basic operations required to change X into Y (or Y into X). The types of operations that we can make are:

- Adding a character to X
- Deleting a character from X
- Replacing a character of X by another character

Example: X: 'CTACCG'
Y: 'TACATG'

We can perform the following operations to change X into Y:

1. Delete the first character. We get: 'TACCG'
2. Add 'A' after the third character. We get: 'TACAT'

3. Add 'T' after the fourth character. We get: 'TACATG', which is Y.

Therefore, the edit distance is 3.

The standard approach to solving this problem is through dynamic programming.

2.1 Dynamic Programming solution

Let us assume that d(i,j) is the edit distance between substrings of the first 'i' characters in X and the first 'j' characters in Y.
If $|X| = |Y| = n$, then d(n,n) is the edit distance between X and Y.

Edit distance $\approx$ n-size of largest alignment (because the largest alignment is less likely to change or be deleted).
$\Rightarrow$ Edit distance $\leq 2 *$ n-size of largest alignment

As we can see from the previous example, the substring corresponding to X_i=3 is

'CTA' and for Y_i=2 is 'TA'. This corresponds to an edit distance d(3,2). The following possibilities arise with this kind of matching:

1. X_i is matched with Y_i
2. X_i is removed
3. Some character is added after X_i
4. X_i is replaced with Y_i

These following statements can be made respectively from the above:

1. d(i,j) = d(i-1, j-1)
2. d(i,j) = d(i-1, j) +1
3. d(i,j) = d(i, j-1) +1
4. d(i,j) = d(i-1, j-1) +1

We compute d(i,j) by looking at the following piecewise function:

$$\begin{cases} min\{d(i-1,j-1)+w, d(i-1,j)+1, d(i,j-1)+1\} & ,i \geq 1, j \geq 1, \text{ [w=0 if } X_i=Y_j\text{, otherwise w=1]} \\ j & ,i = 0 \\ i & ,j = 0 \end{cases}$$

The pseudocode for this is:

```
FOR i=0 to n DO
        FOR j=0 to n DO
                Compute d(i,j)
```

The running time of this solution is $O(n^2)$. This algorithm leads to an exact solution.

3 Faster algorithm for Edit Distance

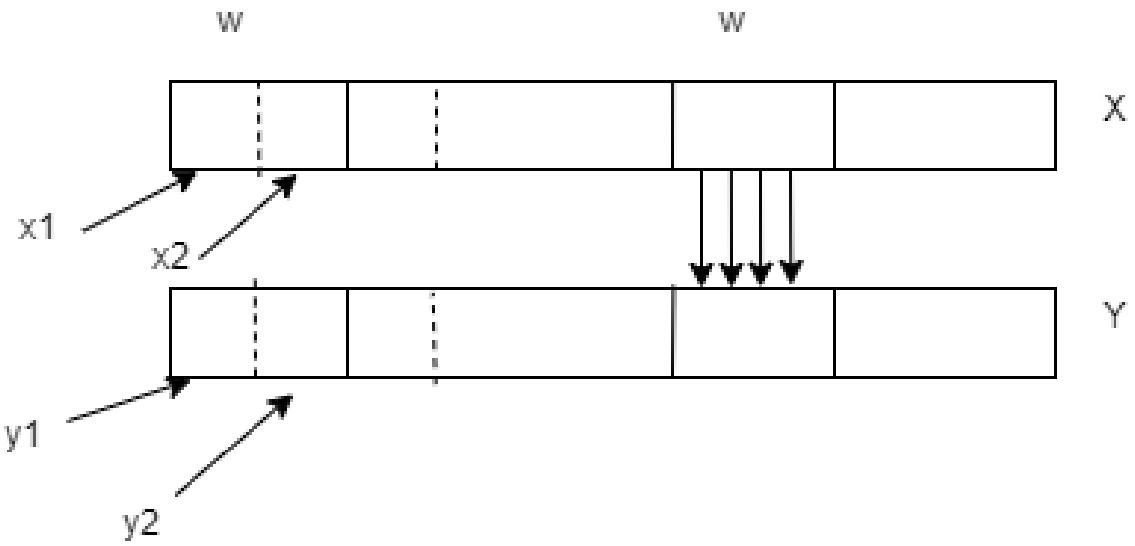
$[Masek - Peterson(1980)]$
This algorithm provides an improvement in running time over the previous algorithm. Its running time is $O(n^2/log^2 n)$ for $|\Sigma| = O(1)$ (the case where the alphabet set is constant).
If the Strong Exponential Time Hypothesis (SETH) holds, then no algorithm will solve the edit distance problem in $n^{2-O(1)}$. This means that the edit distance problem cannot be solved in time $n^{1.999\cdots}$. Further, this implies that the 3SAT (NP-Complete) problem cannot be solved in time $2^{O(n)}$.

4 Approximate algorithm to find Edit Distance

We define the Approximation Factor (AF) for the algorithm in question to be:

$$\frac{ED \text{ } Estimated \text{ } by \text{ } the \text{ } algorithm}{Optimal \text{ } ED} \geq 1$$

The goal of this algorithm is to minimize AF for the worst case.
The following are examples of approximate algorithms developed over time:



- AF: $\log n^{O(1/\epsilon)}$ approximate algorithm with running time $O(n^{1+\epsilon})$ for any $\epsilon > 0$ [$Andoni - Krauthgamer - Onak$ (2008)]
- AF: $O(1)$ approximate algorithm with running time time $O(n^{1.618})$ [$Chakraborty - Das - Goldenberg - Kouchy - Saks$ (2018)]
- AF: $2^{O(1+1/\epsilon)}$ approximate algorithm with running time $O(n^{1+\epsilon})$ [$Andoni - Nasatzki$]

We will look into the second algorithm above more closely. Here, We reduce the problem of the edit distance into the edit distance between two substrings. We refer back to optimal alignment. The replacement operation can be simulated by a character insertion and a character deletion. This increases the edit distance by a constant factor.
We can think of the alignment as a function π that matches each position of a character into another position of the character with $\perp$ denoting that the character doesnt match with any character in the other string. This can be represented as:
$\pi : [n] \longrightarrow [n] \cup \perp$

with the restrictions that:
$\pi(i), \pi(j) \neq \perp$
$\pi(i) < \pi(j)$

We break X and Y into substrings of fixed length w (as shown in the figure above), where w $\approx n^{0.9}$. We use x_i to denote the substring at the ith character of X and y_i to denote the substring at the ith character of Y. We want to know how a substring of length w in X gets aligned to some part of Y.
We can make an observation here. Let's assume that $X_i \sim Y_i$ (some match occurs). If w matches a substring of length T where $T > w$, then:

$EditDistance \geq T - w$, and T can be truncated to make t-w.

$EditDistance \approx \min \sum_i \frac{ED(X_i, Y_{\pi(i)})}{w}$

The main reasoning here is that computing the edit distance of strings can be reduced to computing the edit distance of substrings.