The 5th National Big Data Health Science Conference
University of South Carolina
Columbia, SC

Feb. 2 - 3, 2024

An Algorithm for the Constrained Longest Common Subsequence and Substring Problem

Rao Li
University of South Carolina Aiken

Joint work with Jyotishmoy Deka, Kaushik Deka, and Dorothy Li

Subsequences and Substrings

-Let ∑ be an alphabet and **S** a string over ∑. A subsequence of a string **S** is obtained by deleting zero or more letters from **S**.

If **S** = “ACGTU”, then “ATU” is a subsequence of **S**.

-A substring of a string **S** is a subsequence of S consists of consecutive letters in **S**.

If **S** = “ACGTU”, then “CGT” is a substring of **S,** “ATU” is not a substring of **S**,

-Every substring of **S** is also a subsequence of **S**.

-The empty string is a subsequence and a substring of any string.

The Longest Common Subsequence Problem for Two Strings

-The longest common subsequence problem for two strings X and Y is to find a longest string, denoted LCSSeq(X, Y), which is a subsequence of both X and Y.

-Obviously, the set of LCSSeq(X, Y) and the set LCSSeq(Y, X) are the same.

$|LCSSeq(X, Y)| = |LCSSeq(Y, X)|$.

The Longest Common Substring Problem for Two Strings

-The longest common substring problem for two strings X and Y is to find a longest string, denoted LCSStr(X, Y), which is a substring of both X and Y.

-Obviously, the set of LCSStr(X, Y) and the set LCSStr(Y, X) are the same.

|LCSStr(X, Y)| = |LCSStr(Y, X)|.

The Longest Common Subsequence and Substring Problem for Two Strings

-In [1], Li, Deka, and Deka introduced the longest common subsequence and substring problem for two strings X and Y which is to find a longest string, denoted LCSSeqStr(X, Y), that is a subsequence of X and a substring Y.

-[1] R. Li, J. Deka, and K. Deka, An algorithm for the longest common subsequence and substring problem, Journal of Math and Informatics 25 (2023) 77-81.

The Longest Common Subsequence and Substring Problem for Two Strings

-In general, the set of LCSSeqStr(X, Y) and the set LCSSeqStr(Y, X) are not the same. |LCSSeqStr(X, Y)| $\neq$ |LCSSeqStr(Y, X)|.

-In [1], Li, Deka, and Deka designed an algorithm for LCSSeqStr(X, Y). The time and space complexities of the algorithm are O(|X| |Y|), where |X| and |Y| are the lengths of strings X and Y, respectively.

Examples

- Suppose **X** = “GAAAACCCT” and **Y** = “GACACACT”.

-“AC” is a longest common substring for **X** (resp. **Y**) and **Y (**resp. **X)**. Thus |LCSStr(**X**, **Y**)| = |LCSStr(**Y**, **X**)| = 2.

-“ACT” is a longest common subsequence and substring for **X** and **Y**. Thus |LCSSeqStr(**X**, **Y**)| = 3.

-“ACCCT” is a longest common subsequence and substring for **Y** and **X**. Thus |LCSSeqStr(**Y**, **X**)| = 5.

-“GAAACT” is a longest common subsequence for **X** (resp. **Y**) and **Y** (resp. **X**). Thus LCSSeq(**X**, **Y**)| = |LCSSeq(**Y**, **X**)| = 6.

The Three Problems

-$|LCSStr(X, Y)| = |LCSStr(Y, X)| \leq |LCSSeqStr(X, Y)| \leq$

$|LCSSeq(X, Y)| = |LCSSeq(Y, X)|$.

-$|LCSStr(X, Y)| = |LCSStr(Y, X)| \leq |LCSSeqStr(Y, X)| \leq$

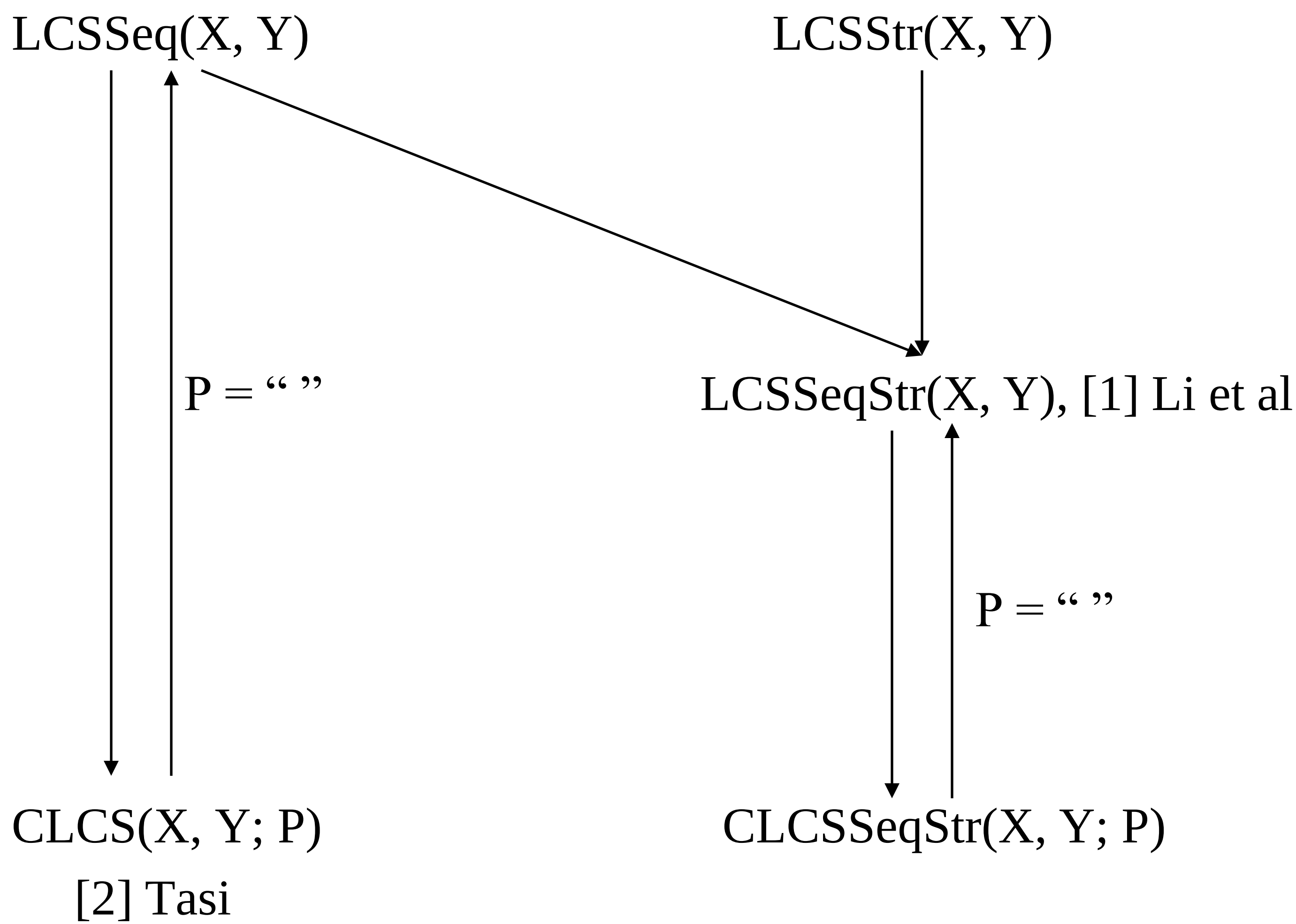
$|LCSSeq(X, Y)| = |LCSSeq(Y, X)|$.

-$|LCSStr(X, Y)| = |LCSStr(Y, X)| \leq \min\{|LCSSeqStr(X, Y)|, |LCSSeqStr(Y, X)|\}$

$\leq \max\{|LCSSeqStr(X, Y)|, |LCSSeqStr(Y, X)|\}$

$\leq |LCSSeq(X, Y)| = |LCSSeq(Y, X)|$.

The Problems



The Constrained Longest Common Subsequence Problem for Two Strings

-Tsai [2] extended the longest common subsequence problem for two strings to the constrained longest common subsequence problem for two strings and a constrained string P.

-[2] Y. T. Tsai, The constrained longest common subsequence problem,

Information Processing Letters 88 (2003) 173-176.

The Constrained Longest Common Subsequence Problem for Two Strings

-For two strings X, Y, and a constrained string P, the constrained longest common subsequence problem for two strings X and Y with respect to P is to find a longest string Z: = CLCSSeq(X, Y; P) such that Z is a subsequence of both X and Y and P is a subsequence of Z.

-Clearly, if P is an empty string, then CLCSSeq(X, Y; P) = LCSSeq(X, Y).

The Constrained Longest Common Subsequence Problem for Two Strings

-“Such a problem could arise in computing the homology of two biological sequences which have a specific or putative structure in common” quoted from [2].

-Tsai [2] designed an $O(|X^2 |Y|^2 |P|)$ time algorithm for CLCSSeq(X, Y; P).

The Constrained Longest Common Subsequence and Substring Problem for Two Strings

-Motivated by Tasi’s work, we introduced the constrained longest common subsequence and substring problem for two strings and a constrained string.

The Constrained Longest Common Subsequence and Substring Problem for Two Strings

-For two strings X, Y, and a constrained string P, the constrained longest common subsequence and substring problem for two strings X and Y with respect to P, is to find a longest string Z: = CLCSSeqStr(X, Y; P) such that Z is a subsequence of X, a substring of Y, and P is a subsequence of Z.

-Clearly, if P is an empty string, then CLCSSeqStr(X, Y; P) = LCSSeqStr(X, Y) in [1].

The Constrained Longest Common Subsequence and Substring Problem for Two Strings

-Suppose **X** = “GAAAACCCT”, **Y** = “GACACACT”, **P** = “AC”.

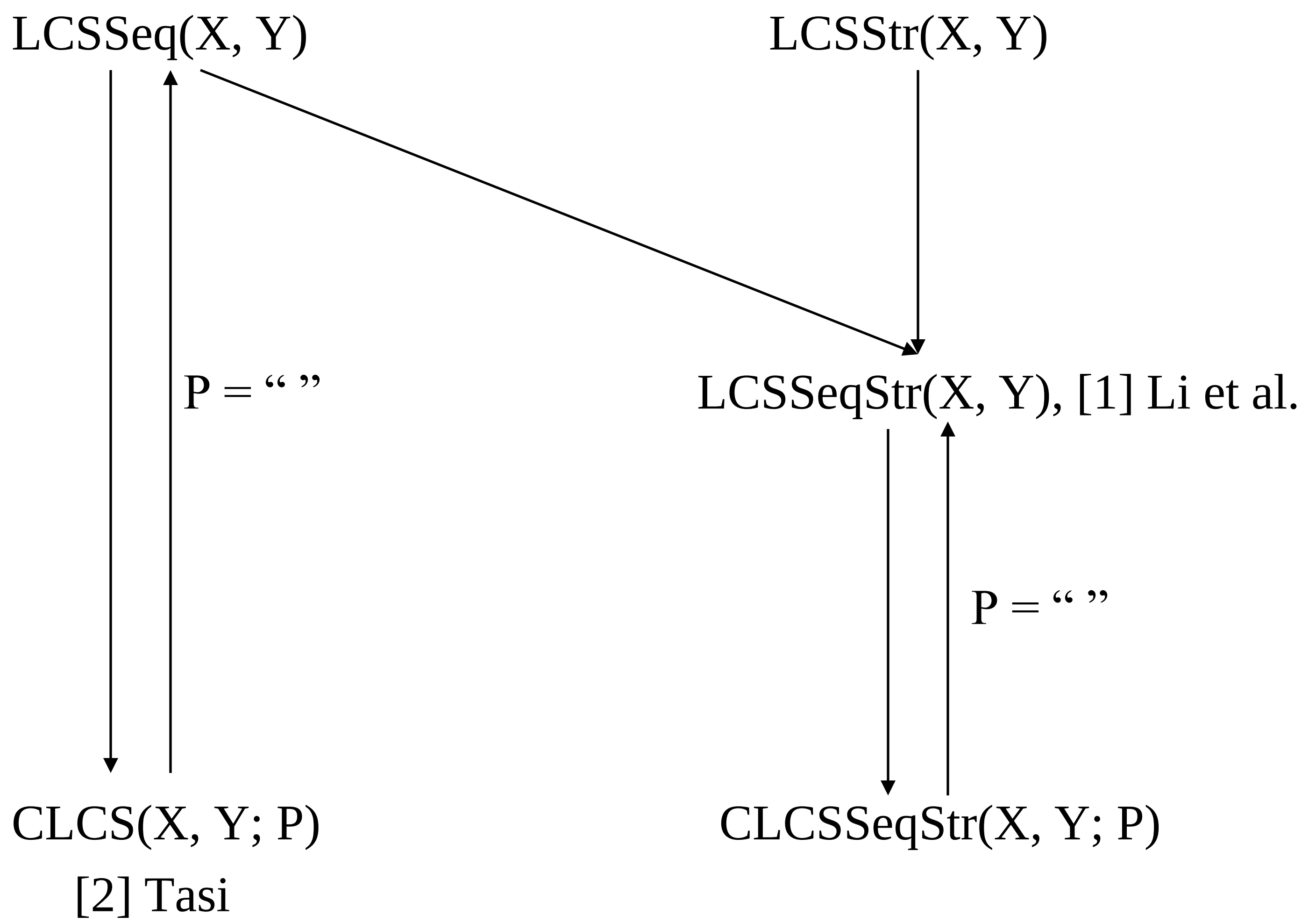
-“ACT” is a constrained longest common subsequence and substring for **X** and **Y**.

Thus |CLCSSeqStr(**X**, **Y; P**)| = 3.

-“GAAACT” is a constrained longest common subsequence for **Y** (resp. **X**) and **X** (resp. **Y**). Thus |CLCSSeq(**X**, **Y; P**)| = |LCSSeq(**Y**, **X; P**)| = 6.

-In general, |CLCSSeqStr(**X**, **Y; P**)| ≤ |CLCSSeq(**X**, **Y; P**)|.

The Problems



The Constrained Longest Common Subsequence and Substring Problem for Two Strings

-Using dynamic programing (DP), we designed an algorithm for finding CLCSSeqStr(X, Y; P). Both time complexity and space complexity of our algorithm are O(|X| |Y| |P|).

The Algorithm

-Let $S = s_1\ s_2\ ...\ s_r$ be a string over an alphabet $\sum$, The r prefixes of

S are defined as $S_1 = s_{1,}\ S_2 = s_1 s_2,\ S_3 = s_1\ s_2\ s_3, \ldots,$ and $S_r = s_1\ s_2\ ...\ s_r$.

S_0 is defined as an empty string.

-The r suffixes of S are defined as $T_1 = s_1\ s_2\ ...\ s_r$, $T_2 = s_2\ s_3\ ...\ s_r$, ..., $T_{r-1} = s_{r-1}s_r$,

and $T_r = s_r$,

The Algorithm

-Let $X = x_1\ x_2\ ...\ x_m$, $Y = y_1\ y_2\ ...\ y_n$, and $P = p_1\ p_2\ ...\ p_r$. Define Z[i, j, k] as a string

satisfying the following conditions.

(1) it is a subsequence of $X_i = x_1\ x_2\ ...\ x_{i,}$

(2) it is a suffix of $Y_j = y_1\ y_2\ ...\ y_{j,}$

(3) it has $P_k = p_1\ p_2\ ...\ p_k$ as a subsequence,

(4) under (1), (2) and (3), its length is as large as possible,

where $1 \le i \le m$, $1 \le j \le n$, and $1 \le k \le r$.

The Algorithm

-We will use a 3-dimensional array $M[m + 1][n + 1][r + 1]$ to store $|Z[i, j, k]|$.

Namely, $M[i][j][k] = |Z[i, j, k]|$, where $0 \leq i \leq m$, $0 \leq j \leq n$, $0 \leq k \leq r$.

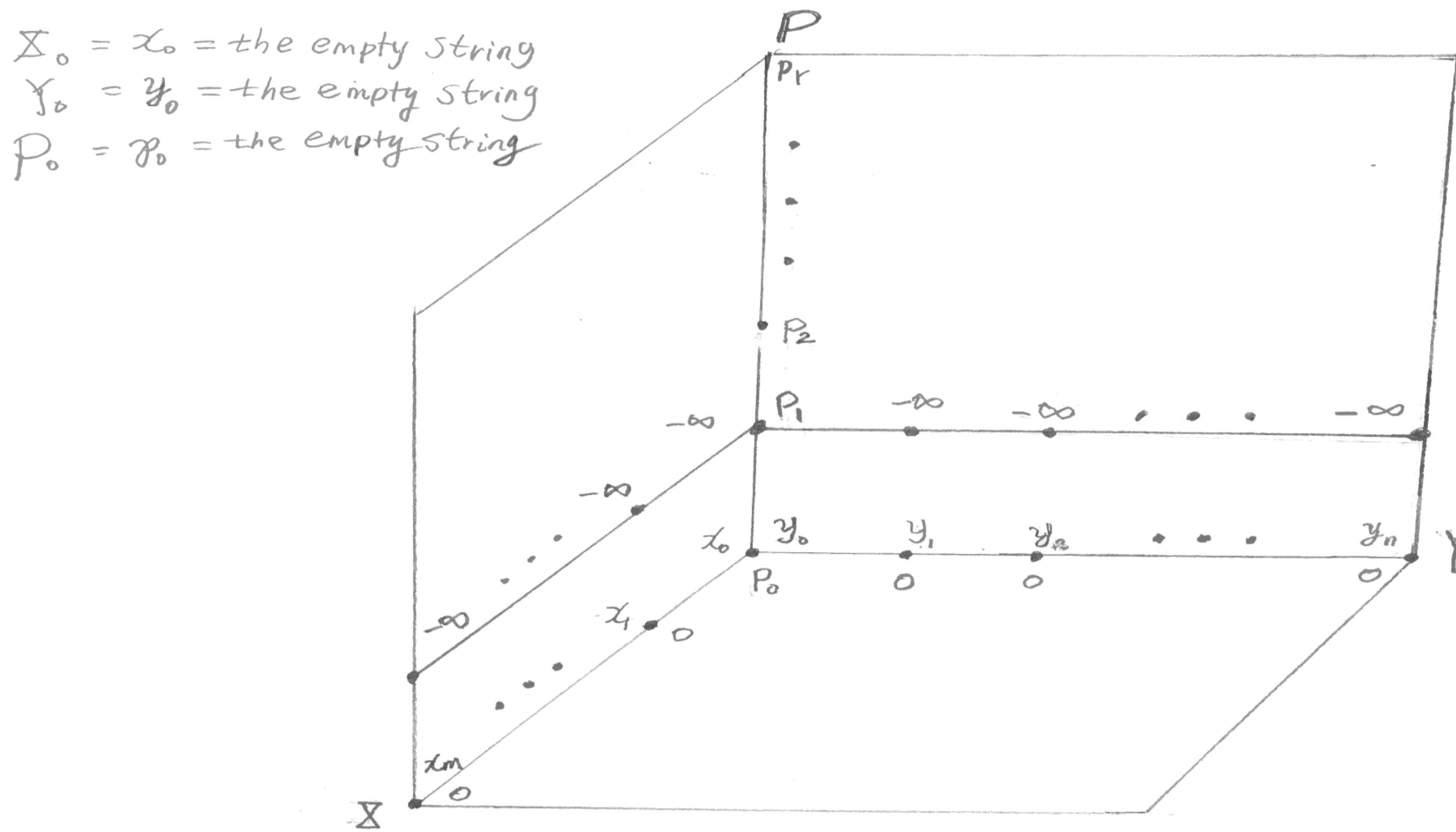
-We will recursively fill in the cells in M.

-Firstly, we fill in the boundary cells in array M.

The Algorithm

-Filling in the boundary cells.

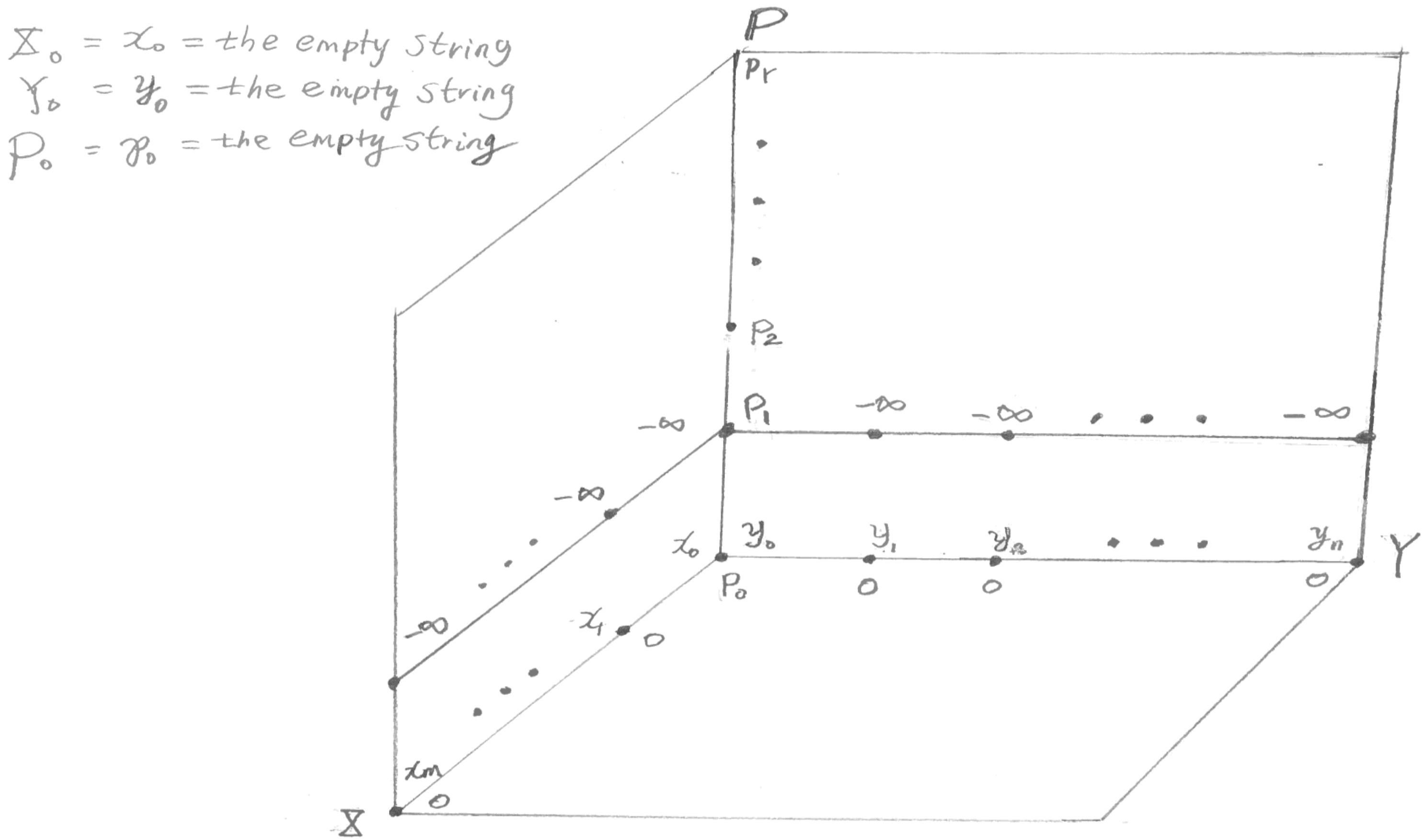
[1] If $i = 0$ and $k = 0$ the length of a string which is a subsequence of X_i, a suffix of Y_j, and has P_k as a subsequence is zero. Thus $M[0][j][0] = 0$, where $0 \le j \le n$.



The Algorithm

-Filling in the boundary cells.

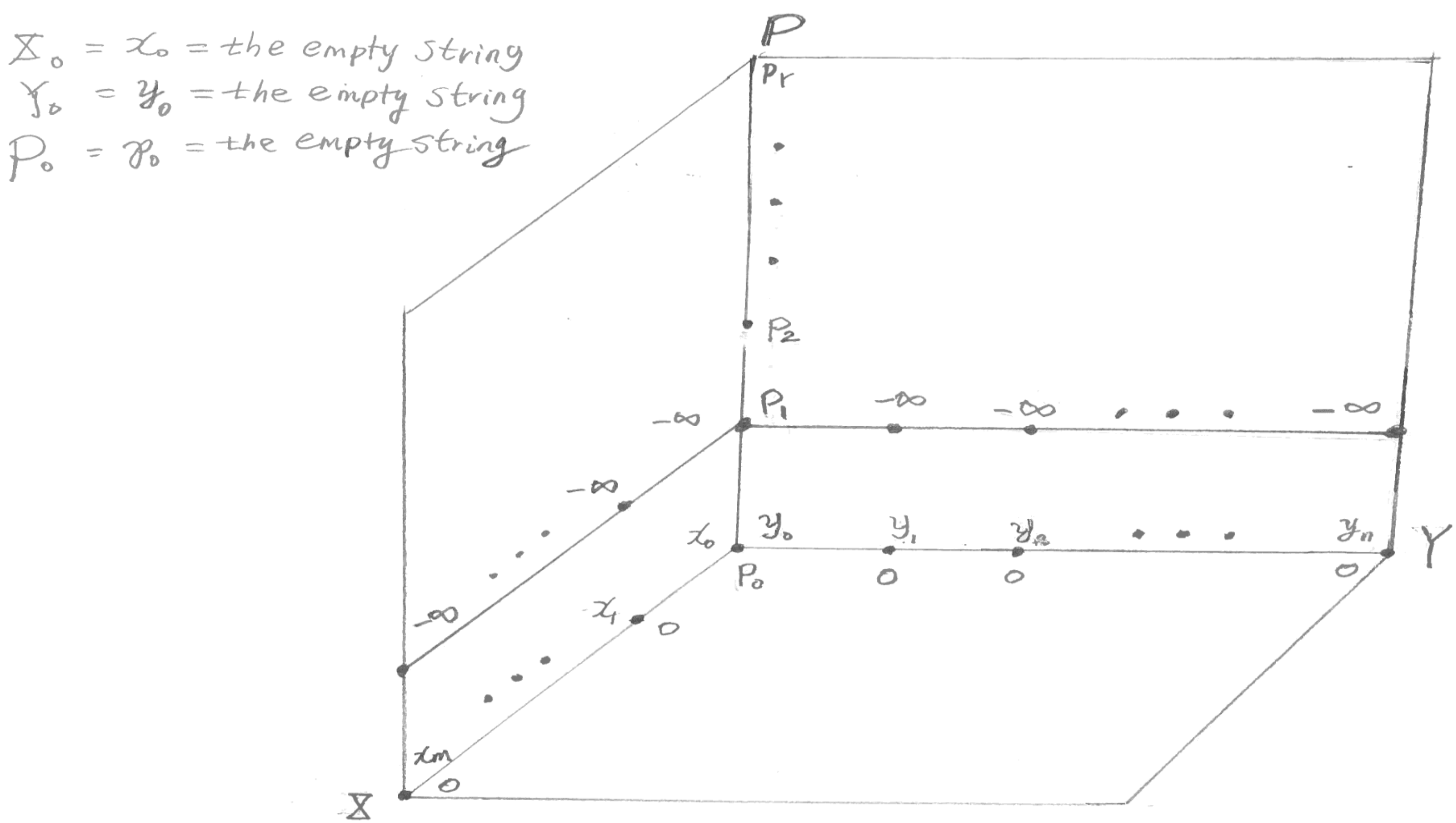
[2] If j = 0 and k = 0, the length of a string which is a subsequence of X_i, a suffix of Y_j, and has P_k as a subsequence is zero. Thus M[i][0][0] = 0, where $0 \leq i \leq m$.



The Algorithm

-Filling in the boundary cells.

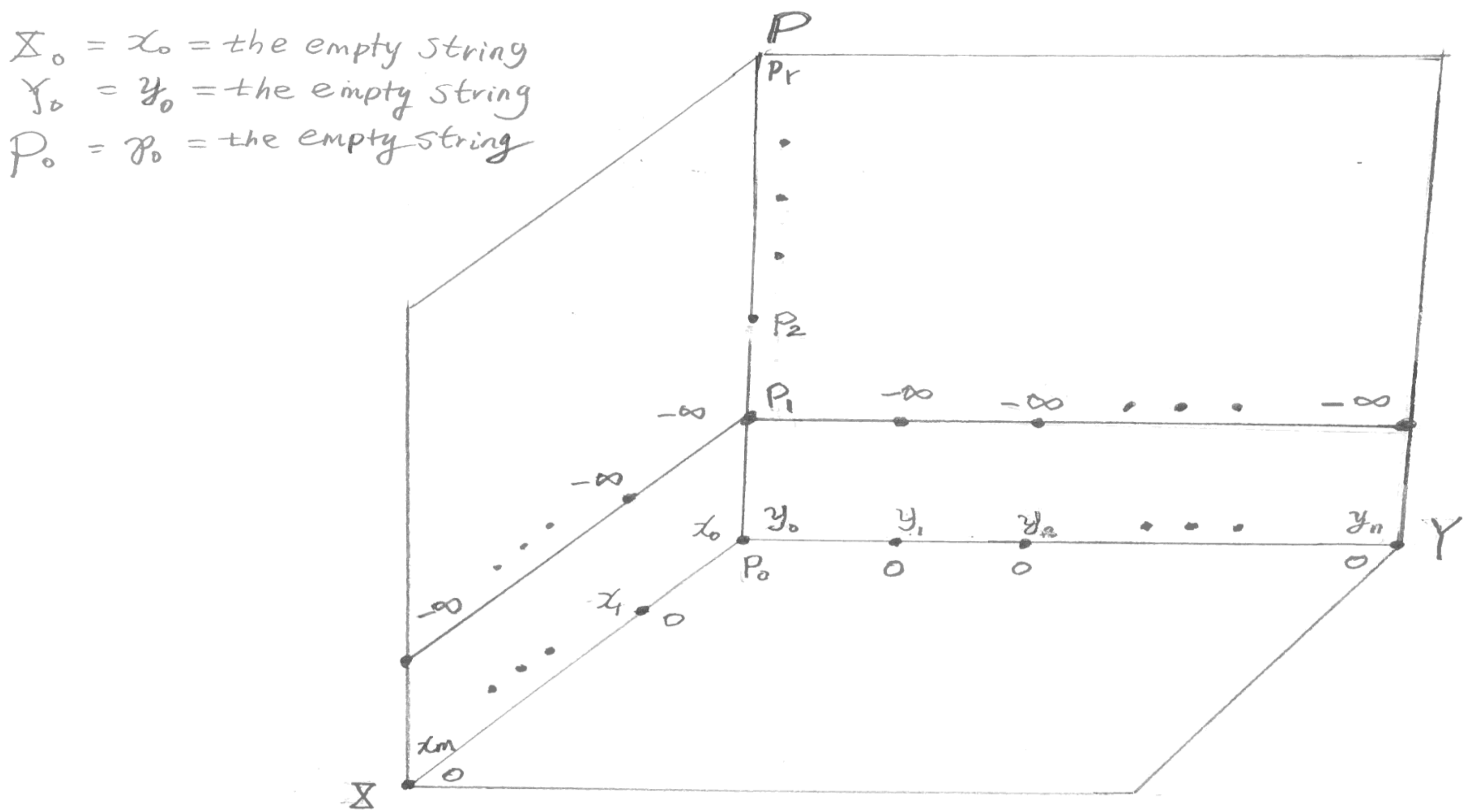
[3] If $i = 0$ and $k \geq 1$, there is not a string which is a subsequence of X_i, a suffix of Y_j, and has P_k as a subsequence. Thus $M[0][j][k] = -\infty$, where $0 \leq j \leq n$ and $1 \leq k \leq r$.



The Algorithm

-Filling in the boundary cells.

[4] If $j = 0$ and $k \geq 1$, there is not a string which is a subsequence of X_i, a suffix of Y_j, and has P_k as a subsequence. Thus $M[i][0][k] = -\infty$, where $0 \leq i \leq m$ and $1 \leq k \leq r$.

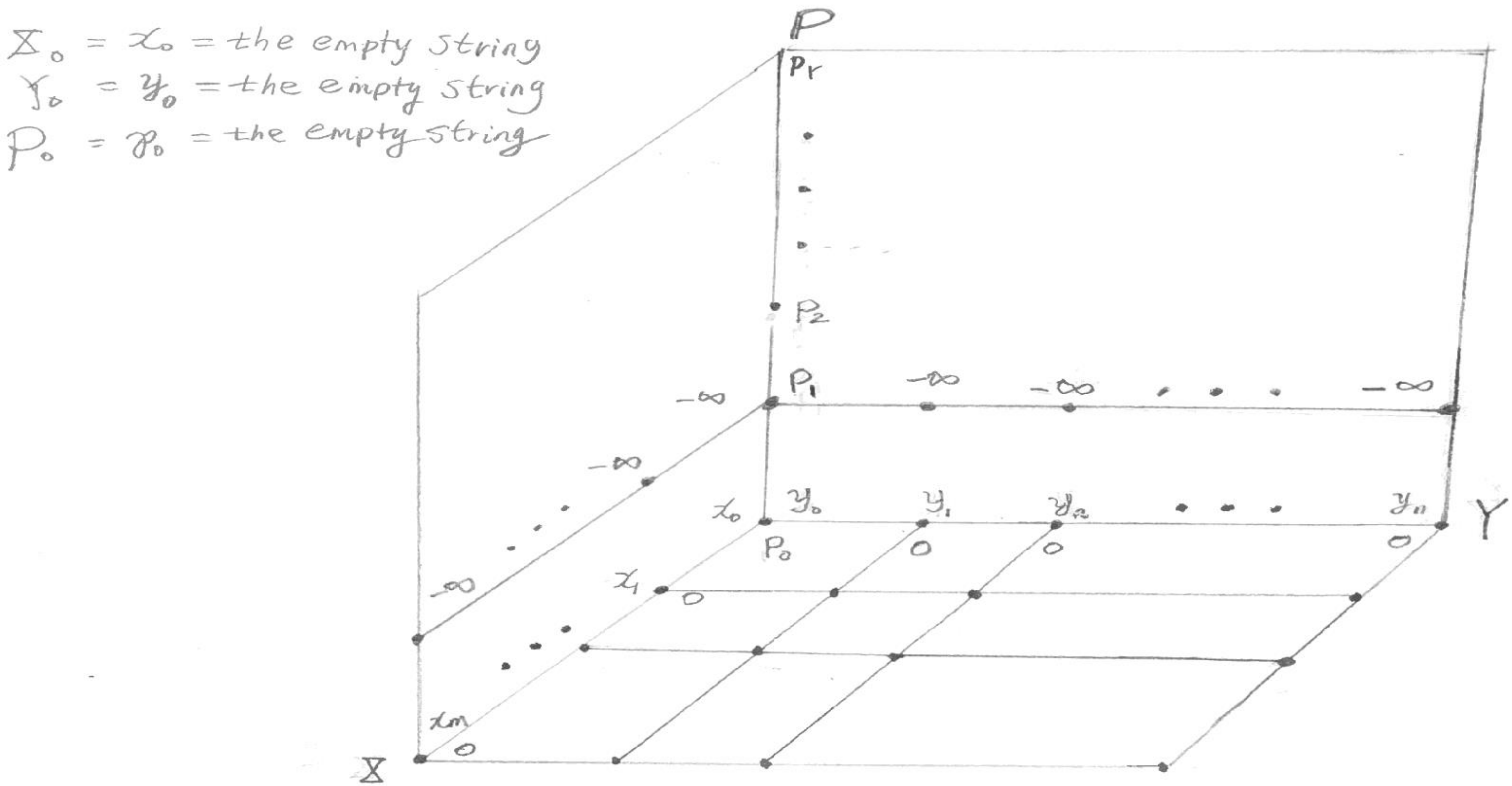


The Algorithm

-Filling in the boundary cells.

[5] If k = 0 or P_k is an empty string. Then CLCSSeqStr(X, Y; P_k) = LCSSeqStr(X, Y) in [1]. The cells of M[i][j][0], where $1 \leq i \leq m$ and $1 \leq j \leq n$, can be filled in by the following rules.

If $x_i = y_j$, then M[i][j][0] = M[i - 1][j - 1][0] + 1. If $x_i \neq y_j$, then M[i][j][0] = M[i - 1][j][0].



The Algorithm

-Filling in other cells.

-**Claim 1.** Suppose that $X_i = x_1\ x_2\ ...\ x_i$, $Y_j = y_1\ y_2\ ...\ y_j$, and $P_k = p_1\ p_2\ ...\ p_k$, where $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq k \leq r$. If $Z[i, j, k] = z_1\ z_2\ ...\ z_a$ is a string satisfying conditions

(1) it is a subsequence of $X_i = x_1\ x_2\ ...\ x_i$,

(2) it is a suffix of $Y_j = y_1\ y_2\ ...\ y_j$,

(3) it has $P_k = p_1\ p_2\ ...\ p_k$ as a subsequence,

(4) under (1), (2) and (3), its length is as large as possible,

where $1 \leq i \leq m$, $1 \leq j \leq n$, and $1 \leq k \leq r$.

Then we have only the following possible cases and the statement in each case is true.

The Algorithm

-Filling in other cells.

-<u>Claim 1.</u>

Case 1. $x_i = y_j = p_k$. We have $|Z[i, j, k]| = |Z[i - 1, j - 1, k - 1]| + 1$ in this case.

Case 2. $x_i = y_j \neq p_k$. We have $|Z[i, j, k]| = |Z[i - 1, j - 1, k]| + 1$ in this case.

Case 3. $x_i \neq y_j$, $x_i \neq p_k$, and $y_j = p_k$. We have $|Z[i, j, k]| = |Z[i - 1, j, k]|$ in this case.

Case 4. $x_i \neq y_j$, $x_i \neq p_k$, and $y_j \neq p_k$. We have $|Z[i, j, k]| = |Z[i - 1, j, k]|$ in this case.

Case 5. $x_i \neq y_j$, $x_i = p_k$, and $y_j \neq p_k$. This case cannot happen.

The Algorithm

-Filling in other cells.

-**<u>Claim 2.</u>** Suppose there is not a string which is a subsequence of $X_i = x_1\ x_2\ ...\ x_i$,

a suffice of $Y_j = y_1\ y_2\ ...\ y_j$, and has $P_k = p_1\ p_2\ ...\ p_k$, as a subsequence, where

$1 \leq\ i \leq\ m$, $1 \leq\ j \leq\ n$, $1 \leq k \leq r$. Namely, Z[i, j, k] doesn't exist. Then

[1]. If $x_i = y_j = p_k$, then there is not a string which is a subsequence of X_{i-1}

$= x_1\ x_2\ ...\ x_{i-1}$, a suffix of $Y_{j-1} = y_1\ y_2\ ...\ y_{j-1}$, and has $P_{k-1} = p_1\ p_2\ ...\ p_{k-1}$

as a subsequence. **Namely, if Z[i, j, k] does not exist, then Z[i - 1, j - 1, k - 1] does not exist either.**

The Algorithm

-Filling in other cells.

-<u>Claim 2.</u>

[2]. If $x_i = y_j \neq p_k$, then there is not a string which is a subsequence of

$X_{i-1} = x_1\ x_2\ ...\ x_{i-1}$, a suffix of $Y_{j-1} = y_1\ y_2\ ...\ y_{j-1}$, and has $P_k = p_1\ p_2\ ...\ p_k$

as a subsequence. **Namely, if Z[i, j, k] does not exist, then Z[i - 1, j - 1, k]**

does not exist either.

The Algorithm

-Filling in other cells.

-<u>Claim 2.</u>

[3]. If $x_i \neq y_j$, $x_i \neq p_k$, and $y_j = p_k$, then there is not a string which is a subsequence of $X_{i-1} = x_1\ x_2 \ldots x_{i-1}$, a suffix of $Y_j = y_1\ y_2 \ldots y_j$, and has $P_k = p_1\ p_2 \ldots p_k$ as a subsequence. **Namely, if Z[i, j, k] does not exist, then Z[i - 1, j, k] does not exist either.**

The Algorithm

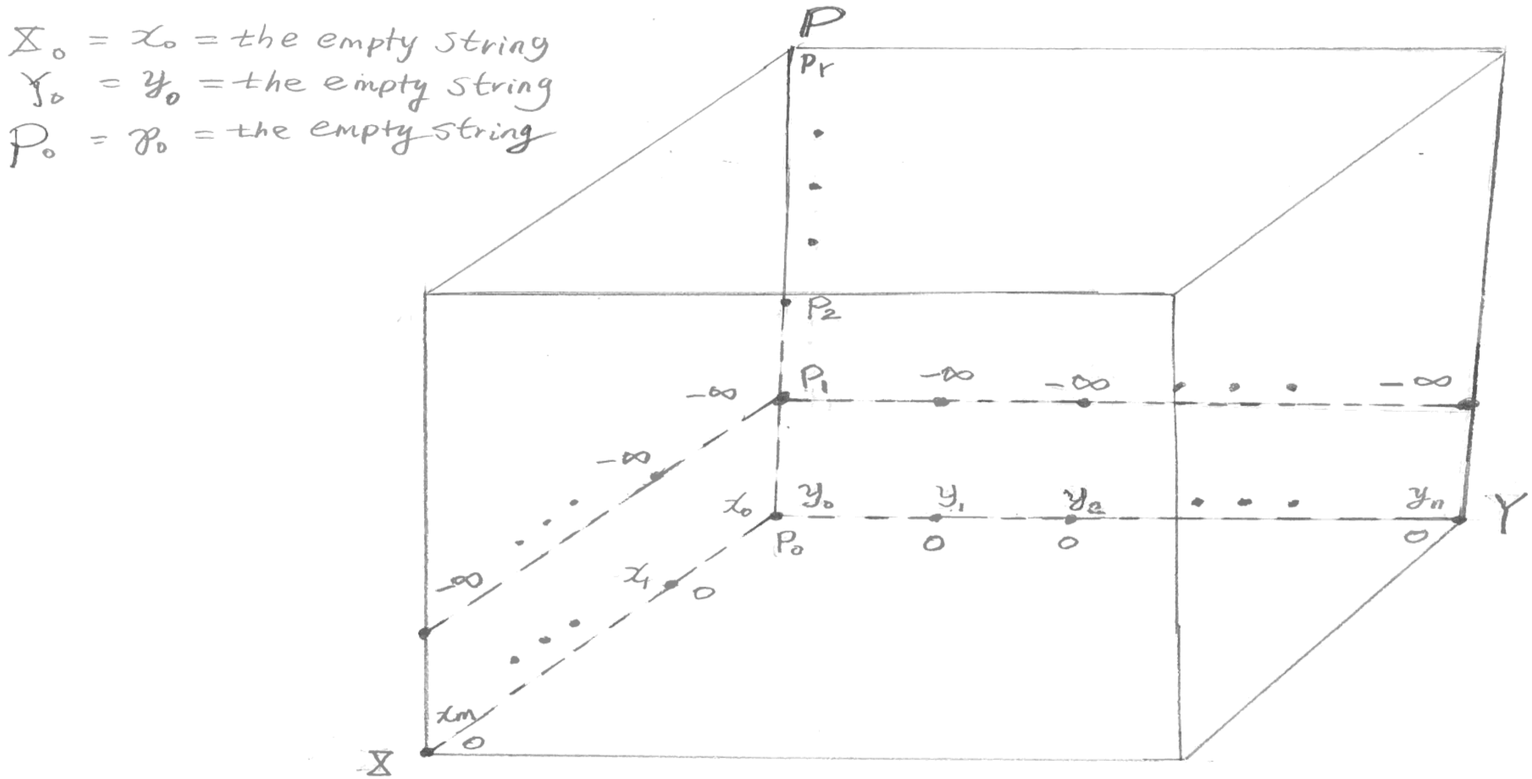
-Filling in other cells.

-<u>Claim 2.</u>

[4]. If $x_i \neq y_j$, $x_i \neq p_k$, and $y_j \neq p_k$, then there is not a string which is a subsequence for $X_{i-1} = x_1\ x_2\ ...\ x_{i-1}$, a suffix of $Y_j = y_1\ y_2\ ...\ y_j$, and has $P_k = p_1\ p_2\ ...\ p_k$ as a subsequence. **Namely, if Z[i, j, k] does not exist, then Z[i - 1, j, k] does not exist either.**

The Algorithm

-**Claim 3.** Let U_k be a longest string which is a subsequence of X, a substring of Y, and has P_k as a subsequence. Then $|U_k| = \max\{|Z[i, j, k]| : 1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq r\}$.
Thus $|U_r| = \max\{|Z[i, j, r]| : 1 \leq i \leq m, 1 \leq j \leq n\} = |CLCSSubStr(X, Y; P)|$.



The Algorithm

-$|CLCSSeqStr(X, Y; P)| = \max\{M[i][j][r]: 0 \le i \le m, 0 \le j \le n\}$.

-We can also find the CLCSSeqStr(X, Y; P) when we write a program.

-The time complexity of our algorithm is

$$O((|X| + 1)(|Y| + 1)(|P| + 1)) \sim O(|X|\ |Y|\ |P|).$$

-The space complexity of our algorithm also is

$$O((|X| + 1)(|Y| + 1)(|P| + 1)) \sim O(|X|\ |Y|\ |P|).$$

Thanks