Variants of Longest Common Subsequence Problem

by

Kranthi Chandra

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by
Professor Stanisław P. Radziszowski

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 2016

The project "Variants of Longest Common Subsequence Problem" by Kranthi Chandra has been examined and approved by the following Examination Committee:

Project Committee Chair

Project Committee Reader

Project Committee Observer

Dedication

To all my teachers and professors

Acknowledgements

I am grateful to the committee chair, Professor Stanisław P. Radziszowski whose constant guidance helped me deliver my best work. My sincere gratitude to Professor Hans-Peter Bischof for his valuable inputs.

Contents

1 Introduction **7**
- 1.1 Constrained LCS: Multiple substring exclusive constraints 7
- 1.2 K-substring LCS 7
- 1.3 LCS with variable gapped constraints 7
- 1.4 Our Work 7

2 Background **9**
- 2.1 Longest Common Subsequence 9
- 2.2 Properties of the LCS problem 9
- 2.3 Applications 10

3 LCS Problem with Multiple Substring Exclusive Constraints **12**
- 3.1 Problem Definition 12
- 3.2 Preliminaries 13
- 3.3 Dynamic Programming Algorithm 13

4 K-substring LCS problem **17**
- 4.1 Problem Definition 17
- 4.2 Preliminaries 18
- 4.3 Dynamic Programming solution 18

5 Variable gapped LCS **21**
- 5.1 Preliminaries 21
- 5.2 Problem Definition 22
- 5.3 Naive Algorithm 22
- 5.4 Improved Algorithm 23
- 5.5 Incremental sequence maximum queries problem 24
- 5.6 Use of ISMQ in the improved algorithm 25
- 5.7 Union-find problem 25

6 Experiments **28**
- 6.1 Multiple exclusive constrained LCS 28
- 6.2 LCS-k 30
- 6.3 Variable gapped LCS 30

7 Conclusion **34**
- 7.1 Current Status 34
- 7.2 Lessons Learned 34
- 7.3 Future Work 35

Abstract

Longest common subsequence is a classical problem in computer science. It is a problem of finding longest subsequence common to the given input sequences [2]. Information in various applications such as bioinformatics is expressed as sequence of characters (e.g. DNA or gene sequence). LCS is a similarity measure of these character sequences and has practical applications in genomics and molecular biology. For example, LCS is used in finding evidence for species common origin and also used in gene discovery and evolutionary tree construction. Motivated by practical scenarios in applications of genomics and computational biology, different variants of LCS problem were introduced for more accurate measure of similarity. This project is to study the following variants of longest common subsequence. 1. Constrained LCS: Multiple substring exclusive constraints [3]. 2. LCS in k-length substrings [2]. 3. Variable-gapped LCS [1]. In this project, these variants are implemented and compared with the dynamic programming approach of standard LCS problem.

1 Introduction

Longest common subsequence (LCS) is a problem of computing longest subsequence common to the given input sequences. Different variants of LCS were introduced due to the various practical scenarios in the applications of genomics and computational molecular biology. This project studies the algorithms to solve the following variants of LCS problem.

1.1 Constrained LCS: Multiple substring exclusive constraints

In this problem, the computed LCS must exclude the characters present in the set of constraint strings. This problem was initially considered as NP-Hard problem but later it was found that it is is a non NP hard problem [3]. A new polynomial time algorithm for the Multiple substring exclusive constraints problem is given in the paper Zhu et al in [3].

1.2 K-substring LCS

In this problem, maximal number of k substrings matching in the given two sequences are determined by preserving order of appearance [2]. In order to improve the accuracy of similarity measure, LCS-k was introduced by ensuring that the adjacent characters in the LCS are also adjacent in both the given sequences [2]. LCS problem is a special case of LCS-k with $k = 1$.

1.3 LCS with variable gapped constraints

This problem proposed in paper Yung-Hsing et al in [1] has applications in molecular biology and genetics. It can be used in motif pattern discovery problems especially in scenarios when all the positions of desired pattern are not important and also in the process of long multiple repeats extraction in DNA sequences [4]. For instance, consider sequences, $A = RCLPCRR$ and $B = RPPLCPLRC$, the detected LCS with gap constraint k=2 is $R..L..C..R$. Here, in the study of protein sequences, the gaps(dots) '.' in computed LCS are filled with any type of amino acids. Previous works were focused on fixed gaps. But in real time applications, the gaps are not fixed. Hence, this project studies the LCS problem with variable gaps.

1.4 Our Work

In this project, different algorithms are studied to solve the variants of LCS. For the experiment, Constrained LCS, K-substring LCS and Variable-gapped LCS are

implemented and compared with the dynamic programming method of LCS problem.

2 Background

2.1 Longest Common Subsequence

Longest Common Subsequence (LCS) is a problem of finding longest subsequence common to the given input sequences. This problem is not equivalent to longest common substring problem. This is because in LCS problem, there is no restriction for characters picked for subsequence to be in consecutive positions in the given sequences.

LCS problem contains optimal substructure i.e problem can be broken into subproblems and these subproblems can be overlapping. That is, the higher level subproblems can reuse the lower level subproblem in order to avoid recomputation. Due to these properties, dynamic programming technique can be used to solve LCS problem.

The recurrence relation for solving LCS Problem is as follows:

$$LCS[i,j] = \begin{cases} 0 & \textit{if } i = 0 \textit{ or } j = 0; \\ LCS[i-1, j-1] + 1 & \textit{if } X[i] = Y[j]; \\ max(LCS[i-1, j], LCS[i, j-1]) & \textit{if } X[i] \neq Y[j]; \end{cases}$$

The above relation is used to extend the LCS length for every prefix pair $(X[1...i], Y[1...j])$. Here, $LCS[i,j]$ indicates the length of $LCS(X[1...i], Y[1...j])$. The length of $LCS(X,Y)$ is given by $LCS(n,n)$. $LCS(X,Y)$ can be obtained by backtracking from $LCS(n,n)$.

Dynamic programming is well known technique to solve the problems. In this technique, the complex problem is broken into simpler subproblems. These subproblems are solved individually and their solutions are stored. Hence, when similar sub-problem occurs, we can use the previously stored solution instead of recomputing. In this way, we can reduce the time complexity.

2.2 Properties of the LCS problem

LCS dynamic approach is based on the following two properties.

First Property:

Consider two sequences A and B of lengths n and m respectively. Let the elements of these sequences are denoted as $a_1, a_2...a_n$ and $b_1, b_2...b_m$ respectively.

Then, $LCS(A_n, B_m) = LCS(A_{n-1}, B_{m-1}) + a_n$ if $a_n = b_n$

Here, + indicates that last element of sequence A is appended to the prefix of sequence.

This property states that when last element of sequence A and sequence B are equal, then this last element can be removed from the sequences to make them shorter. Then LCS is computed on this shorter sequences. In this way, elements are removed until the last element of sequences A and B are equal. LCS is computed on this shorter sequence and last element is appended to this LCS to obtain the final LCS output.

Second Property:

Second property states that if the last elements of sequences A and B are not equal then LCS is the maximum of the sequences $LCS(A_{n-1}, B_m)$ and $LCS(A_n, B_{m-1})$.

The first famous dynamic programming solution of LCS , was appeared in 1974 [6]. The dynamic programming solution to LCS problem was invented by Wagner and Fischer. This approach runs in $O(n^2)$ time. This time complexity was improved to $O(n^2/logn)$ using Four Russian technique by Masek and Paterson[7]. Then they were other algorithms for LCS problem using different parameters. Like, Myers[8] and Nakatsu[9] proposed an algorithm which runs in $O(nD)$ time. Here, D is the Levenshtein distance between the given two input strings. Also, another approach was proposed by Hunt and Szymanski [5] has time complexity of $O((R + n)logn)$. Here R is the number of all possible matches.

2.3 Applications

1. Molecular Biology: LCS can be used as similarity measure in the study of DNA sequences. When a new DNA sequence is found, researchers tend to identify other similar sequences. The four submolecules of DNA can be represented as 4 characters (for. e.g. $ATGC$) and the length of LCS can be used as similar measure between these sequences.

2. File Comparison: In the comparison of two versions of same file, LCS can be used to find the changes being made in the modified version. In this scenario, each line in the files must be considered as a single character of the sequences.

3. Screen Redisplay: The text editors require to update the screen display when changes are made to the file. LCS helps to identify the parts of the display that

are correct and does not require any changes.

3 LCS Problem with Multiple Substring Exclusive Constraints

This problem introduces notion of constraints on the characters picked for LCS. In this problem, the computed LCS must exclude the characters present in the set of constraint strings. This problem was initially considered as NP-Hard problem in [10] but later it was found that it is is not NP hard problem [8]. A new polynomial time algorithm for the Constrained LCS problem is given in the paper Zhu et al in [8].

3.1 Problem Definition

Let $X = x_1x_2x_n$ and $Y = y_1y_2y_m$ of lengths n and m be two given input sequences and let $P = P_1, , P_d$ be set of d constraints of total length r, then multiple exclusion constraints LCS(M-STR-EC-LCS) problem is to compute LCS of X and Y that excludes each of the constraints $P_i \epsilon P$ as a substring.

Constrained LCS is broadly generalized into four types i.e finding Longest Common Subsequence (or substring) with inclusion (or exclusion) of characters from constraint string P [3]. Then it further generalized to use set of constraint strings instead of one single constraint string. Hence, constrained LCS is of following types,

Problem	Output
M-SEQ-IC-LCS	The longest common subsequence of X and Y including $P_i \in P$ as a subsequence
M-STR-IC-LCS	The longest common subsequence of X and Y including $P_i \in P$ as a substring
M-SEQ-EC-LCS	The longest common subsequence of X and Y excluding $P_i \in P$ as a subsequence
M-STR-EC-LCS	The longest common subsequence of X and Y excluding $P_i \in P$ as a substring

Figure 1: Constrained LCS[3]

Paper in [3] proposes algorithm for M-SEQ-EC-LCS. The failure functions of Knuth Morris Pratt is used for the string matching task. The dynamic programming approach used in paper[3] is based on this idea. This approach runs in $O(nmr)$. Here, n is length of sequence A, m is length of sequence B and r is total length of constraint strings.

3.2 Preliminaries

Let $X[i:j] = x_i x_{i+1} x_j$ denotes a substring of sequence X that starts from position i and ends in position j. When $i \neq 1$ or $j \neq n$, then $X[i:j] = x_i x_{i+1} x_j$ is proper substring of sequence X. When $i = 1$, then $X[i:j] = x_i x_{i+1} x_j$ is called as prefix of X and when $j = n$, $X[i:j] = x_i x_{i+1} x_j$ is called as suffix of X.

3.3 Dynamic Programming Algorithm

Keyword tree is used as data structure in this approach. Keyword tree is used to process the characters in constraint string P. This tree is rooted directed tree for constraint string set P which satisfies following conditions. Every edge of the keyword tree is labeled with exactly one character. Two edges which come out of same node will have different labels. Every string P_i in constraint string set P is mapped to some node v in tree T and the characters from the root to node v gives out the string P_i. Every leaf node of keyword tree T is mapped to constraint string P_i in set P. For example, consider constraint string set P = (aab, aba, ba) with $d = 3$ and $r = 8$. Here, d is the number of constraints and r is total length of these strings,

Let $L(i)$ represents the string which is formed by concatenating characters from root to node i. Let $lp(i)$ be the length of longest proper suffix of string $L(i)$ (which is a prefix of some string in tree T).

The arrows represented in figure 2 represent failure links which is direct generalization of failure functions in Knuth Morris Pratt algorithm. $pre(i)$ is used to indicate this failure link. This $pre(i)$ is used to search quickly all occurrences of constraint stings in sequence X.

Let S be any string, then $\sigma(S)$ denotes the unique suffix node with maximum depth.

For example, consider string $S = aabaaabb$, then $\sigma(S)$ is 6. This is because at node 6 in tree T, we can get maximum depth suffix of string S.

Let $Z(i, j, k)$ represents set of all LCSs of $X[i : n]$ and $Y[j : m]$ such that for every $zZ(i, j, k)$, $L(k) \oplus z$ excludes P, where $1 \leq i \leq n, 1 \leq j \leq m, 0 \leq k \leq s$. Here, $\oplus$ represents concatenation symbol. Let f(i, j, k) represent the length of an LCS in $Z(i, j, k)$. $f(i, j, k)$ can be computed by the following recursive formula,

$$f(i,j,k) = \begin{cases} max\{f(i+1,j+1,k), 1+f(i+1,j+1,q)\}, & if\ x_i = y_j\ and\ q < s; \\ max\{f(i+1,j,k), f(i,j+1,k)\}, & otherwise; \end{cases}$$

Here, q represents the $\sigma(L(k) \oplus x_i)$, and the base conditions are $f(i, m+1, k) = f(n+1, j, k) = 0$ for any $1 \leq i \leq n$, $1 \leq j \leq m$, and $0 \leq k \leq s$. Using this recursive formula, paper in [3] proposes dynamic programming solution given in Algorithm 1.

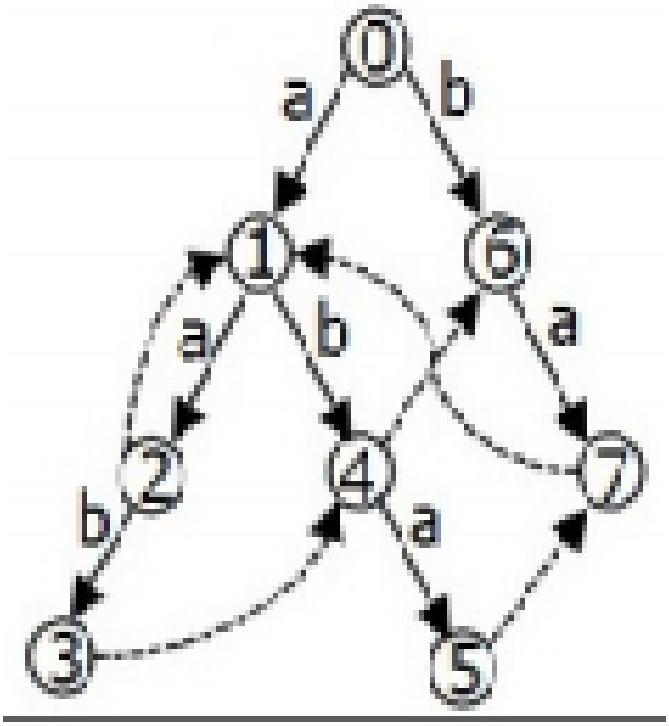

Figure 2: Keyword tree[3]

$\sigma(L(k)\oplus$ xi) in step 9 of Algorithm 1 can be computed using Algorithm 2

When there is an edge from node k to node g labeled xi, then $\sigma(L(k) \oplus x_i) = g$ if there is an edge (k, g) out of the node k labeled x_i. But when there is no edge from k labeled x_i, then matched node label need to be changed to longest proper suffix of $L(k)$. Then corresponding node h will have edge (h, g) labeled x_i and $\sigma(L(k) \oplus x_i)$ will be equal to g. Pre-computed table (k, ch) of function $\sigma(L(k) \oplus ch)$ is maintained for every character $ch \; \epsilon \; P$ and $1 \leq k \leq s$. Algorithm 3 is used to compute this table.

To compute all the values in this table, it takes $O(s|\Sigma|)$ time. (Here $ch \; \epsilon \; \sigma$ and $1 \leq k \leq s$). With this pre-computation,we can compute $\sigma(L(k) \oplus ch)$ in $O(1)$ time for every character ch. The loop in the Algorithm 1 can be computed in $O(1)$ time. Hence, required LCS computation takes $O(nmr)$ time with $O(r|\Sigma|)$ pre-processing time.

Algorithm 1 M-STR-EC-LCS

Input: Strings $X = x_1 \cdots x_n$, $Y = y_1 \cdots y_m$ of lengths n and m, respectively, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r

Output: The length of an LCS of X and Y excluding P

1: Build a keyword tree T for P
2: **for all** i, j, k , $1 \leq i \leq n, 1 \leq j \leq m$, and $0 \leq k \leq s$ **do**
3: $f(i, m+1, k) \leftarrow 0, f(n+1, j, k) \leftarrow 0$ {boundary condition}
4: **end for**
5: **for** $i = n$ down to 1 **do**
6: **for** $j = m$ down to 1 **do**
7: **for** $k = 0$ to s **do**
8: $f(i, j, k) \leftarrow \max\{f(i+1, j, k), f(i, j+1, k)\}$
9: $q \leftarrow \sigma(L(k) \oplus x_i)$
10: **if** $x_i = y_j$ **and** $q < s$ **then**
11: $f(i, j, k) \leftarrow \max\{f(i+1, j+1, k), 1 + f(i+1, j+1, q)\}$
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** $f(1, 1, 0)$

Figure 3: Algorithm 1

Algorithm 2 $\sigma(k, ch)$

Input: Integer k and character ch

Output: $\sigma(L(k) \oplus ch)$

1: **while** $k \geq 0$ **do**
2: **if** there is an edge (k, h) labeled ch out of the node k of T **then**
3: **return** h
4: **else**
5: $k \leftarrow pre(k)$
6: **end if**
7: **end while**
8: **return** 0

Figure 4: Algorithm 2

Algorithm 3 $\lambda(k, ch)$

Input: Integer k, character ch

Output: Value of $\lambda(k, ch)$

1: **if** $k > 0$ **and** $\lambda(k, ch) = 0$ **then**
2: $\lambda(k, ch) \leftarrow \lambda(pre(k), ch)$
3: **end if**
4: **return** $\lambda(k, ch)$

Figure 5: Algorithm 3

4 K-substring LCS problem

LCS is a problem of computing matching symbols between sequences where as K-substring LCS problem (LCS-k) is a problem of computing matching non-overlapping substrings between sequences. Hence, LCS-k is a generalized version of LCS .

Subsequence is retrieved from a sequence by deleting zero or more symbols whereas substring is a consecutive part of the string. LCS problem is used as measure of sequence similarity. In LCS problem, the consecutive matches in the LCS are not adjacent to each other in the original given sequences. They can have different spacings in the given sequences. Consider following example sequences,

$A = (GTG)^{n/3}$

$B = (TCC)^{n/3}$

We observe that there are no matches that are consecutive in the given large n/3 sized sequences. The longest common subsequence separated by large number of elements in the given sequences cannot be used as accurate measure of similarity. In order to improve the accuracy of similarity measure, LCS-k problem was introduced by ensuring that the adjacent characters in the computed LCS are also adjacent in both the given sequences [3]. In this problem, maximal number of k substrings matching in the given two sequences that preserve the order of appearance are determined. LCS problem is special case of LCS-k with $k = 1$.

4.1 Problem Definition

For a given two sequences A and B, the K-substring LCS problem (LCS-k) problem is to find the maximal l such that there exist l pairs of substrings of length k (called k-strings).

Definition 1. The Longest Common Subsequence in k Length Substrings Problem (LCSk):

Input: Two sequences $A = a_1a_2...a_n$, $B = b_1b_2...b_n$ over alphabet Σ.

Output: The maximal $ls.t.$ there are l substrings, $a_{i1}...a_{i1+k1}...a_{il}...a_{il+k1}$, identical to $b_{j1}...b_{j1+k1}...b_{jl}...b_{jl+k1}$ where a_{if} and b_{jf} are in increasing order for $1 \leq f \leq l$ and where two k length substrings in the same sequence, do not overlap.

To compute LCS-k problem, we can apply LCS algorithm on the given two sequences and then backtrack dynamic programming table and mark symbols picked in the common subsequence. Then, we can verify whether these marked symbols appear in consecutive k length substrings in both the given input sequences and delete them if they does not appear. This method can find the common subsequence in k length substrings but the computed common subsequence may or may not have the optimal length. For instance, consider following two input sequences. $A = TGCGTGTG$ and $B = GTTGTGCC$

When LCS algorithm is applied on the above sequences, we may get $TTGTG$ as an output. This output contains a single non-overlapping pair matching. But we can observe another common subsequence T G T G with two pair matchings. Hence, other efficient technique for LCS-k problem were studied in paper [3].

Dynamic programming approach for LCS-k problem Let $A_{i-k+1...i}$, $B_{j-k+1.j}$ represents substrings of length k. Let (i, j) in M represents a match if $A_{i-k+1...i} = B_{j-k+1.j}$. The basic idea of this approach is based on following recurrence.

$$LCSK_{i,j} = max \begin{cases} LCSK_{i,j-1} \\ LCSK_{i-1,j} \\ LCSK_{i-k,j-k} + kMatch(i,j) \end{cases}$$

Here, $LCSK_{i,j} = 0$ if $i < k$ or $j < k$.

The report of matches are started from the end of the k-string instead of start of the k-string. This approach runs in $O(n^2)$.

4.2 Preliminaries

A Match of k-consecutive characters is termed as k-matching. Let $k - match(i, j)$ represents a match of consecutive characters of length k in the prefix $A[1...i+k1]$ and $B[1...j + k1]$ of sequences A and B. $k - match(i, j)$ is defined as follows:

$$kMatch(i,j) = \begin{cases} 1 & if\ a_{i+f} = b_{j+f},\ for\ every\ 0 \leq f \leq k-1; \\ 0 & otherwise; \end{cases}$$

Let $candidates(i, j)$ represents list of all possible longest possible subsequences with k matchings in the prefix $A[1...i+k1]$ and $B[1...j+k1]$. Then $pred(i, j)$ represents list of last k-matchings in $candidates(i, j)$. Let $p\ \epsilon\ pred(i, j)$ from $candidates(i, j)$, then length of p gives the number of k-matchings.

For example, consider the below two sequences, Let $candidates(5, 3)$ be common subsequences in pairs of $B[1...4] = GTTG$ and of $A[1...6] = TGCGTG$, thus, $candidates(6, 4)$ contains TG, GT. We have two options to obtain this output i.e. a_1a_2 or a_5a_6. Hence, $pred(i, j) = (1, 3), (5, 3), (4, 1)$. Predecessors helps for backtacking of the longest common subsequence in k length substrings. The LCS can be obtained by following recursive formula,

4.3 Dynamic Programming solution

Dynamic programming approach is about choosing the maximum of the prefixes. If the length of any prefixes is of same length, then choice of the prefixes is not significant

in this case and it will not reflect the future selections. But in LCS-k problem, the choice will effect the future selections. Hence, predecessors need to be saved in order to backtrack the optimal solution.

Let $LCS-k(i,j)$ denote the longest common subsequence in k-length substrings in the prefix $A[1...i+k1]$ and $B[1...j+k1]$.

The standard LCS algorithm involves finding of maximum of three options of prefixes of LCS. If these options of prefixes are of same length, then selecting one of the options will not effect the future matches. But in LCS-k problem, the choice of the prefixes of same length will have an effect on future matches. For example, consider the two sequences A and B with k value = 2.

	1	2	3	4	5	6	7	8
A =	T	G	C	**G**	**T**	G	**T**	**G**
B =	**G**	**T**	T	G	**T**	**G**	C	C

Figure 6: Example[2]

$LCSk[3,3]$ value is equal to 1 due to a match GC (a_1a2 to b_2b_3) or CG (a_2a3 to b_3b_4). The common subsequences of both of these options has same length, but selecting match GC (a_1a2 to b_2b_3) is optimal as it gives further match GT (a_3a4 to b_4b_5). The second option CG cannot be selected as we cannot subsequence due to overlapping. Hence, we require to store all predecessors in order to backtrack the optimal solution. But this information will increase exponentially when the LCS computation proceeds. Hence, the following lemma given in [2] states that only maximal of previously computed common subsequences is required.

In this approach proposed in [2], a two dimentional table of size $(nk+1)^2$ is maintained.

After computing, $LCSk[i,j]$, the $pred(i,j)$ is immediately computed by following formula,

If $LCSk_{i,j-1} = LCSk_{i-1,j} = LCSk_{i-2,j-2+1}$, and $kMatch(i,j) = 1$ then $pred(i,j) = pred(i,j-1) \cup pred(i-1,j) \cup (i,j)$.

If $LCSk_{i,j-1} = LCSk_{i-1,j}$ and $kMatch(i,j) = 0$ then $pred(i,j) = pred(i,j-1) \cup pred(i-1,j)$.

The computation of K-match function requires to find k-matchings at every table entry $LCSk(i,j)$. But the matches are enlarged at the diagonal of $LCSk$ table (cell[i, j] to cell [i+ 1, j+ 1]). Hence a diagonal counter *dcount* is computed which gives the length of longest match in the suffixes of $A[1...i]$ and $B[1...j]$.

For every table cell (i,j), a diagonal counter dcount is computed according to the following formula,

$$dcount(i,j) = \begin{cases} 1 + dcount(i-1, j-1) & if\ a_i = b_j \\ 0 & if\ a_i \neq b_j \end{cases}$$

Following figure illustrates the LCS-k(i,j) computation for sequences $A = CTGCTTTG$ and $B = CTTGCTTT$ with $k = 2$.

		1 C	2 T	3 T	4 G	5 C	6 T	7 T	8 T
1	C	1 (1,1) dc=1	1 (1,1) dc=0	1 (1,1) dc=0	1 (1,1) dc=0	1 (1,1)(1,5) dc=1	1 (1,5) dc=0	1 (1,5) dc=0	- - dc=0
2	T	1 (1,1) dc=0	1 (1,1) dc=2	1 (1,1)(2,3) dc=1	1 (1,1) dc=0	1 (1,1)(1,5) dc=1	1 (1,5) dc=2	1 (1,5) dc=1	- - dc=1
3	G	1 (1,1) dc=0	1 (1,1) dc=0	1 (1,1) dc=0	1 (1,1) dc=2	1 (1,1)(1,5) dc=0	1 (1,5) dc=0	1 (1,5) dc=0	- - dc=0
4	C	1 (4,1) dc=1	1 (4,1) dc=0	1 (1,1),(4,1) dc=0	1 (4,1) dc=0	2 (4,5) dc=3	2 (4,5) dc=0	2 (4,5) dc=0	- - dc=0
5	T	1 (4,1) dc=0	1 (4,1),(5,2) dc=2	1 (4,1) dc=1	1 (4,1) dc=0	2 (4,5) dc=0	2 (4,5),(5,6) dc=4	2 (4,5),(5,7) dc=1	- - dc=1
6	T	1 (4,1) dc=0	1 (4,1),(6,2) dc=1	1 (4,1) dc=3	1 (4,1) dc=0	2 (4,5) dc=0	2 (4,5),(6,6) dc=1	3 (6,7) dc=5	- - dc=2
7	T	1 (4,1) dc=0	1 (4,1) dc=1	2 (7,3) dc=2	2 (7,3) dc=0	2 (7,3),(4,5) dc=0	2 (7,3)(6,6) dc=0	3 (6,7) dc=2	- - dc=6
8	G	- dc=0	- dc=0	- dc=0	- dc=3	- dc=0	- dc=0	- dc=0	- dc=0

Figure 7: Numbers denote length of the common subsequence. The pairs in parenthesis gives all possible predecessors. The diagonal counter dcount is denoted by dc^2

We can compare $dcount[i-k+1][j-k+1]$ with k while computing $LCSk(i,j)$ instead of using $k-match$ function. With the use of $dcount$, the $kmatch$ is reduced to constant time operations[2]. LCS-k can be computed in $O(n)$ time with $O(kn)$ space where n is the length of the sequences and k is length of the substring.

5 Variable gapped LCS

Subsequence is retrieved from a sequence by deleting zero or more symbols whereas substring is a consecutive part of the string. There exist gaps between the characters picked for the LCS. Constraints on these gaps was introduced by the gapped LCS variant.

Initially fixed gap LCS (FGLCS) problem was introduced by Iliopoulos and Rahman [4]. In this problem, fixed gap value k is given and the distance between two consecutive matches is required to be limited to at most k+1. This problem is solved in $O(nm)$ when the gap constraints are fixed to single integer (Here, n and m are lengths of two input sequences).

This gapped LCS problem has applications in molecular biology and genetics. It is used in motif pattern finding problems especially in scenarios where all the positions of the pattern are not important i.e some positions of the pattern can be filled by any character. For instance, consider sequences, $A = RCLPCRR$ and $B = RPPLCPLRC$, the detected LCS with gap constraint $k = 2$ is $R..L..C..R$. Here, in the study of protein sequences, the gaps(dots) '.' in computed LCS are filled with any amino acids. Previous works were focused on LCS with fixed gaps. But in real applications, the gaps are not fixed. Hence, this project studies the LCS problem with variable gaps (VGLCS) proposed in paper Yung-Hsing et al in [1]. The variable gapped LCS problem provides more flexibility in analyzing sequences.

The paper Yung-Hsing et al in [1] gives two algorithms for variable gapped LCS problem. Initially, a simple algorithm is given which is adapted from the fixed gapped LCS algorithm [2] and this algorithm runs in $O(n^2m^2)$ time. Then, the algorithm is modified to improve the time complexity to $O(nm)$ using the technique called incremental suffix maximum query (ISMQ). An optimal approach is proposed for the incremental suffix maximum query (ISMQ) problem and then this approach is used to solve variable-gapped LCS problem.

5.1 Preliminaries

In this problem, variable gap constraints are defined by two gap functions G_A and G_B. Here, A and B are two input sequences. $G_A(i)$ and $G_B(j)$ are the two gap constraints of i^{th} character of A and j^{th} character of B respectively. The gap constraint is an integer value which limits the distance between the character and its previous character in the sequence. Suppose when the i^{th} character in A is picked in the common subsequence, then its previously picked character is bounded by the distance $G_A(i) + 1$. For example, if the gap constraint $G_A(i) = 3$ for some i^{th} character in A, then the gap between this character and previously picked character for LCS must be at most 3. When the gap functions $G_A(i) = k$ and $G_B(j) = k$, then we can use

the variable gapped LCS approach to solve the fixed gapped LCS problem. Following figure illustrates the variable-gapped LCS problem.

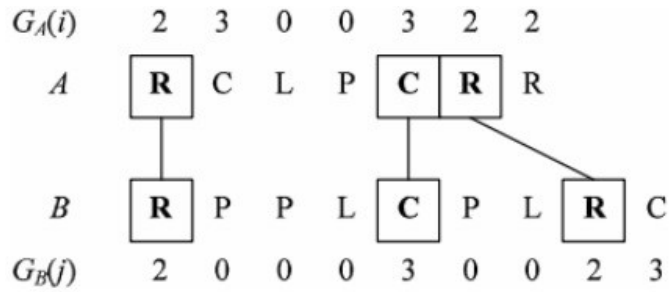

Figure 8: Example in [3] for illustrating the VGLCS between two sequences $A = RCLPCRR$ and $B = RPPLCPLRC$ with G_A = [2, 3, 0, 0, 3, 2, 2] and G_B = [2, 0, 0, 0, 3, 0, 0, 2, 3]

5.2 Problem Definition

Given a sequence S, let $S[i]$ denote the i^th character in S,

Definition 1: Variable gap subsequence (VGS): Given a sequence S and a gap subsequence of length and its gap function G_S, a subsequence $S = S[i_1], S[i_2], , S[i_p]$ is called a VGS of length p in S if $i_x - i_{x1} \leq G_S(i_x) + 1$, for $2 \leq x \leq p$ [1].

Definition 2. The variable gap common subsequence (VGCS): Given two sequences A and B with their gap functions G_A and G_B , a sequence Z is a VGCS of A and B if Z is both a VGS of A with G_A and a VGS of B with G_B [1].

Given two sequences A and B with their gap functions G_A and G_B, the variable-gapped LCS (VGLCS) problem is to find the longest matching list that forms a variable gap common subsequence between A and B [1].

5.3 Naive Algorithm

The following two $n \times m$ arrays are maintained in this algorithm.

$F[i][j]$: It indicates maximum length of the variable-gapped common subsequence between $A[1, i]$ and $B[1, j]$ whose last common character is $A[i]$ and $B[j]$. $F[i][j]$ is irrelevant when there is no match, hence it is stored as zero.

$V[i][j]$: It indicates the length of the longest matching list that forms a variable gap common subsequence between $A[1, i]$ and $B[1, j]$.

Let $M_G(i, j)$ denote the set of matching indices (i', j') such that

$$1 <= i' <= i - 1, 1 <= j' <= j - 1, A[i'] = B[j'], i - i' <= G_A(i) + 1$$

and j-j'¡= $G_B(j)$ + 1 (1)

This is straight forward implementation of following recursive formula.

$$F[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0; \\ 0, & \text{if } A[i] \neq B[j]; \\ max\{_{(i',j')\epsilon M_G(i,j)} F[i'][j']\} + 1, & \text{if } A[i] = B[j]; \end{cases}$$

$$V[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0; \\ max\{V[i-1]V[j], V[i][j-1]\}, & \text{if } A[i] \neq B[j]; \\ max\{F[i][j], V[i-1][j], V[i][j-1]\}, & \text{if } A[i] = B[j]; \end{cases}$$

Consider following example,

Let $X = ABCCDEFGACD$ and $Y = AFCGFCABD$ be two given sequences.

$X[1] = Y[1] = A$ is a match. If we consider $X[7] = Y[2] = F$ as a second match, then we get subsequence $S = AF$ common to both $X[1..7]$ and $Y[1..2]$. But when the gap constraint is $K = 4$, then we cannot continue with the second match ($X[7] = Y[2] = F$) after the first match ($X[1] = Y[1] = A$). This is because it violates the gap constraint. We need to start with a new common subsequence instead of continuing with the previously found subsequence. To handle this, we compute and store two values namely $F[i][j]$ and $V[i][j]$.

For every character in the sequence, the gap can be in the range from 0 to G_A+1. Hence, we need to take maximum of all the possible options and store in $F[i][j]$. This is a local update. $V[i][j]$ is for global update to store the information of the LCS found so far. When the local LCS turns into the global one, then the corresponding global value in $V[i][j]$ changes.

In the worst case, the set of matches in $M_G(i, j)$ can be $O(nm)$. Hence, the time complexity of this algorithm is $O(n^2m^2)$.

5.4 Improved Algorithm

In the improved algorithm, Variable-gapped LCS problem will be related to Incremental sequence maximum queries (ISMQ) problem. Then an optimal approach is proposed to solve incremental suffix maximum query problem and then this approach is used to find an efficient algorithm for variable-gapped LCS problem which runs in $O(nm)$ time.

5.5 Incremental sequence maximum queries problem

Incremental suffix maximum queries (ISMQ) is a problem of finding maximum number in the suffix of given string array of numbers. In a given string array D, $SMQD(i)$ is the maximum number in the suffix $D[i, |D|]$ of $D(1 <= i <= |D|)$. If the input array D is known, then we can scan D reversely and store the $SMQD(i)$ values in $O(|D|)$ time. Then, each $SMQD(i)$ value can be found out in $O(1)$ time. But here, D is assumed to be incremental. For example, let D be an array of integers $10, 3, 7, 2, 5$. $SMQD(i)$ is $10, 7, 7, 5, 5$. When new element is added in D (for e.g. number 8), then $D = 10, 3, 7, 2, 5, 8$ and $SMQD(i)$ changes to $10, 8, 8, 8, 8, 8$. Hence, $SMQD(i)$ may vary as D grows. Therefore, one time scanning of D is not suitable for finding $SMQD(i)$.

The above mentioned recursive formula in figure 2 of naive algorithm defines the use of two $n \times m$ arrays $V[i][j]$ and $F[i][j]$ to store the information of variable-gapped LCS. The improved algorithm considers following two $n \times m$ arrays to store the required maximum values.

$Col[i][j]$: It indicates maximum value in one dimensional subarrays $F[i - 1 - G_A(i), i - 1][j, j]$

$All[i][j]$: It indicates maximum value in one dimensional subarrays $Col[i, i][j - 1 - G_B(j)$, j-1].

In other words, $All[i][j]$ stores maximum value of the two-dimensional $(G_A(i)+1)$ $\times$(G$_B(j)$+ 1)$rectangle$F[i-1-G$_A(i), i-1][j-1-G_B(j), j-1]$.

Based on these variables, following algorithm is proposed to find the variable-gapped LCS problem.

Algorithm Algorithm for Finding VGLCS

for $i = 1$ to n **do**
  for $j = 1$ to m **do**
    $Col[i][j] \leftarrow max\{F[i-1-G_A(i)][j], F[i-G_A(i)][j], \cdots, F[i-1][j]\}$.
    $All[i][j] \leftarrow max\{Col[i][j-1-G_B(j)], Col[i][j-G_B(j)], \cdots, Col[i][j-1]\}$.
    if $A[i] = B[j]$ **then**
      $F[i][j] \leftarrow All[i][j] + 1$.
      $V[i][j] \leftarrow max\{F[i][j], V[i-1][j], V[i][j-1]\}$.
    else
      $F[i][j] \leftarrow 0$.
      $V[i][j] \leftarrow max\{V[i-1][j], V[i][j-1]\}$.
    end if
  end for
end for
Retrieve the VGLCS by tracing $V[n][m]$.

Figure 9: Improved variable-gapped LCS algorithm[1]

Following Lemma is given in paper in [1] to prove that improved algorithm computes VGLCS.

Lemma 1: Algorithm 1 solves the VGLCS problem in O(nm) time, provided that each $Col[i][j]$ and $All[i][j]$ can be determined in $O()$ time.

In order to verify the correctness of this lemma, it is proved that improved algorithm is an implementation of above mentioned recursive formula in figure 2. Consider both the cases $A[i] = B[j]$ and $A[i] \neq B[j]$ in the improved algorithm in figure 3.

Case 1 : When $A[i] = B[j]$, $F[i][j]$ value is given by $All[i][j]$, which stores the maximum length of the variable-gapped common subsequence between $A[1, i]$ and $B[1, j]$ whose last common character is $A[i]$ and $B[j]$. $V[i][j]$ value is computed in the same way it is computed in the recursive formula of simple algorithm in figure 2.

Case 2 : When $A[i] \neq B[j]$, Both $F[i][j]$ and $V[i][j]$ are computed using recursive formula of simple algorithm in figure 2.

From this, it is evident that improved algorithm is an implementation of the recursive formula of simple algorithm in figure 2. Hence, if each $Col[i][j]$ and $All[i][j]$ in the loop is determined in $O(\alpha)$ time, then $V[n][m]$, $F[n][m]$, $Col[n][m]$, and $All[n][m]$ can be determined in $O(\alpha nm)$ time.

5.6 Use of ISMQ in the improved algorithm

Incremental suffix maximum queries (ISMQ) is a problem of finding maximum element in the suffix of given array of numbers. The arrays Col[i][j] and All[i][j] in the algorithm stores the required maxima. When each column of array F and row of array Col are considered as incremental array of numbers, then ISMQ can be used to compute values of arrays $Col[i][j]$ and $All[i][j]$. Hence, variable-gapped LCS problem can be related to ISMQ problem.

With the help of ISMQ, variable-gapped LCS can be computed in $O(\alpha nm)$ when each $Col[i][j]$ and $All[i][j]$ can be computed in $O(\alpha)$ time. ISMQ can be handled using a balanced binary search tree or a van Emde Boas tree which reduces α to $logn$ or $loglogn$ respectively.

But Yung-Hsing et al in [1] proposes an efficient technique to handle ISMQ which further reduces $O(\alpha)$ to $O(1)$. This technique is based on the union-find problem.

5.7 Union-find problem

Union-find problem (also called as disjoint set union problem) is a classical problem in computer science. Union-find data structure (data structure used to solve union-find problem) has following three operations.

1. $make(x, C)$ 2. $find(x)$ 3. $merge(x, y, C)$

Function $make(x, C)$ is used to create new set x named C. Function $find(x)$ gives the name of the set and function $merge(x, y, C)$ is used to combine two different sets

(with values x and y) into one set. Here, no element can belong to two different sets. This helps in preserving uniqueness of the set.

ISMQ problem can be reduced to Union-find problem. Here, indices of the elements in the number array of ISMQ problem are considered as the elements of the set and the maximum value of the elements is used as the name of the set. Hence, when $find(x)$ function is invoked it gives the name of the set which is the required maximum value. This strategy can be used to compute ISMQ queries.

Let D be an incremental array of numbers and d_i represents suffix maximum number in the suffix $D[i, |D|]$ of D (i denotes index of the element). $LetW = w1, w2, , w|W|$ be a list of increasing indices.Consider an example set D = 10, 3, 7, 2, 5, 8. Following figure 4 from left to right explains how the set D is processed in Union-find problem. Here, numbers in the circles represent the indices of the numbers of array D. Function $make(x, C)$ is used to create a node for every index. Elements belong to the same set when their respective nodes are surrounded by the same bold oval. Suffix maximum number is used as the name of the set. The name of the set is denoted by the bold number beside the bold oval.

Function $unite(x, y, C)$ (or $merge(x, y, C)$) is invoked when d_i ¿= $d_w|W|$ (d_i is the suffix maximum for each element in $d_w|W|$). The two elements are merged into one set and the set is named after the maximum element. This arrangement ensures that the set is named by suffix maximum number in the given array D. Function $find(x)$ retrieves the name of the set. Hence, $find(x)$ can be used to find the suffix maximum number. In this way, we can use union-find data structure to solve ISMQ problem. This process is proposed in Algorithm 2 in figure 5.

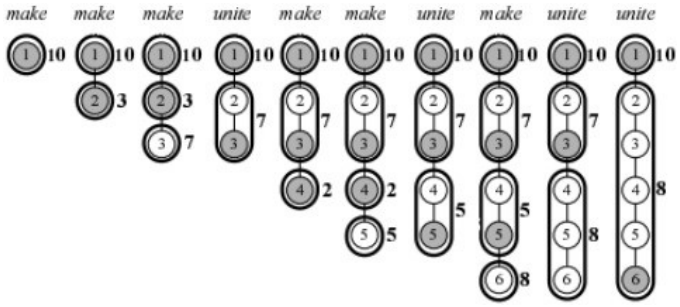

Figure 10: Example of Union-find problem[1]

Following Algorithm is used to compute the $Col[i][j]$ and $All[i][j]$ using union-find strategy.

With this technique, ISMQ operation time can be reduced from α to 1. Hence, the time complexity is reduced to $O(nm)$ to find variable-gapped LCS.

Algorithm Answering ISMQ with Union-find Operations

Create an empty union-find data structure, and an empty W with $|W| = 0$.
$i \leftarrow 1$.
while there exists input d_i **do**
  $make(i, d_i)$.
  while $|W| > 0$ and $d_i \geq d_{w_{|W|}}$ **do**
    //If $d_i \geq d_{w_{|W|}}$, then d_i is the suffix maximum for each element in $d_{w_{|W|}}$.
    $unite(w_{|W|}, i, d_i)$.
    Delete $w_{|W|}$ from W.
  end while
  Append i to W.
  while there exists a query $SMQ_D(j)$ **do**
    Report $SMQ_D(j) = find(j)$.
  end while
  $i \leftarrow i + 1$.
end while

Figure 11: Algorithm to solve ISMQ problem using union-find data structure[1]

6 Experiments

Experiments are implemented using system with Memory: 5.0 GiB and Processor: AMD A105750M APU with Radeon(tm) HD Graphics 4. The variants of LCS: Variable gapped LCS, k-substring LCS and Constrained LCS: multiple exclusion constraints are developed in Java programming language.

The algorithms are evaluated using random strings of length 5 to 5000 over alphabets of size 4 (e.g. nucleotide sequences input structure: A,C,G,T)

For evaluation, the three variants Multiple exclusive constrained LCS (M-STR-EC-LCS), K-substring LCS (LCS-k) and Variable gapped LCS (VGLCS)are implemented and compared with the LCS dynamic approach shown in Fig. 12.

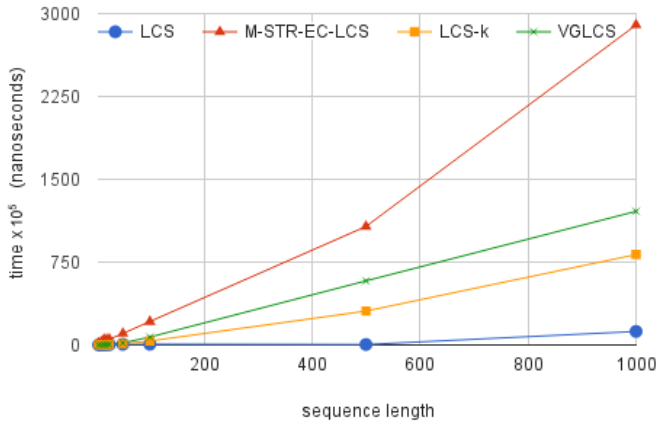

Figure 12: $|P| = 5$, $k = 2$

Fig. 12 gives results with constant number of constraints $|P| = 5$ in M-STR-EC-LCS, $k = 2$ (k is length of substring in LCS-k) and variable gap range [0-2] in VGLCS.

6.1 Multiple exclusive constrained LCS

In this problem, the set of constraints P are processed using a Trie data structure (keyword tree).

Trie:

It is a ordered tree data structure. It is kind of a search tree and can be used as a prefix tree. Unlike binary search trees, the keys are not associated with respective nodes itself but instead associated with the position of the respective node. Also, the values are represented by the leaf nodes.

In Trie, searching of data takes $O(n)$ time in worst case, where n is the length of the search string.

Aho-Corasick Algorithm[10] is used to implement Trie data structure. It is string searching algorithm. It builds finite state machine to represent keyword tree and various links between the internal nodes. These links helps in transition to other nodes that share same prefix and also links such as failure links allows transition between failed string matches.This algorithm builds a keyword tree in linear time.

The results in the figure 13 represents the measurements of average running time of Multiple substring exclusive constrained LCS (M-STR-EC-LCS) with different values of $|P|$ (number of constraints) with each constraint string of length = 5. The average running time of M-STR-EC-LCS increases with the increase in the number of constraints.

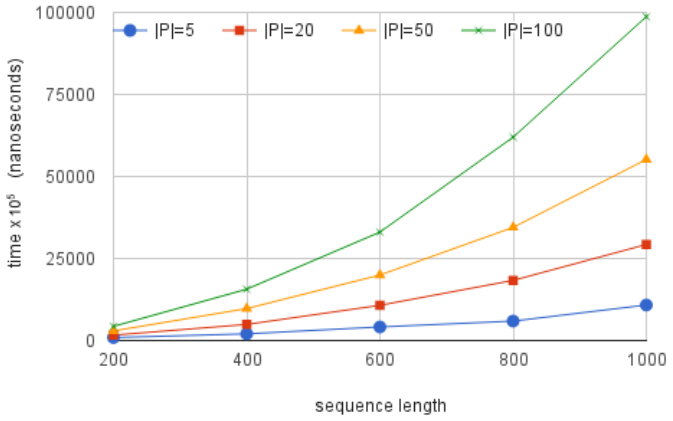

Figure 13: Constraint string length = 5

The average running time of M-STR-EC-LCS with different values of length of the constraint strings and constant number of constraints is measured in figure 14. Here, r represents the total length of the constraint strings. In this figure 14, number of constraints $|P| = 5$ is taken. First line represents the average running time with constraint string length = 3 and $|P| = 5$. Hence, total length of constraint strings r = 15. Similarly, other series line represents average running time with $|P| = 5$ and constraint strings of length 5 (r = 25) and 10 (r = 50) respectively.

From figure 14, observed that, with increasing length of the sequences, there is larger magnitude of difference in average running times when constraint string length is increased. Next, number of constraints is increased to $|P| = 20$ and similar values of constraint string lengths in figure 14 are used to measure the average running time in figure 15.

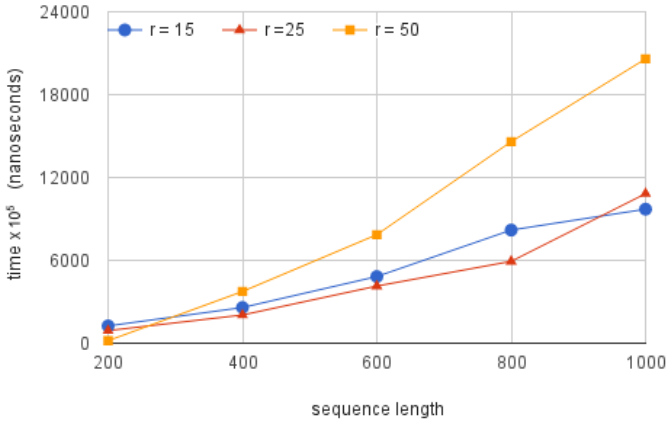

Figure 14: $|P| = 5$

Observed that the average running time increases with the increase in the number of constraints and when the increase in the total constraint string length r is small, the average running time is not affected.

6.2 LCS-k

Two $n \times m$ tables are maintained for *dcount* and LCS length. *dcount* is used to store the information about the k-matches.

LCS-k is evaluated by measuring average running time with different values of k (length of substring) shown in figure 16. Fig. 16 shows results for k = 10 and k = 100.

From this analysis, the average running time is not effected with increase in the value of k for the sequences of length ≤ 800. The running time depends on the number of k-matches rather than the value of k. For the sequences of length > 800, if there is larger increase in the k value, then average running time increases.

6.3 Variable gapped LCS

Two $n \times m$ tables are maintained for local update and global update of length of LCS with variable gaps. The average running times with increasing range of gap values are evaluated. Figure 17 shows the results gap value in range [0-2] and range [0-9]. The high peaks in the graph are due to the higher gap value. The gap values G are not fixed, we need to compute LCS such that the gap between respective match and its previous match is not greater the gap value G. Consider gap value as $G = 3$, then the gap can be 0 or 1 or 2. We need to evaluate all these gap options and select

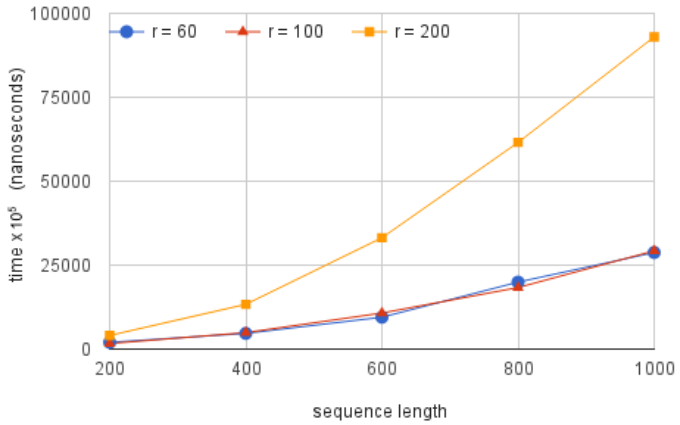

Figure 15: $|P| = 20$

gap such that we get optimal solution. Hence, higher the gap value G, higher is the number of options to be evaluated which results in more number of comparisons.

To analyze the effect of increase in gap values on average running time, fixed gaps are considered. Fig. 18 gives the average running times with gaps = 2, 10, 50. When there is high increase in the gap values, there is larger magnitude of difference in the average running time.

In all cases in the variants M-STR-EC-LCS, LCS-k, VGLCS, the increase in their respective parameters resulted in higher magnitude of increase in running time for the sequences greater than 600.

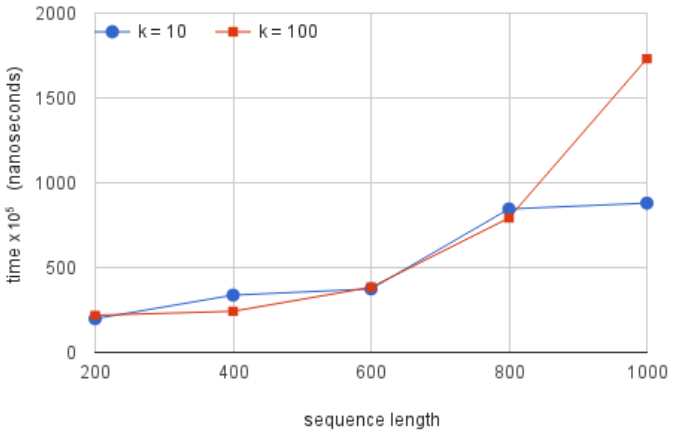

Figure 16: k = 10 and k = 100

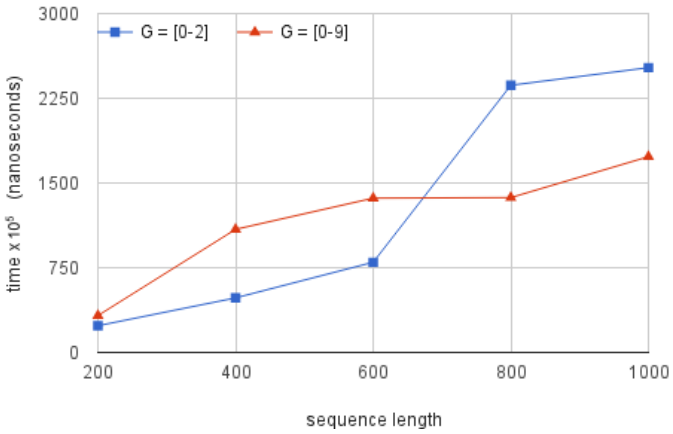

Figure 17: gap range [0-2] and range [0-9]

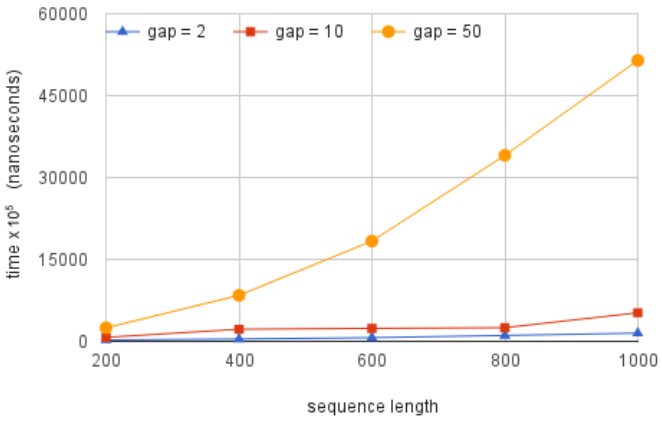

Figure 18: Fixed gaps = 2, 10, 50

7 Conclusion

There are many research works on improving the time complexity of LCS problem. Apart from these, research on variants which represent the modified problem definition of LCS received much attention due to various applicatons in the fields of genomics and computational molecular biology. This study explains how the LCS dynamic programming approach is modified or how efficient techniques of different problems are incorporated in the dynamic approach to solve the variants of the LCS problem.

In this project, the performance of the multiple exclusive constrained LCS variant is evaluated by varying the number of constraints and the length of the constraint strings. The k-length substring LCS problem is evaluated by increasing the value of k and the variable gapped LCS problem is evaluated by increasing the range of gap values. All these variants are evaluated by increasing length of the input sequences. With the proposed techniques, the Variable gapped and LCS-k variants can be solved in linear time and Constrained LCS in can be solved in polynomial time. These variants can be used to build a generalized model of similarity measure which enables to find the similarity between sequences in different purpose applications.

Substring exclusive constraints, variable gap constraints and maximal k-substring matches are introduced by these variants.

The LCS similarity measure tools such as Blast uses dynamic programming. These tools have limited purpose usage. These variants cane be used to build a generalized model of a similarity measure which can cover varied purposes.

7.1 Current Status

Extended versions of LCS variants are implemented i.e. Single substring exclusive constraint problem is extended by multiple substring exclusive constraints, Fixed gapped LCS is extended by Variable gapped LCS and Longest common substring problem is extended by maximal k-length substrings problem. The improved methodologies of these extensions to obtain linear time solutions are studied.

7.2 Lessons Learned

In this project, learned about the background work and improved approaches of different variants of LCS. The traits of the variants of LCS and use of techniques like ISMQ, Keyword tree and failure function in Knuth-Morris Pratt to improve the time complexity are studied. Studied different algorithms to solve the problems and learnt how the traits of the problem are used in the design of their respective efficient solving techniques. Also, understood that the issues caused when straight

forward implementation of LCS dynamic approach is applied to solve the variant problems. With this improved techniques, the changes in the parameters of the respective variants had a greater effect on running time for sequences greater than 600.

7.3 Future Work

All these experiments are conducted for two sequences. Further, this work can be extended in finding similarity between multiple sequences. In order to achieve same performance for multiple sequence, the variants can be implemented on multiple processors. Also, other variants of different applications such as inclusive constrained LCS can be incorporated in the study.

References

[1] Yung-Hsing Peng, Chang-Biau Yang.: *Finding the gapped longest common subsequence by incremental suffix maximum queries, Inf. Comput. 237:,* pages 95-100, 2014.

[2] G. Benson, A. Levy, and B. R. Shalom. *Longest Common Subsequence in k-length substrings, CoRR, vol. abs/1402.2097,* 2014.

[3] Zhu D., Wang L., Tian J., and Wang X. *A Simple Polynomial Time Algorithm for the Generalized LCS Problem with Multiple Substring Exclusive Constraints, IAENG International Journal of Computer Science, vol. 42, no.3,* pages 214-220, 2015.

[4] C.S. Iliopoulos, M.S. Rahman, *Algorithms for computing variants of the longest common subsequence problem, Theor. Comput. Sci. 395,* pages 255267, 2008.

[5] J. W. Hunt and T. G. Szymanski. *A fast algorithm for computing longest common subsequences. Communications of the ACM, 20(5):* pages 350353, 1977.

[6] R.A. Wagner and M.J. Fischer, *The string-to-string correction problem, J. ACM 21:* pages 168173, 1974.

[7] William J. Masek, Mike Paterson, *A faster algorithm computing string edit distances, J. Comput. Syst. Sci. 20 (1) (1980)* pages 1831.

[8] J. Eugene W. Myers, *An O(ND) difference algorithm and its variations, Algorithmica 1 (2) (1986),* pages 251266.

[9] Narao Nakatsu, Yahiko Kambayashi, Shuzo Yajima, *A longest common subsequence algorithm suitable for similar text strings, Acta Inf. 18 (1982),* pages 171179.

[10] J. Aho A.V., Corasick M.J., *Efficient string matching: an aid to bibliographic search, Commun ACM 18(6), 1975,* pages 333-340.