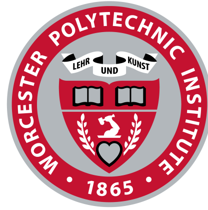


Exploring Longest Common Subsequences and the Chvátal-Sankoff Constants

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in
Computer Science and Mathematical Sciences

Project Team

Chase Miller
Andrew Salls
Duncan Soiffer

Project Advisors

George T. Heineman
Daniel Reichman
Gábor N. Sárközy

Submitted: April 2024

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects.

Abstract

We provide an introduction to the longest common subsequence (LCS) problem and the Chvátal-Sankoff constants, and obtain new results related to these areas. By improving upon a previous lower-bound algorithm, we obtain state-of-the-art lower bounds for all but one Chvátal-Sankoff constant, with particular attention paid to the Chvátal-Sankoff for two binary strings. Accompanying this, we also prove various properties describing the effect of modifying a pair of strings on their longest common subsequence. We additionally create a web application with interactive visualizations designed to allow users to learn and explore these properties and other facets of the LCS problem.

Acknowledgements

This research was performed using computational resources supported by the Academic & Research Computing group at Worcester Polytechnic Institute. We would also like to thank James Kingsley, Director of High Performance Computing and Faculty Support, for his advice and assistance in understanding and best taking advantage of the inner structure of the Turing cluster.

Contents

1 Introduction **2**

2 Background and Related Work **4**
2.1 Definitions and Context 4
2.2 Calculating Longest Common Subsequences 6
2.3 Applications 8
2.3.1 Computational Biology 8
2.3.2 Computational Linguistics 9
2.3.3 Other Fields 10

3 Improving Bounds on the Chvátal-Sankoff Constants **12**
3.1 The Chvátal-Sankoff Constants 12
3.2 History 14
3.3 The Kiwi-Soto Algorithm 16
3.4 Algorithm Modifications 20
3.4.1 Parallelization 20
3.4.2 Indexing 20
3.4.3 Array Reductions and Symmetries 21
3.4.4 Sequential Memory Access 21
3.5 Results 24

4 LCS Properties and BLISS Playground **27**
4.1 Foundations and Rationale 27
4.2 Properties of Longest Common Subsequences 28
4.2.1 General Properties 28
4.2.2 Edit Operation Properties 30
4.2.3 Distribution Properties 33
4.3 BLISS Playground 33
4.3.1 Edit Properties Explorer 35
4.3.2 Matrix Builder 37
4.3.3 Derivation and Configuration Visualizer 41
4.3.4 Distribution Explorer 43
4.4 Implementation Details 45

4.4.1 Edit Properties Explorer 45
4.4.2 Matrix Builder 48
4.4.3 Derivation and Configuration Visualizer 49
4.4.4 Distribution Explorer 50

5 Conclusion and Future Work **51**
5.1 Future Work 52

A Complete Results for General $\gamma_{\sigma,d}$ Constants **59**

B Proofs of LCS Properties **62**

Chapter 1

Introduction

The longest common subsequence (LCS) problem arises frequently in computer science. Given two or more strings, the longest string that is a subsequence of all of those strings is called the longest common subsequence of those strings. For instance, *abba* is an LCS of the strings *ababa* and *acbbca*. Note that the input strings do not have to match in length. A group of strings may also have more than one longest common subsequence. As an illustration, the strings *abac* and *babc* have two longest common subsequences: *abc* and *bac*. Longest common subsequences play a pivotal role in various domains such as computational biology [27, 41], cryptography [13, 40], and computational linguistics [34, 26, 37]. Understanding longest common subsequences and the statistics surrounding them provides crucial insights into the similarities and differences between sequences of symbols.

One major area of active research in this field is determining the expected length of the LCS of two uniformly random strings, divided by the length of the strings, as their lengths tend to infinity. These values are known as the Chvátal-Sankoff constants, with one constant for each alphabet size (or one constant for each pair of alphabet size and string set size, in the extended notation) [14, 30]. While the values of these constants are not known, it is known that each one converges to a specific numerical value as the lengths of the strings grows to infinity, and there have been extensive efforts in the literature to improve upper and lower bounds on their values.

During the course of this project, we achieved several significant results related to the LCS problem and the Chvátal-Sankoff constants, which we have divided into the following categories:

Improved Lower Bounds on the Chvátal-Sankoff Constants We demonstrate new best lower bounds for the Chvátal-Sankoff constants, improving upon *all but one lower bound* previously calculated in the literature. Our research focuses primarily on improving the lower bound for the Chvátal-Sankoff constant for pairs of binary strings, but we also extend our analysis to encompass arbitrary alphabets and varying numbers of strings, thereby providing improved bounds applicable to a broader spectrum of data. This extension is significant as it addresses real-world scenarios where data may not conform strictly to binary representations, enabling more accurate assessments in fields such as computational biology and anomaly

detection.

Properties of Longest Common Subsequences We give a summary of some properties of longest common subsequences, accompanied by their proofs in the appendix. While a number of the properties we present were already known, we also believe that, to the best of our knowledge, we prove several new properties concerning single and two-character edits and the occurrences of longest common subsequences of particular lengths. By investigating the complicated links between sequences and their LCS, we hope to contribute to the development of more efficient algorithms for calculating Chvátal-Sankoff constants and to further understanding of the LCS problem more generally.

BLISS Playground Lastly, we introduce BLISS (Binary Longest Inter-String Subsequence) Playground, an interactive web application with visualizations and puzzles to help users better understand both the longest common subsequence problem and its many interesting properties. BLISS Playground aims to provide an interactive tool for visually exploring the longest common subsequence problem beyond just its dynamic programming solution, both for educational and research purposes.

In the following sections, we present an outline of our research's scope and objectives, as well as discuss the theoretical foundations, techniques, experimental results, and implications of our findings.

Chapter 2

Background and Related Work

2.1 Definitions and Context

To introduce the longest common subsequence problem, we begin by defining some basic concepts. A mathematically well-versed reader may find it possible to skip this section.

Definition 1. A ***set*** is "a well-defned collection of objects" [21, p. 1]. "The objects in a set are called its ***elements*** or ***members***... The order of describing a set doesn't matter, nor does repetition of its members" [43, pp. 3, 4]. A set can be denoted in one of two ways:

1. Roster Notation: A set can be denoted by listing its elements within braces, separated by commas. For example, the set containing the elements a, b, and c can be denoted as
$$\{a, b, c\} = \{a, a, b, c, b\} = \{a, c, b\}$$
2. Set-Builder Notation: A set can also be denoted by specifying the properties that its elements must satisfy.
$$\{x \mid x \text{ is positive, even, and less than } 10\} = \{2, 4, 6, 8\}$$

Definition 2. A ***sequence*** is "a list of objects in some order" [43, p. 6]. Unlike a set, the repetition and arrangement of the elements of a sequence is significant. For our purposes, we additionally provide the constraint that all of the elements of a sequence belong to the same set. We denote a sequence by listing its elements within parentheses, separated by commas.

$$(a, b, c) \neq (a, a, b, c, b) \neq (a, c, b)$$

Using these concepts, we can formally introduce some important definitions from formal language theory.

Definition 3. An ***alphabet*** is "a finite, non-empty set of objects called ***symbols***" [43, p. 13]. We use Σ to represent an alphabet. For example, $\Sigma = \{0, 1\}$ represents an alphabet containing the symbols 0 and 1 (this particular alphabet is called the ***binary alphabet***).

Definition 4. A ***string over an alphabet*** is "a finite sequence of symbols taken from that alphabet" [43, p. 14]. The term ***character*** is often used interchangeably with symbol in the context of strings. However, for our purposes, we add a slight nuance to its definition: A ***character*** is a symbol that has a specific position within a string.

$$\text{Symbols in string } \mathtt{little} : \mathtt{l}, \mathtt{i}, \mathtt{t}, \mathtt{e}$$
$$\text{Characters in string } \mathtt{little} : \mathtt{l}, \mathtt{i}, \mathtt{t}, \mathtt{t}, \mathtt{l}, \mathtt{e}$$

For convenience, the characters of a string are written one after another, rather than being separated by commas.

$$\mathtt{cat} = (\mathtt{c}, \mathtt{a}, \mathtt{t})$$

The ***length*** of a string is "the number of characters it contains" [43, p. 14]. We use $|s|$ to denote the length of a string s. The string of length zero is called the ***empty string*** and is denoted by λ.

With these definitions out of the way, we can now define the pivotal concept of subsequences.

Definition 5. A ***subsequence*** of some sequence S is "a sequence that can be obtained by deleting elements of S" while preserving the order of the remaining elements [33, p. 366]. For example, *meats* is a subsequence of the string *mathematics*, but *cheat* is not because the ordering of the characters has not been maintained. It is important to note that the characters in a subsequence do not need to be contiguous in the original string. Also, the subsequence does not need to have fewer characters than the original string, meaning that any string is a subsequence of itself.

We indicate that a string s is a subsequence of a string w by writing $s \sqsubseteq w$. Additionally, for brevity, we write $s \sqsubseteq s_1, s_2, \ldots, s_k$ to indicate that s is a subsequence of each of the strings $s_1, s_2, \ldots, s_k$.

Definition 6. A ***substring*** of some string w is "a string of consecutive characters in w" [35, p. 19]. For example, *put* is a substring of the string *computer*, but *mute* is not because not all of the characters are contiguous. A substring is also sometimes called a ***slice*** due to how it captures the idea of extracting a portion of a string.

We define $s[i, j)$ as the substring of s beginning with the ith character (inclusive) and ending with the jth character (exclusive). If $i = j$, then $s[i, j) = \lambda$. Additionally, we use $s[k]$ as a shorthand for $s[k, k+1)$, i.e., the kth character of s.

$$\mathtt{understandable}[5, 10) = \mathtt{stand}$$

Finally, we have reached the point where we can define the focus of this section, the longest common subsequence (LCS):

Definition 7. "Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a sequence of both X and Y. In the ***longest common sequence problem...*** the goal is to find a maximum-length common subsequence of X

and Y" [15, p. 394]. This problem is naturally extendable to the maximum-length common subsequence of n sequences, which we refer to as the n-longest-common-subsequence problem. In both cases, we abbreviate longest common subsequence as LCS.

Note that a longest common subsequence of a set of strings is not necessarily unique. For instance, *aa*, *bb*, and *ab* are all a longest common subsequence of the strings *baba* and *aabb*. We denote the set of all possible LCS as $\mathrm{LCS}(s_1, s_2, \ldots, s_n)$ and the length of these strings (which is one value since they are all equal length) as $\mathcal{L}(s_1, s_2, \ldots, s_k)$.

2.2 Calculating Longest Common Subsequences

One question that arises often in computer science is: how can we find the LCS, given input strings x, y? We begin by outlining a naive approach. Take two strings, x with length m and y with length n, where $n \leq m$. Then, iterate through every possible subsequence of y. For each subsequence s, if $s \sqsubseteq x$, then s is a common subsequence. If s is at least as long as the currently saved strings, then save it and remove all shorter subsequences currently saved. At the end of the program, the saved strings will be an LCS of x and y.

Definition 8. "Given functions $f, g : \mathbb{R} \to \mathbb{R}$ with g nonnegative, we say

$$f = \mathcal{O}(g)$$

if and only if there exist a constant $c \geq 0$ and an x_0 such that for all $x \geq x_0$, $|f(x)| \leq cg(x)$" [15, p. 530].

For example, the runtime of taking the average of a set of numbers is $\mathcal{O}(n)$, where n is the amount of numbers in the set. The singular step of dividing the total sum by the amount of numbers is not represented in the runtime, as with large enough n the singular step's impact on the runtime is negligible. In this notation, coefficients are typically omitted, ie. we would write $\mathcal{O}(x^2)$ instead of $\mathcal{O}(2x^2)$.

Definition 9. ***Dynamic programming*** is a programming technique that recursively splits problems into sub-problems, storing the results of each sub-problem as it is calculated to avoid having to calculate the result multiple times [15, p. 362]. Dynamic programming problems are typically optimization problems, where the goal is to maximize or minimize some value [15].

Definition 10. An $a_1 \times a_2 \times \ldots a_n$ ***array*** is an n-dimensional grid of elements. Along each axis $i \in \{1, \ldots, n\}$, there are a_i ***indices***, and each element in the array is associated with one index per axis. As a result, each element in an array is unique identified by their n indices.

We use computer science convention, so indices are ***zero-indexed***, meaning that the index for axis i is a value in $\{0, \ldots, a_i - 1\}$. Notably, this means that a_i is not a valid index.

Definition 11. "A ***vector*** is a one-dimensional array of numbers" [15, p. 1215]. We write vectors as a comma-separated list of symbols, enclosed by square brackets to differentiate them from sequences, e.g., $[a, b, c]$.

Definition 12. "A ***matrix*** is a rectangular array of numbers" [15, p. 1214], and is typically written as a grid of whitespace-separated symbols enclosed by a single pair of square brackets, e.g.,

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

While the naive algorithm for finding the LCS proposed earlier works, it is highly inefficient. It is easy to see that the runtime for iterating through each possible subsequence will be $\mathcal{O}(2^n)$, where n is the length of the binary string. Checking if the subsequence is also in X will take approximately $\mathcal{O}(n)$. Combined, the asymptotic runtime becomes $\mathcal{O}(n \cdot 2^n)$. Instead, a ***dynamic programming*** technique can be applied to find the LCS much more efficiently. In fact, the resultant algorithm for finding an LCS is so elegant that it is often used by textbooks as an example to explain dynamic programming (see e.g., *Introduction to Algorithms* [15]).

For calculating the length of the LCS, we observe that the problem is an optimization problem where our goal is to maximize the length of our chosen common subsequence. We apply dynamic programming by recursively removing characters from the ends of one of our strings. For each pair of partial strings, we calculate their LCS length, store the subsequence of that length, and slowly add back more and more of the original strings until we have found the maximum common subsequence length for our original strings.

Algorithm 1 defines a simple dynamic programming algorithm for finding the length of the LCS for two strings, x and y. It utilizes a two-dimensional array to store the results of partial input strings, and references those stored results rather than recalculating everything for every iteration. The algorithm outputs an $m \times n$ matrix T, where $T[i][j]$ corresponds the LCS of the first i characters of x and the first j characters of y. Thus, we can get the LCS of x and y by accessing the table at $T[m][n]$.

For example, with $x =$ `apple`, $y =$ `ape`, $T[3][2]$ is 2, while $T[5][3]$, which we expect to be the LCS length of the original strings, is 3, as expected.

We now prove that this algorithm correctly obtains the LCS length for two strings. Proof. Let x and y be input strings to our function. We define a recursive equation for $T[i][j]$ as the length of the longest common subsequence of $x[0, i)$ and $y[0, j)$ in such a way that $T[i][j]$ is only dependent on previously calculated values. Trivially, by storing all values of T when they are calculated, we will only need to calculate each value once.

$$T[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ T[i-1][j-1] + 1 & \text{if } x[i-1] = y[j-1] \\ \max\bigl(T[i][j-1], T[i-1][j]\bigr) & \text{otherwise} \end{cases}$$

This equation for T is well-known, and can be derived in a straight forward manner using properties defined later, specifically Property 3 and Property 4. ■

Note that Algorithm 1 can easily be expanded to find the k-LCS length by using a $|s_1| \times |s_2| \times \cdots \times |s_k|$ array and updating the code accordingly [28].

Algorithm 1: Longest Common Subsequence Table [7]

Input: x: string, y: string
Output: T: longest common subsequence table

$m \leftarrow |x|$
$n \leftarrow |y|$
$T \leftarrow (m+1)$ by $(n+1)$ matrix of zeros
for $i \leftarrow 1$ **to** $m+1$
  for $j \leftarrow 1$ **to** $n+1$
    if $x[i-1] = y[j-1]$
      $T[i][j] = T[i-1][j-1] + 1$
    else
      $T[i][j] = \max\big(T[i][j-1], T[i-1][j]\big)$
return T

We now consider the runtime of Algorithm 1. Since the algorithm applies dynamic programming and saves previous results, each element in T only performs a single calculation. Thus, the runtime is $\mathcal{O}(nm)$.

This runtime is exponentially better then the naive approach, which had a runtime of $\mathcal{O}(n \cdot 2^m)$. Further improvements are minimal or rely on additional assumptions, as it is conjectured that sub-quadratic runtimes are near-impossible [1].

While there is minimal room to improve algorithms for directly finding the LCS length, there is also a class of algorithms dedicated to finding the approximate LCS length. For instance, Hajiaghayi et al. take advantage of several properties of the LCS to obtain an approximately $\mathcal{O}\big(\sqrt{n}\big)$ runtime [22]. For alphabets with only two characters, Xiaoyu He and Ray Li defined an approximately linear $\big(\mathcal{O}(n)\big)$ runtime [24].

2.3 Applications

2.3.1 Computational Biology

A sequence's edit distance is one of the most foundational aspects of computational biology, with the longest common subsequence being one of the most common ways of defining the edit distance. While the LCS has, over time, been replaced with edit distance metrics better aligned with underlying biological meaning, it continues to have strong connections to the field of computational biology.

Sequence Alignment

Given multiple sequences of biological data, a common goal is to align the sequences, such that as many characters as possible are located at the same index within the sequence. Doing this makes it much easier to analyze the sequences. In order to align characters, spaces

are inserted into the sequence to shift characters. It is known that finding the maximal alignment is equivalent to finding the LCS of the sequences [19]. The traditional LCS problem is equivalent to global alignment, which aligns the entire sequence. However, it is more common for computational biologists to make use of local sequence alignment, which finds alignments for substrings instead of the entire sequence (e.g. BLAST [8]).

Phylogenetic Construction and Analysis

Phylogenetic construction is the process of creating an ancestry tree for species. In creating a phylogenetic tree, the goal is to place new species on the tree such that they are close to other species that diverged from a recent common ancestor, and far away from species who diverged from a common ancestor a long time ago. Here, the challenge is to create a metric for how related two species are to each other. One way of doing this is to take the LCS of the two species' DNA. If the species are closely related, their DNA should be similar. If the species are unrelated, random mutations over time result in their DNA being effectively unrelated. Thus, if the DNA's LCS is significantly longer than expected, it can be inferred that the two species are more closely related [41]. Of course, since DNA sequences are so long, the expected value of the LCS length is almost exactly the (un-normalized) Chvátal-Sankoff constant. (Additional LCS-based criterion for sequence homology can be found in [25]).

2.3.2 Computational Linguistics

In computational linguistics, the Longest Common Subsequence, as an edit distance metric, proves to be particularly useful in natural language processing, where it is used to compare words, sentences, and documents. Specific applications include spelling correction, file comparison, plagiarism detection, and sequence pattern matching.

Spelling Correction

One significant application of the LCS lies in spelling correction. A real-time spell-check system within a word processor or a search engine works by identifying and rectifying misspelled words. A dictionary of recognized words is stored. When a user inputs a misspelled word, the system generates a list of suggestions by comparing the misspelled word with the words in the dictionary. The LCS algorithm is instrumental in this process. If the LCS metric edit distance between the misspelled word and a word in the dictionary is within a certain threshold, the word is added to the pool of suggestions. The top suggestions, based on the probability of the word being the intended word, are then presented to the user. [32, 34, 50]

Data Comparison and Revision Control

When comparing files, the LCS method is a valuable tool for identifying differences between two files. Version control systems like Git, collaborative writing tools like Google Docs, and file comparison tools like Unix's `diff` make considerable use of this technique.

These tools employ algorithms which compare the previous and current versions of a file by identifying the longest common subsequence between them, sometimes using entire words or lines (in the case of code) as symbols. The differences between the two versions are typically highlighted using this information, making it possible to trace file alterations over time with efficiency. With these discrepancies brought to light, a streamlined collaboration and document management processes is made possible. [26]

Plagiarism Detection

The LCS algorithm can also be used in plagiarism detection. By comparing a document with a database of existing documents, the algorithm can identify the longest common subsequence using the file comparison technique previously described. Typically, the length of the LCS is weighted against the total length of the document to calculate a similarity score. This helps to limit the occurrence false positives and negatives. If the similarity score is above a certain threshold, the document is flagged for further review. This capability is instrumental in academia, publishing, and content creation industries, where maintaining originality and integrity is paramount. [4]

Sequence Pattern Matching

Furthermore, the LCS algorithm proves invaluable in sequence pattern matching tasks. For example, when searching for a specific pattern within a text, the algorithm can be used to identify the longest common subsequence between the pattern and the text. This information can then be used to determine the presence of the pattern within the text. This, in conjunction with matching algorithms that determine the position of the pattern within the text, is used in search engines, text editors, and data processing tools. [23]

2.3.3 Other Fields

As Chvátal and Sankoff note, one motivation for studying the longest common subsequences of random strings is in calibrating interpretation of results for LCS computations on strings whose values are *not* random [14]. If a sequence is returned whose length is significantly above or below the expectation, it can be inferred that the strings correspond (or do not correspond) in some meaningful way. This idea is the basis for most applications which make explicit use of the Chvátal-Sankoff constants: the LCS of (usually two) strings is used as a (basis for a) similarity metric between those strings, and the Chvátal-Sankoff constant is used as a prior to understand what similarity score represents a significant deviation from the average case. Among other uses, this notion can be directly applied to perform anomaly detection.

Anomaly Detection

Given multiple sequences of data, we are often interested in detecting if one of the sequences diverges greatly from the other sequences at some point. To determine if a sequence

has diverged "enough" to be significant, there are many different similarity metrics that can be used to assign a numerical value to the divergence. There are several similarity metrics which use the LCS. Of note is the normalized LCS ($nLCS$) similarity metric, defined as (for two strings of length n and m) as:

$$nLCS = \frac{\mathcal{L}(x,y)}{\sqrt{xy}}$$

The expected value of this metric is equal to a variant of the Chvátal-Sankoff constant defined for strings of unequal length, indicating that the Chvátal-Sankoff constants serve as a reasonable threshold for marking sequences as significant outliers. [10]

Data Compression

Biological data can grow to be very large. As such, efficient data compression is a necessity. Fortunately, biological data tends to have substrings that repeat often, such as the substrings representing amino acids within DNA, or on a larger scale, entire chromosomes. Unfortunately, random mutations often result in a large number of small differences between substrings that would otherwise be identical.

One way of compressing biological data while accounting for these random mutations is to instead look for common subsequences. If two subsequences are similar enough, then we can encode the substrings containing those subsequences with the same common value, and then use a much smaller amount of additional data to encode the differences from the common value. Of course, to maximize the efficiency of these encoded substrings, we want the common subsequences to be as long as possible - the LCS of the substrings. Using the LCS in this way is a key part of the GRS algorithm and other similar algorithms. [6, 48]

Of course, nothing here strictly requires that the data be biological. These algorithms can also be applied to other fields. In fact, the GRS algorithm is based on Unix's $diff$ command, so the algorithm originated outside of computational biology to begin with.

Stratigraphy

Stratigraphy is the study of geological strata, or the layers of rock that form over time. One point of interest in stratigraphy is to analyze the amount that two different locations' strata are related. One method of doing this is to treat the strata obtained at each location as a sequence, where the symbols are the specific lithographic units (types of rock) that make up each strata. Then, the two locations are more correlated with each other if the LCS of their strata is longer. [44]

Chapter 3

Improving Bounds on the Chvátal-Sankoff Constants

In this chapter, we begin by formally describing the Chvátal-Sankoff constants and the basic mathematical concepts required to understand them. Then, we give a brief history of the previous work in the literature on calculating the constants. Next, we describe an important algorithm by Kiwi and Soto [30] for calculating lower bounds on the general Chvátal-Sankoff constants and explain how it simplifies in the binary two-string case. We then detail our modifications to the algorithm which allowed us to compute new best lower bounds, with a focus on the binary algorithm. Finally, we report the new world-record lower bounds we obtained through these computations.

3.1 The Chvátal-Sankoff Constants

In order to describe the Chvátal-Sankoff constants, we first begin by introducing the probabilistic definitions that underlie the constants. We focus mainly on random events with finite or countably infinite number of outcomes, as this type of event is most relevant to the Chvátal-Sankoff constants. (For d strings of length l using an alphabet of size σ, there are exactly $\sigma^{d\ell}$ unique outcomes for a random string, a finite number). We first give a formal treatment of the definitions, and then an informal summary afterward.

Necessary Definitions

Formally, we must understand the following four definitions:

Definition 13. "A ***probability*** space has three components:

- A ***sample space*** Ω, which is the set of all possible outcomes of the random process modeled by the probability space.
- A family of sets $\mathcal{F}$ representing the allowable events, where each set in $\mathcal{F}$ is a subset of the sample space Ω; and

- a probability function $\Pr : \mathcal{F} \to \mathbb{R}$ satisfying Definition 14" [38, p. 3].

Definition 14. "A ***probability function*** is any function $\Pr : \mathcal{F} \to \mathbb{R}$ that satisfies the following conditions:

> **Condition 1.** For any event E, $0 \leq \Pr(E) \leq 1$
>
> **Condition 2.** $\Pr(\Omega) = 1$
>
> **Condition 3.** For any finite or countably infinite sequence of pairwise mutually disjoint events $E_1, E_2, E_3, \ldots$,
>
> $$\Pr\left(\bigcup_{i \geq 1} E_i\right) = \sum_{i \geq 1} \Pr(E_i)\text{" [38, p. 3].}$$

Definition 15. "A ***random variable*** X on a sample space Ω is a real-valued function on Ω: that is, $X : \Omega \to \mathbb{R}$. A ***discrete random variable*** is a random variable that takes on only a finite or countably infinite number of values.

The event '$X = a$' includes all the basic outcomes of the sample space in which the random variable X assumes the value a. That is '$X = a$' represents the set $\{s \in \Omega | X(s) = a\}$. We denote the probability of that event by

$$\Pr(X = a) = \sum_{s \in \Omega \mid X(s) = a} \Pr(s)\text{" [38, pp. 20–21].}$$

Definition 16. "The ***expectation*** of a discrete random variable X, denoted by $\mathbb{E}[X]$, is given by $\mathbb{E}[X] = \sum_i i \Pr(X = i)$ where the summation is over all values in the range of X. The expectation is finite if $\sum_i i \Pr(X = i)$ converges; otherwise, the expectation is unbounded" [38, p. 21].

Informally, to generally understand the Chvátal-Sankoff constants, one need only grasp the concept of expectation. The expectation of some variable essentially denotes the average value of that variable across all of its possible values (weighted by their likelihood). It is like asking the question, "On average, what should I expect this value to be?"

The Constants

In 1975, Chvátal and Sankoff [14] began investigating the expected length of a longest common subsequence of two uniformly random strings of length ℓ over an alphabet of size σ. They observed that the function that returns the expected length of the LCS of those strings, divided by ℓ, is *superadditive* with respect to ℓ.

Definition 17. A function f is ***superadditive*** if

$$f(x + y) \geq f(x) + f(y)$$

for all x, y in the domain of f [14].

In other words, if we randomly select two pairs of strings of length $\ell_1 + \ell_2$, on average, the normalized length of their LCS will be at least as large as the sum of the normalized LCS length of a random pair of strings length ℓ_1 and the normalized LCS length of a random pair of strings length ℓ_2. This superadditive property has also been shown to hold for arbitrary numbers of (more than just two) strings.

This property is very useful. Since the normalized expected length of the LCS is bounded above by 1 (the length of the LCS cannot exceed the lengths of the strings), this means that as the lengths of d random strings grow to infinity, the expected length of their LCS, divided by the strings' length, converges to a constant (by Fekete's Theorem) [14]. These constants are precisely what are known as the Chvátal-Sankoff constants, with one constant for each pair of d and σ values. Formally,

Definition 18. The ***Chvátal-Sankoff constant*** for $\gamma_{\sigma,d}$ is defined as

$$\gamma_{\sigma,d} = \lim_{\ell \to \infty} \frac{\mathbb{E}(X_{\sigma,d,\ell})}{\ell}$$

where $X_{\sigma,d,\ell}$ is a discrete random variable for the length of the longest common subsequence of d strings of length ℓ with each character independently and uniformly selected from an alphabet with σ symbols.

When γ is left without subscripts, it indicates $\gamma_{2,2}$, the constant for pairs of strings containing only two possible symbols (e.g., *0* and *1*). This is sometimes referred to as simply 'the Chvátal-Sankoff constant'.

The Chvátal-Sankoff constants serve as proportionality constants, crucial for understanding the behavior of LCS under random conditions. However, while we have upper and lower bounds on them, their exact values are **not** known. Determining them is a well-known open problem in computer science–appearing in several textbooks–and there have been extensive efforts to both tighten these bounds and estimate the true values of the constants [46, 49].

3.2 History

In this section, we give a brief history of the efforts that have gone into bounding and approximating the Chvátal-Sankoff constants, highlighting the most important works. We focus pimarily on the constant that deals only with pairs of binary strings, as it is by far the most studied in the literature.

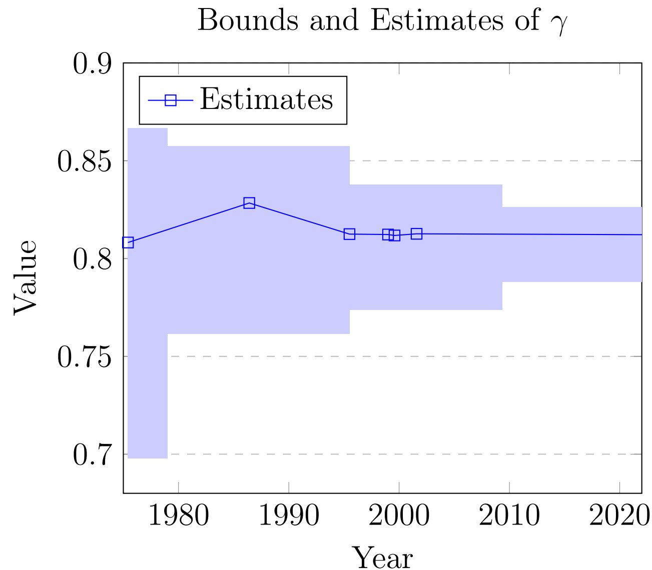
As a result of the superadditive property discussed in Section 3.1, one possible approach to estimate a lower bound for the Chvátal-Sankoff constant is to compute the LCS for all possible pairs of strings of length n, and then calculate the mean length of those subsequences. However, while effective for small values of n, this method requires $\mathcal{O}(2^n \cdot 2^n)$ LCS calculations, and in practice the mean grows very slowly relative to the number of calculations required. To achieve strong lower bounds, more sophisticated methodologies are necessary.

In 1975, Chvátal and Sankoff, whom the constants were later named after, were the first people to suggest the existence of a constant. They established in their original paper

Researchers	Year	Bounds	Estimate
Chvátal and Sankoff [14]	1975	$0.697844 \leq \gamma \leq 0.866595$	$\gamma \approx 0.8082$
Deken [20]	1979	$0.7615 \leq \gamma \leq 0.8575$	
Steele [45]	1986		$\gamma \stackrel{?}{=} \frac{2}{1+\sqrt{2}} \approx 0.8284$
Dančík and Paterson [17, 18, 16]	1995	$0.77391 \leq \gamma \leq 0.83763$	$\gamma \approx 0.81225 \pm 0.00025$
Boutet de Monvel [9]	1999		$\gamma \approx 0.812282$
Baeza-Yates et al. [5]	1999		$\gamma \approx 0.8118$
Bundschuh [12]	2001		$\gamma \approx 0.812653$
Lueker [36]	2009	$0.788071 \leq \gamma \leq 0.82628$	
Bukh and Cox [11]	2019		$\gamma \approx 0.8122$

Table 3.1: History of prominent bounds and estimates of the binary Chvátal-Sankoff constant

Figure 3.1: The best upper and lower bounds on γ and estimates of γ over time.



that the Chvátal-Sankoff constant for the binary case (i.e., $\gamma_{2,2}$) falls between 0.697844 and 0.866595. Through two sets of Monte-Carlo simulations, a technique built upon extensive and repeated random sampling, they approximated the value to be around 0.8082. Deken [20] initiated efforts to refine these bounds in 1979, culminating in 1983 with a tighter interval of 0.7615 to 0.8575, achieved via diverse matching algorithms. Despite their differences, these algorithms adhere to a common procedural framework; the algorithms initialize a pointer on each sequence, alternate movements until encountering an increase in the count of differing elements between the pointers, match characters when possible, and resume the process from the matched points, disregarding any prior characters.

In 1986, Steele [45] postulated that the exact value of the binary Chvátal-Sankoff constant equals $\frac{2}{1+\sqrt{2}}$. This conjecture stemmed from a heuristic approach inspired by coin-flip sequences, assuming an independent and identical distribution of each coin flip. Much of this

conjecture's development was spurred by the contributions of Arratia and Waterman [2, 3].

Dančík and Paterson [18] introduced a novel method in 1995 utilizing deterministic finite automata to scan symbols from both strings and generate a common subsequence, albeit not necessarily the longest. They use a Markov chain simulation, which is a simulation that performs graph traversal where edges represent memoryless transition probabilities, to efficiently generate LCS probabilities. Their findings hinted that the Chvátal-Sankoff constant likely approximates 0.81225 ± 0.00025.

Boutet de Monvel [9], in early 1999, conducted extensive Monte-Carlo simulations akin to Chvátal and Sankoff, albeit optimized, parallelized, and with increased sample size. His prediction for the binary constant stood at 0.812282. Later that year, Baeza-Yates, Gavaldà, Navarro, and Scheihing [5] estimated the constant at 0.8118, leveraging finite state automata and analysis of string generation complexity (via Kolmogorov complexity analysis). In 2001, Bundschuh employed a lattice framework to tackle the longest common subsequence problem, resulting in an approximated value of 0.812653.

In 2009, Lueker [36] optimized and adapted Dančík and Paterson's algorithm to refine the bounds to 0.788071 to 0.82628. This disproved Steele's conjectured value for the binary constant, as Lueker's upper bound fell below Steele's prediction. That same year, Kiwi and Soto [30] used a generalized version of Lueker's lower-bound algorithm to compute lower bounds for a wider range of alphabets and numbers of strings. These were the state-of-the-art results before our project.

In 2019, Bukh and Cox [11] defined an alternate Markov chain based on a model of frog dynamics. They first define $\Delta(2n) = \mathcal{L}\big(st, uv\big) = \mathcal{L}\big(s, u\big) - \mathcal{L}\big(t, v\big)$ where s, t, u, v are independent, random strings of length n. They propose a conjecture asserting the existence of constants c_1 and c_2 such that $E[\Delta(n)]$ is asymptotically equivalent to $c_1\sqrt[3]{n}$, and the square root of the variance of $\Delta(n)$ is asymptotically equivalent to $c_2\sqrt[3]{n}$. Analysis of their computational data provided to support their conjecture indicated values close to $c_1 \approx \frac{1}{2}$ and $c_2 \approx \frac{1}{4}$. Using their conjecture, they estimate the value of the binary Chvátal-Sankoff constant as 0.8122.

Also of note is work by Kiwi, Loebl, and Matoušek [31] in 2005 which proves, via subadditivity arguments, that $\lim_{\sigma\to\infty} \gamma_{\sigma,2}\sqrt{\sigma} = 2$. Additionally, in 2022, Tiskin [47] demonstrated that by linking $\gamma_{2,2}$ to the parameters of a specific stochastic (i.e., random) particle process, the constant is the solution to an explicit (but intractably large) system of polynomial equations with integer coefficients.

3.3 The Kiwi-Soto Algorithm

Here we introduce the algorithm of Kiwi and Soto [30] for calculating lower bounds on the Chvátal-Sankoff constants, which we used, with several modifications, to obtain our results. This algorithm is based on the overall process described by Lueker [36], but generalized for arbitrary values of σ and d. The algorithm takes as inputs the size of the alphabet (σ), the number of strings (d), the string length used in calculating a lower bound (ℓ), and the number of iterations to attempt to converge to a lower bound (n). For a matrix $\mathbf{v}$, we use $\mathbf{v_i}$ to

represent the ith row of $\mathbf{v}$. Throughout this paper, we adopt the convention that all matrices and vectors are represented in bold. A numerical value in bold is a vector where every element is that value. Additionally, all vectors contain real numbers and are of length $\sigma^{d\ell}$, corresponding to the number of possible *d-tuple* of strings where each string has ℓ characters taken from an alphabet of size σ.

Definition 19. "A [finite] sequence with k elements is called a ***k-tuple***" [43, p. 6]. Unlike a regular sequence, we allow the elements of a k-tuple to belong to different sets.

Algorithm 2: The Feasible Triplet Algorithm

Function `FeasibleTriplet`(σ, d, ℓ, n):

  $\mathbf{v} \leftarrow \mathbf{0}$ `//` $\mathbf{0}$ `is a` $d \times \sigma^{d\ell}$ `matrix of zeros`

  $(\mathbf{u}, r, \epsilon) \leftarrow (\mathbf{v_0}, 0, 0)$

  for $i = d, \ldots, n$

    $\mathbf{w} \leftarrow F(\mathbf{v_d}, \mathbf{v_{d-1}}, \ldots \mathbf{v_1})$

    $R \leftarrow \max_{0 \leq j < \sigma^{d\ell}} (\mathbf{w} - \mathbf{v_d})[j]$ `// where` $[j]$ `denotes the` j`th element`

    $\mathbf{W} \leftarrow \mathbf{w} + dR\mathbf{1} - F(\mathbf{w} + (d-1)R\mathbf{1}, \mathbf{w} + (d-2)R\mathbf{1}, \ldots, \mathbf{w})$

    $E \leftarrow \max\{0, \max_{0 \leq j < \sigma^{d\ell}} \mathbf{W}[j]\}$

    if $R - E \geq r - \epsilon$

      $(\mathbf{u}, r, \epsilon) \leftarrow (\mathbf{w}, R, E)$

    Shift the rows of $\mathbf{v}$ up by 1

    $\mathbf{v_d} \leftarrow \mathbf{w}$

  return $(\mathbf{u}, r, \epsilon)$

Motivating this algorithm, Kiwi and Soto proved the following pivotal lemma:

Lemma 1. *Suppose a function* $F : (\mathbb{R}^{\sigma^{d\ell}})^d \mapsto \mathbb{R}^{\sigma^{d\ell}}$ *satisfies the following three properties:*

1. **Monotonicity.** *Given two d-tuples of vectors* $(\mathbf{v_1}, \ldots \mathbf{v_d}) \leq (\mathbf{u_1}, \ldots, \mathbf{u_d})$ *component-wise, then* $F(\mathbf{v_1}, \ldots \mathbf{v_d}) \leq F(\mathbf{u_1}, \ldots, \mathbf{u_d})$ *component-wise.*

2. **Translation Invariance.** $\forall\, r \in \mathbb{R}, F(\mathbf{v_1} + r\mathbf{1}, \ldots, \mathbf{v_d} + r\mathbf{1}) = F(\mathbf{v_1}, \ldots, \mathbf{v_d}) + r\mathbf{1}$.

3. **Feasibility.** *There exists a* **feasible triplet** *for F. That is, there exists a 3-tuple* $(\mathbf{u}, r, \epsilon) \in \mathbb{R}^{\sigma^{d\ell}} \times \mathbb{R} \times [0, r]$ *such that* $F(\mathbf{u} + (d-1)r\mathbf{1}, \ldots, \mathbf{u} + r\mathbf{1}, \mathbf{u}) \geq \mathbf{u} + (dr - \epsilon)\mathbf{1}$ *component-wise.*

Then, for any feasible triplet of F, we have that $\gamma_{\sigma,d} \geq d(r - \epsilon)$.

Thus, if a feasible triplet is found for σ and d, it can be used to calculate a lower bound on $\gamma_{\sigma,d}$ [30]. The overall goal of the Feasible Triple algorithm (Algorithm 2) is to calculate a *good* feasible triplet, that is, a feasible triplet where $(r - \epsilon)$ is as large as possible, for a specific F with these properties.

In Algorithm 2, we use the function F, which can be any function meeting the criteria of Lemma 1, but we use the choice of F given by Kiwi and Soto (see (3.6) on page 526 of [30]). Furthermore, we define an ordering

$$\phi : \{1, ..., \sigma^{d\ell}\} \longleftrightarrow \{(s_1, \ldots, s_d) \mid s_1, \ldots, s_d \text{ strings of length } \ell\}$$

to help with indexing vectors in F. This ordering assigns a unique integer to each d-tuple of strings such that each number assigned is a valid index in our vectors (which are all of length $\sigma^{d\ell}$). We use this ordering for all vectors in the algorithm.

We assume that F and all sub-functions have access to parameters σ, d, and ℓ. As defined by Kiwi and Soto (see (3.6) on page 526 of [30]):

$$F(\mathbf{v_1}, \ldots, \mathbf{v_d}) = \mathbf{b} + \max\{F_z(\mathbf{v_1}, \ldots, \mathbf{v_d}) : z \in \Sigma\}.$$

Here, $\mathbf{b}$ is $\mathbf{1}$ if every string in the tuple at index i starts with the same character, and $\mathbf{0}$ otherwise. Next, we will describe how to compute F_z (Algorithm 3). For this purpose, we introduce two new definitions.

Definition 20. For a string a, we define the first character of a to be the ***head*** of a, denoted as $h(a)$. We define the rest of a to be the ***tail*** of a, denoted $T(a)$. Note that $a = h(a)T(a)$.

Given d vectors and a character $z \in \Sigma$ as input, F_z outputs a new vector. The calculation for the output of F_z at index i is independent of the calculations for any other index. Thus, to simplify our description of F_z, we let the index i be given, use our ordering to obtain strings $(s_1, \ldots, s_d)$, and calculate the value of the output vector at i using them.

Here, N is an ordered set containing the positions of strings in the tuple $(s_1, \ldots, s_d)$ that **do not** start with some character z. The `variate` function iterates through all combinations of $\{c_1, \ldots, c_{|N|}\}$, where each c can be any character in Σ. For each combination, `variate` sums the value of $\mathbf{v_{|N|}}$ at the index corresponding to removing the head of each string that doesn't start with z, and appends c_i to each string, i.e., for a string s_i not starting with z, it is replaced with $T(s_i)c_i$ when indexing $\mathbf{v_{|N|}}$. Finally, F_z divides by the number of possible variations to obtain the mean value of $\mathbf{v}$ across all combinations. In essence, `variate` takes all strings that do not start with z, removes their first character, and adds some new character to the end of each string, doing this for all possible combinations of new characters at the end of the strings. F_z then calculates the mean value of $\mathbf{v_{|N|}}$ at the indices corresponding to all $\sigma^{|N|}$ of the d-tuples created by `variate`.

The Binary Case

The algorithm and related definitions can be greatly simplified (see Algorithm 4) when working only with pairs of strings with an alphabet of size two (that is, $d = 2, \sigma = 2$).

Here, we describe how F simplifies in the binary case. As in the general case, we define an ordering that assigns an integer index to every possible pair of strings. We use the indexing $\mathbf{v}[a, b]$ to represent indexing $\mathbf{v}$ at the index corresponding to the pair of strings (a, b).

$$F(\mathbf{v_1}, \mathbf{v_2})[a, b] = \mathbf{s} + \max(F_0(\mathbf{v_1}, \mathbf{v_2}), F_1(\mathbf{v_1}, \mathbf{v_2})).$$

Algorithm 3: Calculating F_z

```
Function F_z(v_1, ..., v_d):
    N ← (j | j ∈ (1, ..., d), h(s_j) ≠ z)
    return σ^{-|N|} · variate(v_{|N|}, N)
Function variate(v_i, N):
    len ← |N|
    r ← 0
    for c_1 ∈ Σ
        s_{N[1]} ← T(s_{N[1]})c_1          // where N[1] is the first element of N
        for c_2 ∈ Σ
            s_{N[2]} ← T(s_{N[2]})c_2
            ⋱
                for c_len ∈ Σ
                    s_{N[len]} ← T(s_{N[len]})c_len
                    r ← r + v_i[s_1, ..., s_d]
                    Restore all s to their original values
    return r
```

Here, $\mathbf{s}$ is $\mathbf{1}$ if a and b start with the same character, and $\mathbf{0}$ otherwise, and F_0 and F_1 are sub-functions that simplify F_z to the binary case.

We now define F_1 and F_0. Note that F_0 has the same definition as F_1, but swaps *0* and *1* when evaluating $h(A)$ and $h(B)$:

$$F_1(\mathbf{v_1}, \mathbf{v_2})[a, b] = \left\{ \begin{array}{lll} 0, & h(a) = \mathit{1} & h(b) = \mathit{1}\,, \\ \frac{1}{2}(\mathbf{v_1}[a, T(b)0] + \mathbf{v_1}[a, T(b)1]) & h(a) = \mathit{1} & h(b) = \mathit{0}\,, \\ \frac{1}{2}(\mathbf{v_1}[T(a)0, b] + \mathbf{v_1}[T(a)1, b]) & h(a) = \mathit{0} & h(b) = \mathit{1}\,, \\ \frac{1}{4}\left(\sum_{c_1, c_2 \in \{0,1\}} \mathbf{v_2}[T(a)c_1, T(b)c_2]\right) & h(a) = \mathit{0} & h(b) = \mathit{0}\,. \end{array} \right\} \tag{3.1}$$

$$F_0(\mathbf{v_1}, \mathbf{v_2})[a, b] = \left\{ \begin{array}{lll} 0, & h(a) = \mathit{0} & h(b) = \mathit{0}\,, \\ \frac{1}{2}(\mathbf{v_1}[a, T(b)0] + \mathbf{v_1}[a, T(b)1]) & h(a) = \mathit{0} & h(b) = \mathit{1}\,, \\ \frac{1}{2}(\mathbf{v_1}[T(a)0, b] + \mathbf{v_1}[T(a)1, b]) & h(a) = \mathit{1} & h(b) = \mathit{0}\,, \\ \frac{1}{4}\left(\sum_{c_1, c_2 \in \{0,1\}} \mathbf{v_2}[T(a)c_1, T(b)c_2]\right) & h(a) = \mathit{1} & h(b) = \mathit{1}\,. \end{array} \right\} \tag{3.2}$$

Essentially, when one or more of the strings in an index of F_1 starts with *0*, F_1 returns the average of all permutations of those strings where we remove the first character and append an arbitrary new character. When all strings start with *1*, the average is 0, since there are no strings to permute. F_0 does the same but for strings that start with a *1*.

Algorithm 4: The Binary Feasible Triplet Algorithm

```
Function BinaryFeasibleTriplet(ℓ, n):
    v0 ← 0                          // 0 is a vector containing 2^{2ℓ} zeroes
    v1 ← 0
    (u, r, ε) ← (v0, 0, 0)
    for i = 2, ..., n
        v2 ← F(v1, v0)
        R ← max_{0≤j<2^{2ℓ}} (v2 − v1)[j]
        W ← v2 + 2R1 − F(v2 + R1, v2)
        E ← max{0, max_{0≤j<2^{2ℓ}} W[j]}
        if R − E ≥ r − ε
            (u, r, ε) ← (v2, R, E)
        v0 ← v1
        v1 ← v2
    return (u, r, ε)
```

3.4 Algorithm Modifications

Since a value must be computed for every possible d-tuple of strings over an alphabet of size σ, the Feasible Triplet Algorithm (Algorithm 2) has a time complexity of $\Omega(\sigma^{d\ell})$, and since it must store d vectors of size $\sigma^{d\ell}$, it has a space complexity of $\mathcal{O}(d\sigma^{d\ell})$. As the values of σ, d, and ℓ increase, this exponential scaling quickly becomes a barrier to progress. In this section, we describe our approach to overcome some of these limitations. We focus primarily on optimizing the Binary Feasible Triplet Algorithm: of the following techniques, only parallelization was implemented for both versions of the Feasible Triplet Algorithm.

3.4.1 Parallelization

As noted previously, each value of the new vector $\mathbf{v_d}$ is computed independently of every other value in $\mathbf{v_d}$. Accordingly, the algorithm is amenable to parallelization. To implement this, a chosen number of threads is spun up, with each thread calculating the values of $\mathbf{v_d}$ for a distinct slice of the vector.

3.4.2 Indexing

A pair of binary strings can be represented as a single 64-bit unsigned integer so long as their combined length does not exceed 64 characters and a consistent indexing scheme is identified. Given two strings a and b, we chose to index them by interleaving their bits starting with a and filling the remaining bits with zeros on the left. For instance, a pair of strings $a = 1011$ and $b = 0010$ is represented as 0...10001110. We say a 64-bit integer is an

interleaved string pair, or simply a ***pair***, if it represents two binary strings interleaved in this fashion.

This scheme has several desirable properties. Firstly, calculating $\mathbf{v_2}$ can be done (in sequential order) by simply iterating through all integers from 0 to $2^{2\ell} - 1$, which we denote $[0, ..., 2^{2\ell})$. Secondly, we need not check whether the first bit of a and b match: we know ahead of time that pairs $[0, ..., 2^{2\ell-2})$ have the same first bit (0) for a and b, pairs $[2^{2\ell-2}, ..., 2^{2\ell-1} + 2^{2\ell-2})$ have different first bits, and pairs $[2^{2\ell-1} + 2^{2\ell-2}, ..., 2^{2\ell})$ have the same first bit (1). Lastly, this indexing will allow us to determine a method of reading disk memory sequentially once arrays become too large to fit in RAM (Section 3.4.4).

3.4.3 Array Reductions and Symmetries

In the binary case, Lueker notes how it is not necessary to store arrays $\mathbf{v_i}$, for three consecutive integers i, in memory, as the recurrence can be simplified by storing only two arrays (that is, $\mathbf{v_1}, \mathbf{v_2}$) and iterating on those instead, with the bound calculation adjusted appropriately [36]. We also adopt this optimization. As Lueker also notes, one can further observe that since complementing both a and b does not impact the length of their longest common subsequence, we get that

$$\mathbf{v_i}[(\overline{a}, \overline{b})] = \mathbf{v_i}[(a, b)],$$

where $\overline{a}$ represents the binary complement of a [36]. With the indexing scheme defined in Section 3.4.2, for any two indices i and j such that

$$2^{2\ell-1} > i \geq 0 \text{ and } j = 2^{2\ell} - 1 - i,$$

we have that

$$2^{2\ell} > j \geq 2^{2\ell-1} \text{ and } (a_i, b_i) = (\overline{a_j}, \overline{b_j}). \tag{3.3}$$

Thus, we need only iterate strings pairs represented by the integers $[0, ..., 2^{2\ell-1})$.

3.4.4 Sequential Memory Access

As ℓ grows, memory becomes a limiting factor far faster than computation time. For instance, at $\ell = 20$ with 4 bytes for each value stored in the vector, a naive implementation requires

$$2^{2 \cdot 20} \cdot 4 = 4\ 398\ 046\ 511\ 104 \text{ bytes} \approx 4.4 \text{ TB}$$

per vector. Even if symmetry is fully exploited, this requires roughly a terabyte of memory per vector. With our resources, it is infeasible to store all of these values in RAM.

Accordingly, vectors must be read from and written to external (disk) memory. However, operating exclusively from external memory is also infeasible: we experienced over a 100x slow down when reading and writing values only from disk. File I/O time dwarfed the time taken to actually perform the computations for the recurrence, largely because memory accesses were non-sequential.

To alleviate this overhead, reading from and writing to disk must be done in large sequential blocks. As such, a recursive approach is taken to identify contiguous blocks of integers from $[0, ..., 2^{2\ell-1})$ whose accessed values also span a contiguous block of integers. This section outlines the approach taken and justifies why it is correct.

The function as implemented contains three primary loops: $L_{0,0}$, $L_{0,1}$, and $L_{1,0}$. Each loop is valid only for a predetermined range of string pairs, but accepts a start parameter and end parameter to allow for subdivision of that range.

$L_{0,0}$

$L_{0,0}$ iterates over string pairs whose first bits match and calculates the new value in the recurrence according to the last case of F_1 (see Equation 3.1). For a particular a, b string pair, with the interleaved indexing, this is accomplished by the following simple procedure:

Input: x, an interleaved a, b string pair
Output: The new recurrence value for index x

function `SameFirstBit(`x`):`
  Bitshift x left by 2
  return $1 + \frac{1}{4}(\mathbf{v_1}[x] + \mathbf{v_1}[x+1] + \mathbf{v_1}[x+2] + \mathbf{v_1}[x+3])$

Since $L_{0,0}$ iterates only within $[0, ..., 2^{2\ell-2})$, the first bit of a and b will always be 0. Thus, this procedure always accesses the values at $4x$, $4x+1$, $4x+2$, and $4x+3$. For $x \geq 2^{2\ell-3}$, it will access pairs $\geq 2^{2\ell-1}$, so we use (3.3) to transform the pairs back to their symmetric position within the vector. As a result, if x is iterated in sequential order, $L_{0,0}$ first accesses values sequentially within $[0, ..., 2^{2\ell-1})$ and writes out values sequentially within $[0, ..., 2^{2\ell-3})$, and then accesses values sequentially within $(2^{2\ell-1}, ..., 0]$ and writes out values sequentially within $(2^{2\ell-2}, ..., 2^{2\ell-3}]$.

$L_{0,1}$ and its Recursion

$L_{0,1}$ iterates over string pairs whose first bits do not match. It calculates the second case $(h(a) = 0, h(b) = 1)$ from F_0 (see Equation 3.2). This can be accomplished by another simple procedure:

Input: x, an interleaved a, b string pair
Output: The new recurrence value for index x

function `DifferentFirstBit(`x`):`
  $a \leftarrow$ even bits of x
  $b \leftarrow$ odd bits of x
  Zero out first bit of b
  Bitshift b left by 2
  $i \leftarrow a \mid b$ `// Bitwise OR to recombine` a `and` b
  return $\frac{1}{2}(\mathbf{v_1}[i] + \mathbf{v_1}[i+1])$

Since the string pairs are iterated only within $[2^{2\ell-2}, ..., 2^{2\ell-1})$, the first bit of a is always

0 and the first bit of b is always 1. As such, the second bit of b uniquely determines whether it is necessary to access values from $[0, ..., 2^{2\ell-2})$ or $[2^{2\ell-2}, ..., 2^{2\ell-1})$ to calculate the new value for a particular string pair. Similarly, the second bit of a uniquely determines whether it is necessary to access values from the first half or second half of the range determined by the second bit of b, for a total of four possible access ranges of size $2^{2\ell-3}$. For example, by looking at only the second bit of a and b for string pair 0...011010 ($\ell = 3$), we know we need only access values from $[2^{2\ell-3}, ..., 2^{2\ell-2})$. In fact, we know that every string pair in the range $[2^{2\ell-2} + 2^{2\ell-3}, ..., 2^{2\ell-2} + 2^{2\ell-3} + 2^{2\ell-4})$ is constrained to accessing values from the range $[2^{2\ell-3}, ..., 2^{2\ell-2})$. Additionally, since each string pair accesses two distinct values, it is guaranteed that every value in one of these ranges is accessed exactly once.

Within each of these four ranges, we can again subdivide into four blocks of equal size based on the next bit of a and of b, again with the same guarantees on access. This subdivision can be performed recursively until the vector has been divided into chunks small enough to fit into RAM (based on a chosen `stop_depth`). At this point, each chunk can be read *sequentially* into RAM, computation performed for the chunk's corresponding values, and results written sequentially out to disk. This recursion is demonstrated as follows:

```
function Recurse(offset, index_offset, depth):
    num_strs ← 2^(2(ℓ−depth)−2)
    if depth < stop_depth
        Recurse(offset, index_offset, depth + 1)
        Recurse(offset + num_strs/4, index_offset + 2 × num_strs/4, depth + 1)
        Recurse(offset + 2 × num_strs/4, index_offset + num_strs/4, depth + 1)
        Recurse(offset + 3 × num_strs/4, index_offset + 3 × num_strs/4,
          depth + 1)
    else
        load in v1[index_offset] through v1[index_offset + 2 × num_strs]
        start ← 2^(2ℓ−2) + offset
        end ← start + num_strs
        L_{0,1}(start, end)
        write out values v2[start] through v2[end]
```

$L_{1,0}$ and its Recursion

$L_{1,0}$ calculates the third case ($h(a) = 0, h(b) = 1$) from F_1 (see Equation 3.1). A similar recursion as for $L_{0,1}$ can be defined for $L_{1,0}$. However, when the second bit of a is 1, the string pair will access values above $2^{2\ell-1} - 1$. In this case, we use (3.3) to transform the value accesses to their symmetric position within the vector.

Essentially, these recursions serve as wrappers for the algorithm, dictating the portions of $\mathbf{v_2}$ to calculate and the values from $\mathbf{v_1}$ that must be loaded in to facilitate those calculations. By dividing the calculations into specific contiguous chunks, the recursions guarantee that

disk memory is read from sequentially, with no unnecessary values read in, while maintaining the property that disk memory is written to in sequential chunks.

3.5 Results

In our experiments we ran Algorithm 2 for various values of σ, d, and ℓ, and Algorithm 4 for increasingly large values of ℓ. The results from our binary-case computations compared to the results obtained by Lueker are shown in Table 3.2. When reported in the tables, lower bounds are rounded down to their 6th decimal place. To our knowledge, for $\ell \geq 16$, these bounds **exceed any previously reported lower bounds** on γ. We also believe that for all but $\sigma = 14, d = 2$, the results in Table 3.3 **exceed all previously reported lower bounds** for $\gamma_{\sigma,d}$.

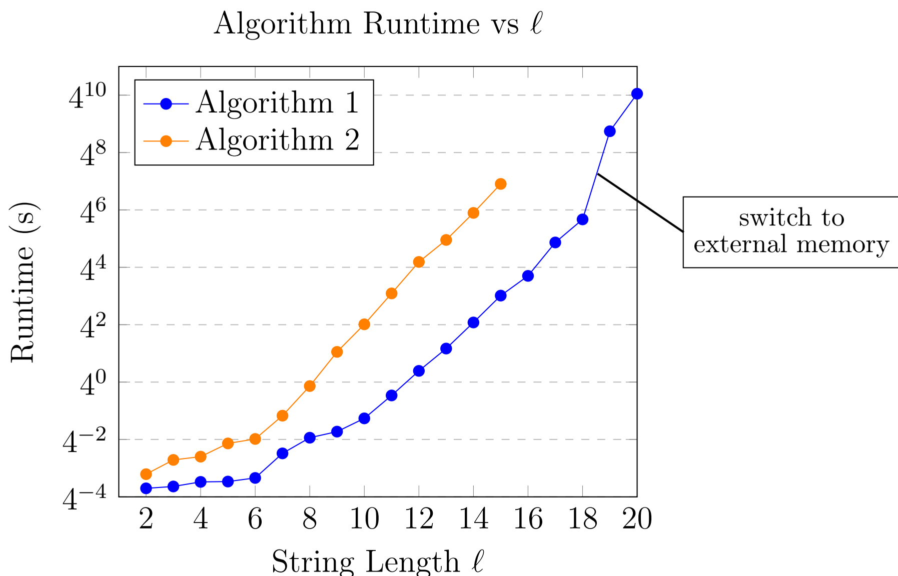


Figure 3.2: Runtime of the Feasible Triplet and Binary Feasible Triplet algorithms for $\gamma_{2,2}$.

ℓ	**Lower bound on γ (Lueker)**	**Lower bound on γ (ours)**
1	0.666666	0.666666
2	0.727272	0.727272
3	0.747922	0.747922
4	0.758576	0.758576
5	0.765446	0.765446
6	0.770273	0.770273
7	0.773975	0.773975
8	0.776860	0.776860
9	0.779259	0.779259
10	0.781281	0.781281
11	0.783005	0.783005
12	0.784515	0.784515
13	0.785841	0.785841
14	0.787017	0.787017
15	0.788071	0.788071
16	-	**0.789021**
17	-	**0.789882**
18	-	**0.790668**
19	-	**0.791389**
20	-	**0.792052**

Table 3.2: Lower bounds on γ for values of ℓ, the string length parameter, from 1 to 20. Bold numbers represent new best lower bounds.

Alphabet size $\sigma = 2$			
Lower bound on $\gamma_{2,d}$			
d	Previous best	Our results	ℓ
2	0.788071	**0.792052**	20
3	0.704473	**0.711548**	10
4	0.661274	**0.664722**	6
5	0.636022	**0.639248**	5
6	0.617761	**0.621057**	4
7	0.602493	**0.607261**	3
8	0.594016	**0.598782**	3
9	0.587900	**0.592177**	3
10	0.570155	**0.582348**	2
11	0.570155	**0.578463**	2
12	0.563566	**0.574268**	2
13	0.563566	**0.571067**	2
14	0.558494	0.558494	1
15	-	**0.558494**	1

Alphabet size $\sigma = 3$			
Lower bound on $\gamma_{3,d}$			
d	Previous best	Our results	ℓ
2	0.671697	**0.682218**	9
3	0.556649	**0.564841**	5
4	0.498525	**0.509237**	4
5	0.461402	**0.474304**	3
6	0.421436	**0.445434**	2
7	0.413611	**0.434513**	2
8	0.405539	**0.425774**	2
9	-	**0.400949**	1

Alphabet size $\sigma = 4$			
Lower bound on $\gamma_{4,d}$			
d	Previous best	Our results	ℓ
2	0.599248	**0.614333**	8
3	0.457311	**0.472979**	4
4	0.389008	**0.405702**	3
5	0.335517	**0.365329**	2
6	0.324014	**0.349848**	2
7	-	**0.317032**	1

Alphabet size $\sigma = 5$			
Lower bound on $\gamma_{5,d}$			
d	Previous best	Our results	ℓ
2	0.539129	**0.549817**	5
3	0.356717	**0.394945**	3
4	0.289398	**0.324337**	2
5	0.273884	**0.302235**	2
6	-	**0.263369**	1

Alphabet size $\sigma = 6$			
Lower bound on $\gamma_{6,d}$			
d	Previous best	Our results	ℓ
2	0.479452	**0.499229**	4
3	0.309424	**0.347798**	3
4	0.245283	**0.277835**	2
5	-	**0.231234**	1

Alphabet size $\sigma = 7$			
Lower bound on $\gamma_{7,d}$			
d	Previous best	Our results	ℓ
2	0.44502	**0.466481**	4
3	0.234567	**0.273275**	2
4	0.212786	**0.242798**	2
5	-	**0.200004**	1

Alphabet size $\sigma = 8$			
Lower bound on $\gamma_{8,d}$			
s	Previous best	Our results	ℓ
2	0.42237	**0.438799**	4
3	0.207547	**0.244709**	2
4	-	**0.187869**	1

Alphabet size $\sigma = 9$			
Lower bound on $\gamma_{9,d}$			
d	Previous best	Our results	ℓ
2	0.40321	**0.414876**	4
3	0.186104	**0.221554**	2
4	-	**0.168164**	1

Alphabet size $\sigma = 10$			
Lower bound on $\gamma_{10,d}$			
d	Previous best	Our results	ℓ
2	0.38656	**0.393811**	4
3	0.168674	**0.202401**	2
4	-	**0.152193**	1

Table 3.3: Lower bounds for $\gamma_{\sigma,d}$ with the value of the string length parameter ℓ we set to achieve our bounds. Bold numbers represent new best lower bounds.

Chapter 4

LCS Properties and BLISS Playground

4.1 Foundations and Rationale

When discussing the properties of longest common subsequences, we consider two main categories: edit and distribution properties. An edit operation is a mapping that transforms one string into another. The four most common edit operations are insertions, deletions, substitutions, and permutations [39]. Distribution properties, on the other hand, are properties that describe how the set of longest common subsequences is distributed across all possible pairs of strings.

For the purposes of this chapter, we will focus on the LCS of two strings. Namely, we will consider the strings x and y, of lengths n and m respectively, over the arbitrary alphabet Σ. Before we outline the properties of longest common subsequences, we first provide some definitions and notation, as well as the motivation behind the use of these properties.

Definition 21. Let w be a subsequence of string x. We say that a character c in w is ***critical*** if removing c from x would make w no longer a subsequence of x.
For example, consider the string *apple* and subsequence *apl*. *l* is critical, since *apl* is not a subsequence of *appe*. *p* is not critical, since *apl* is still a subsequence of *aple*.

Definition 22. An ***affix*** is a character or string that is added to another string to construct a new string. If the affix is added to the beginning of the string, it is called a ***prefix***. If the affix is added to the end of the string, it is called a ***suffix***. If the affix is added at any other position in the string, it is called an ***infix***. Additionally, if part of the affix is added to the beginning of the string and the remainder is added to the end of the string, it is called a ***circumfix***.

Definition 23. A ***permutation*** is a bijective mapping that rearranges the elements of a collection. We will consider two types of permutations:

1. A ***character permutation*** is a permutation that rearranges the characters in a string. We use the notation $\phi(x)$ to denote the string x with its characters permuted according to ϕ.

2. A ***symbol permutation*** is a permutation that rearranges the symbols in an alphabet. In the case where Σ is a binary alphabet, we call the non-trivial symbol permutation ***complementation***. We use the notation $\varphi(x)$ to denote the string x with its symbols permuted according to φ.

Definition 24. Given a string x, we define the ***reverse*** of x, denoted as x^{R}, to be the string formed by reversing the order of the characters in x. For example, the reverse of the string $x = \texttt{abc}$ is $x^{\mathsf{R}} = \texttt{cba}$.

Definition 25. We use the notation $x \setminus x[k]$ to denote the string x with the character at index k removed.

Motivation for Edit Properties Many of the applications of the LCS problem described in Section 2.3 involve the LCS of two strings that have strings edits applied to them. Edit properties describe the relationship between these edited strings and the original strings.

Motivation for Distribution Properties Since the Chvátal-Sankoff constants depend on the expected value of the LCS of random strings, they are inherently tied to the distribution of LCS lengths. Developing stronger distribution properties might help uncover methods for more efficiently calculating the constants.

4.2 Properties of Longest Common Subsequences

We provide an outline of the properties of longest common subsequences here, along with a brief summary of their meaning and significance. The proofs of each property can be found in Chapter B. However, these properties can be difficult to digest on their own. We recommend utilizing BLISS Playground, discussed in the follow section, to better understand them.

4.2.1 General Properties

Property 1. The Length Bounds Property:

For all $x \in \Sigma^n$ and $y \in \Sigma^m$,

$$0 \leq \mathcal{L}\big(x, y\big) \leq \min(n, m)$$

Summary: The length of the longest common subsequence of two strings is always between 0 and the length of the shorter string. This property is a direct consequence of the definition of the longest common subsequence.

Property 2. The Commutative Property:

For all $x \in \Sigma^n$ and $y \in \Sigma^m$,

$$\text{LCS}\big(x, y\big) = \text{LCS}\big(y, x\big).$$

Summary: The longest common subsequence of two strings is the same regardless of the order of the strings. This property is a direct consequence of the definition of the longest common subsequence.

Property 3. The Concatenation Property (Same String)

For all $x \in \Sigma^n$, $y \in \Sigma^m$, and $H, T \in \Sigma^\star$,

$$\text{LCS}\big(HxT, HyT\big) = \Big\{HzT \mid z \in \text{LCS}\big(x, y\big)\Big\}$$

$$\implies \mathcal{L}\big(HxT, HyT\big) = |H| + \mathcal{L}\big(x, y\big) + |T|$$

Summary: The longest common subsequence of two strings prefixed (suffixed) by the same strings is the longest common subsequence of the original strings prefixed (suffixed) by the same strings. This implies that the length of the longest common subsequence of two strings prefixed (suffixed) by the same strings is the sum of the lengths of the prefix (suffix) and the longest common subsequence of the original strings.

Property 4. The Concatenation Property for Distinct Strings

For all $x \in \Sigma^n$, $y \in \Sigma^m$, and $A, B \in \Sigma^\star$ such that $\text{LCS}\big(A, B\big) = \{\lambda\}$,

$$\text{LCS}\big(Ax, By\big) \subseteq \text{LCS}\big(Ax, y\big) \cup \text{LCS}\big(x, By\big)$$

$$\text{LCS}\big(xA, yB\big) \subseteq \text{LCS}\big(xA, y\big) \cup \text{LCS}\big(x, yB\big)$$

Summary: Given two strings and an associated prefix (suffix) for each, the longest common subsequence of the strings with both their prefixes (suffixes) is also a longest common subsequence for the strings when only one has a prefix (suffix).

4.2.2 Edit Operation Properties

Property 5. The Character Insertion Property

For all $x \in \Sigma^n$, $y \in \Sigma^m$, and $C \in \Sigma$, let z be the string formed by inserting C into y at index k. If there exist some $u \sqsubseteq y[0,k)$ and $v \sqsubseteq y[k\rangle$ such that $uCv \sqsubseteq x, z$ and $uv \in \text{LCS}(x,y)$, then

$$\mathcal{L}(x,z) = \mathcal{L}(x,y) + 1$$

If this is not the case, then

$$\mathcal{L}(x,z) = \mathcal{L}(x,y)$$

Summary: When a character is inserted into one string, the length of the longest common subsequence increases by one if the character is critical to the new longest common subsequence. Otherwise, the length remains the same.

Property 6. The Character Deletion Property

For all $x \in \Sigma^n$ and $y \in \Sigma^m$, let z be the string formed by deleting the character at index k from y. If there exists some $w \in \text{LCS}(x,y)$ such that $w \sqsubseteq y \setminus y[k]$, then

$$\mathcal{L}(x,z) = \mathcal{L}(x,y)$$

If this is not the case, then

$$\mathcal{L}(x,z) = \mathcal{L}(x,y) - 1$$

Summary: When a character is removed from one string, the length of the longest common subsequence stays the same if the character was not critical to some previous longest common subsequence. Otherwise, the length decreases by one.

Property 7. The Character Substitution Property

For all $x \in \Sigma^n$, $y \in \Sigma^m$, and $C \in \Sigma$, let z be the string formed by substituting the character at index k in y with C. Consider the following conditions for some $w \in \text{LCS}(x,y)$:

1. $w \sqsubseteq y \setminus y[k]$

2. $uCv \sqsubseteq x, z$ where $u \sqsubseteq y[0,k)$, $v \sqsubseteq y[k\rangle$, and $w = uv$

If both of the conditions are satisfied, then

$$\mathcal{L}\big(x,z\big) = \mathcal{L}\big(x,y\big) + 1$$

If only one of the conditions are satisfied, then

$$\mathcal{L}\big(x,z\big) = \mathcal{L}\big(x,y\big)$$

If neither condition is satisfied, then

$$\mathcal{L}\big(x,z\big) = \mathcal{L}\big(x,y\big) - 1$$

Summary: When a character is substituted in one string, the effect on the length of the longest common subsequence depends on the net result of removing the original character and inserting the new character.

Property 8. The Character Permutation Property

For all $x \in \Sigma^n$ and $y \in \Sigma^m$, let ϕ be a permutation of the characters in y. If z is the string formed by applying ϕ to y, then

$$\mathcal{L}\big(x,z\big) = \mathcal{L}\big(x,y\big) + \delta$$

where δ is an integer in the range $[-k, k]$

Summary: When the characters in one string are permuted, the effect on the length of the longest common subsequence depends on the net result of substituting each original character with its corresponding character in the permutation.

Property 9. The Symbol Permutation Property

For all $x \in \Sigma^n$ and $y \in \Sigma^m$, let φ be a permutation of the symbols in Σ.

$$\mathrm{LCS}\big(\varphi(x), \varphi(y)\big) = \Big\{\varphi(z) \mid z \in \mathrm{LCS}\big(x,y\big)\Big\}$$
$$\implies \mathcal{L}\big(x,y\big) = \mathcal{L}\big(\varphi(x), \varphi(y)\big)$$

Summary: When the symbols in an alphabet are permuted, such that the symbols in each string are permuted in the same way, the length of the longest common subsequence remains the same.

Property 10. The Reversal Property

For all $x \in \Sigma^n$ and $y \in \Sigma^m$,

$$\mathrm{LCS}\big(x^{\mathsf{R}}, y^{\mathsf{R}}\big) = \Big\{ z^{\mathsf{R}} \mid z \in \mathrm{LCS}\big(x, y\big) \Big\}$$

$$\implies \mathcal{L}\big(x, y\big) = \mathcal{L}\big(x^{\mathsf{R}}, y^{\mathsf{R}}\big)$$

Summary: When each string has the order of its characters reversed, the length of the longest common subsequence remains the same.

Property 11. The Slice-Concatenation Property

For all $HxT \in \Sigma^n$, $HyT \in \Sigma^m$, $P \in \Sigma^{|H|}$, and $S \in \Sigma^{|T|}$,

$$\mathcal{L}\big(HxT, HyT\big) = \mathcal{L}\big(PxS, PyS\big)$$

Summary: When the common prefix (suffix) of each string is replaced by a different prefix (suffix) of the same length, the length of the longest common subsequence remains the same.

Property 12. The Removed Character Property

For all $x \in \Sigma^n$, $y \in \Sigma^m$, and $n, m \geq k$, if $\mathcal{L}\big(x \setminus x[k], y \setminus y[k]\big) = \mathcal{L}\big(x, y\big) - 2$, then $x[k] \neq y[k]$.

Summary: If the length of the longest common subsequence decreases by two when the kth character is removed from both strings, then the kth characters of the strings must be different.

4.2.3 Distribution Properties

Property 13. The LCS Count Property:

For all $x \in \Sigma^n$ and $y \in \Sigma^m$,

$$1 \leq \Big|\text{LCS}\big(x, y\big)\Big| \leq \min(n, m).$$

Summary: The number of longest common subsequences of two strings is always between 1 and the length of the shorter string.

Property 14. The Substring Property:

For every $z \in \bigcup_{k=0}^{\min(n,m)} \Sigma^k$, there exist $x \in \Sigma^n$ and $y \in \Sigma^m$ such that $z \in \text{LCS}\big(x, y\big)$.

Summary: Every possible string of length $k < \min(n, m)$ over an alphabet is a longest common subsequence of some pair of strings of lengths n and m over that alphabet.

Property 15. The $n - 1$ Distribution Property:

Across all possible d-tuples of strings of length n and alphabet size $\sigma \geq 2$, every possible string of length $n - 1$ over that alphabet each appears as an LCS exactly $(\sigma n - n + 1)^d - (\sigma n - n + 1)$ times in total.

Note that for pairs of binary strings, this simplifies neatly to $n(n + 1)$.

Summary: Given any string s of length $n - 1$ from an alphabet of size $\sigma \geq 2$, if you tallied the number of times it appears as an LCS for every possible combination of d strings of length n from that same alphabet, that tally would exactly equal $(\sigma n - n + 1)^d - (\sigma n - n + 1)$.

4.3 BLISS Playground

To supplement the theoretical and practical aspects of the properties of longest common subsequences, we have developed a suite of interactive tools to help visualize and understand

the various properties and algorithms associated with the LCS problem. The Binary Longest InterString Subsequence (BLISS) Playground[1] is designed to be user-friendly and intuitive, allowing users to interact with the underlying algorithms and data structures in real-time. The tools are web-based, making them accessible to a wide audience. In this chapter, we provide an overview of the interactive tools and their features, as well as instructions on how to use them.

Figure 4.1: The BLISS Playground homepage

[1]The BLISS tool suite can be found at https://statistics-of-subsequences.github.io/

4.3.1 Edit Properties Explorer

The Edit Properties Explorer visualizes the properties of Section 4.2.2 and helps users further explore the ideas presented in the Matrix Builder tool. The tool presents a matrix containing all pairs of strings of lengths n and m, respectively. The values of n and m can be set by the user, with a maximum value of 5 due to the exponential growth of the data. This matrix can be visualized in two distinct modes: elevation and orthographic mode.

In elevation mode, the matrix cells are extruded, offering users a three-dimensional representation where they can observe the length of the longest common subsequence as a three dimensional plot. Users can interactively manipulate the matrix through panning, zooming, and rotating. Orthographic mode presents the matrix in a two-dimensional format, allowing users to more easily view large portions of the matrix at once.

In both modes, users have the ability to input two binary strings of length n and m, with the corresponding cell highlighted on the matrix. Additionally, the tool displays the LCS length and the set of possible LCS for the given pair of strings. Upon selecting a cell, users can apply various edit operations, including substitution, permutation, slicing and concatenation, complementation, and reversal, to observe how these changes affect the longest common subsequence length. Notably, insertion and deletion operations are excluded since they would alter the string lengths, thus altering the shape of the matrix. Overall, the Edit Properties Explorer aims to expound upon the properties of the longest common subsequence concerning different edit operations, allowing users to investigate the consequences of the properties and of string edits in general in more detail.

Figure 4.2: The user is presented with a 3×4 matrix in the Edit Properties Explorer tool. The matrix is displayed in both elevation and orthographic mode. We can see the length of the LCS for each pair of strings by comparing the colors of the cells to the key.

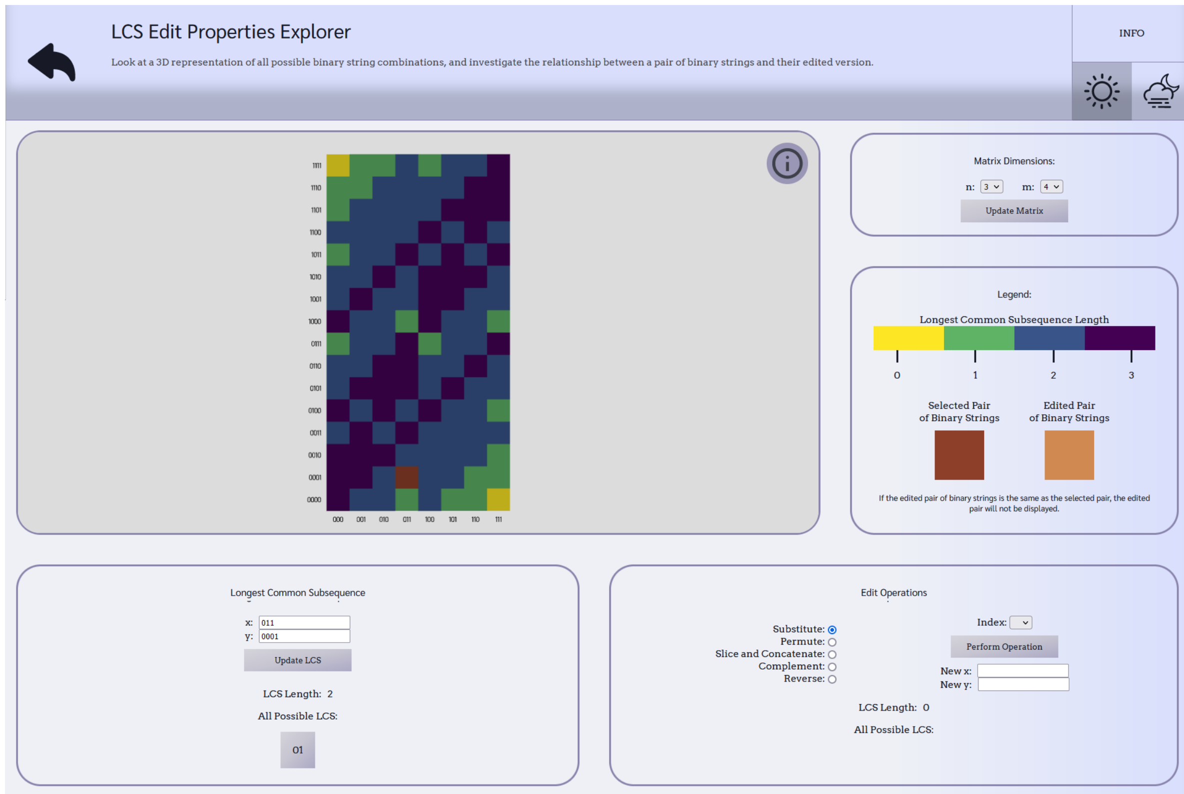


Figure 4.3: The user inputs the strings `011` and `0001` into the Edit Properties Explorer tool. The cell at the intersection of these strings is highlighted in red. Below where we entered the strings, we can see that the length of the LCS is 2, and the set of LCSs is {`01`}.

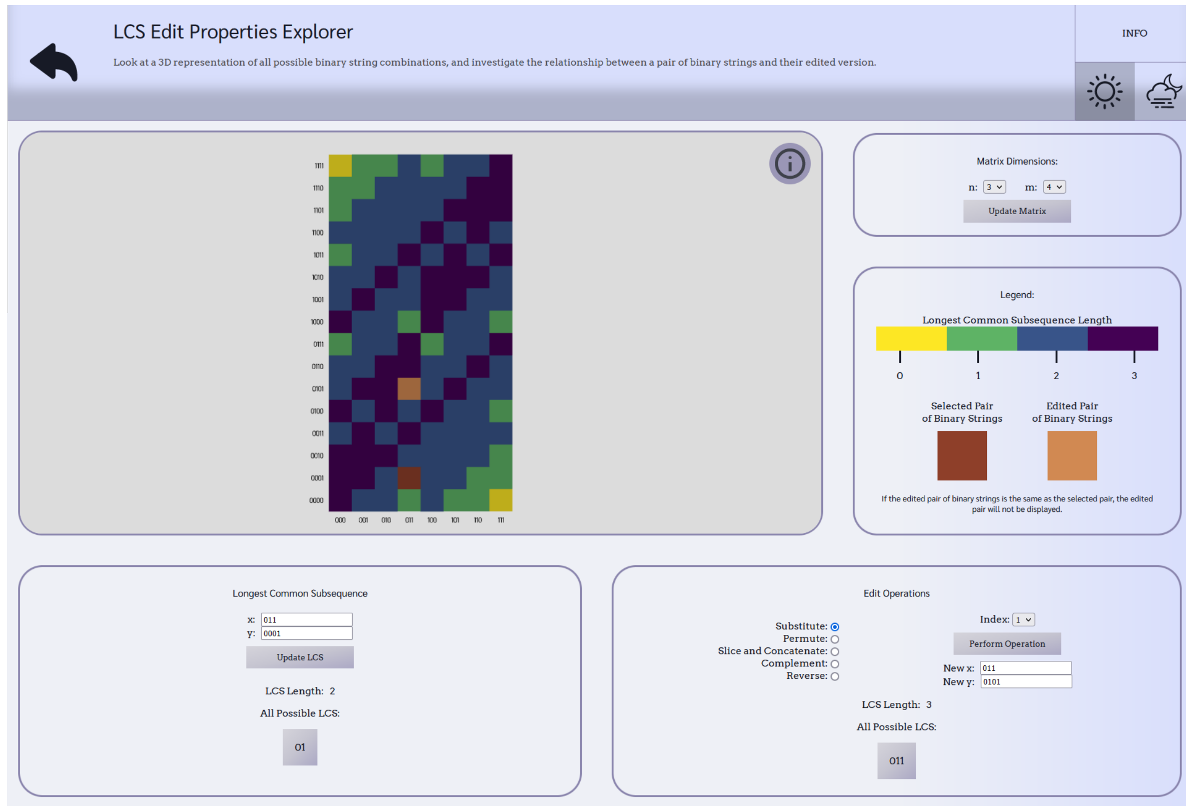


Figure 4.4: The user has applied the substitution operation to the cell at index 1. The new cell is highlighted in orange, and the length of the LCS has increased to 3, and the set of LCSs has been changed to {*011*}.

4.3.2 Matrix Builder

The Matrix Builder tool is used to visualize the relationship between LCS length of different pairs of strings, of length n and m respectively. Similar to the Edit Properties Explorer, the maximum values for n and m are capped at 5 due to the exponential growth of the data. The matrix is structured with all binary strings of length n as columns and all binary strings of length m as rows, initially empty. There are two modes of interaction: exploratory and puzzle.

Exploratory Mode

In general interaction mode, users can select individual cells within the matrix to display the LCS length for that specific pair of strings, denoted by a distinctive color. This interactive selection process allows users to explore and identify LCS lengths for various string pairs. Once users have selected the desired cells, they have the option to fill in the matrix. The user may additionally choose to toggle which properties will be used to fill in the matrix. The tool automatically populates the matrix, but fills in only the cells that can be derived from

the user's original selection (taking into account which properties are selected in the toggle). This feature provides users with a comprehensive visualization of how edit properties can be used to reduce the amount of work needed to calculate LCS lengths.

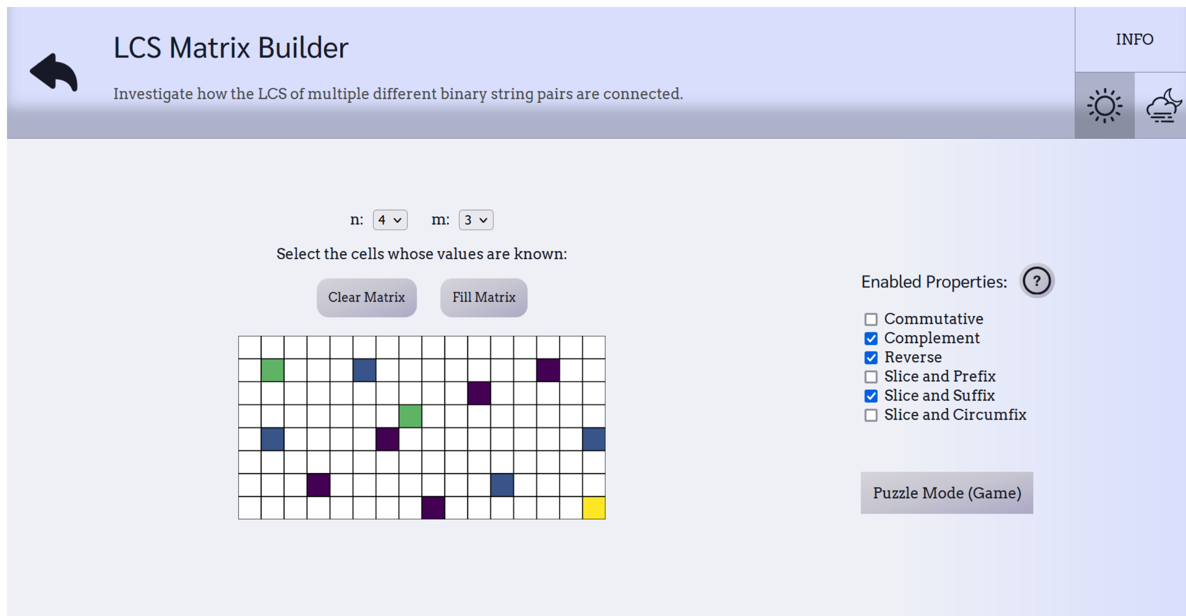


Figure 4.5: The user constructs a 4×3 matrix in the Matrix Builder tool. Several cells have been selected, and we can see the LCS length for each pair of strings. Additionally, the user has toggled the "Complement", "Reverse", and "Slice and Suffix" properties.

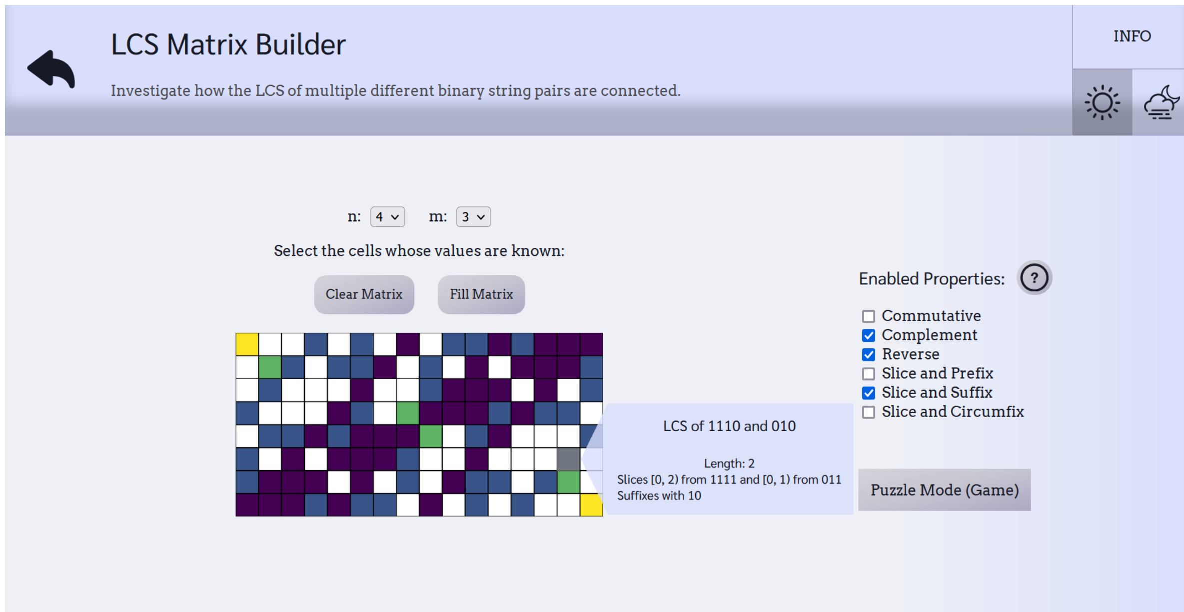


Figure 4.6: The matrix has become populated based on the enabled properties and previously selected cells. If we hover over the cell at the intersection of the strings `1110` and `010`, we can see that the LCS length is 2. We also see that it was derived from suffixing the substrings `1111`[0,2) and `011`[0,1) with `10`

Puzzle Mode

In puzzle mode, users are presented with an empty matrix, a goal matrix, and a set of edit properties. The user must fill in the empty matrix to match the goal matrix using the provided edit properties. This mode is designed to challenge users to think critically about how to apply edit properties to achieve a specific LCS length. The puzzle mode is an engaging way for users to test their understanding of the properties of the LCS and the impact of different edit operations on the LCS length. The game encourages the player to find the minimum number of cells to fill in order to reach the goal matrix, promoting efficient problem-solving strategies.

Figure 4.7: The user is presented with the level select for the Puzzle Mode of the Matrix Builder tool. There are a total of thirty levels, six for each of the five difficulties.

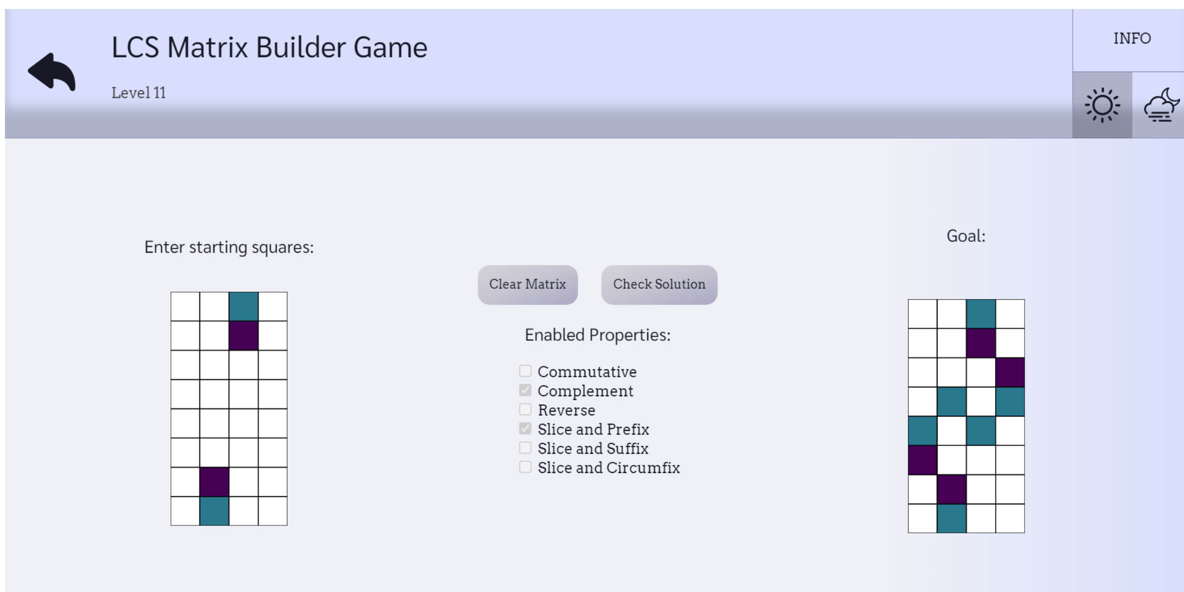


Figure 4.8: The user has selected Level 11. The goal matrix shows what the matrix (to the left) should look like in order to complete the level. Additionally, the enabled properties for this level are "Complement" and "Slice and Prefix". The user has started to fill in the matrix to solve the puzzle.

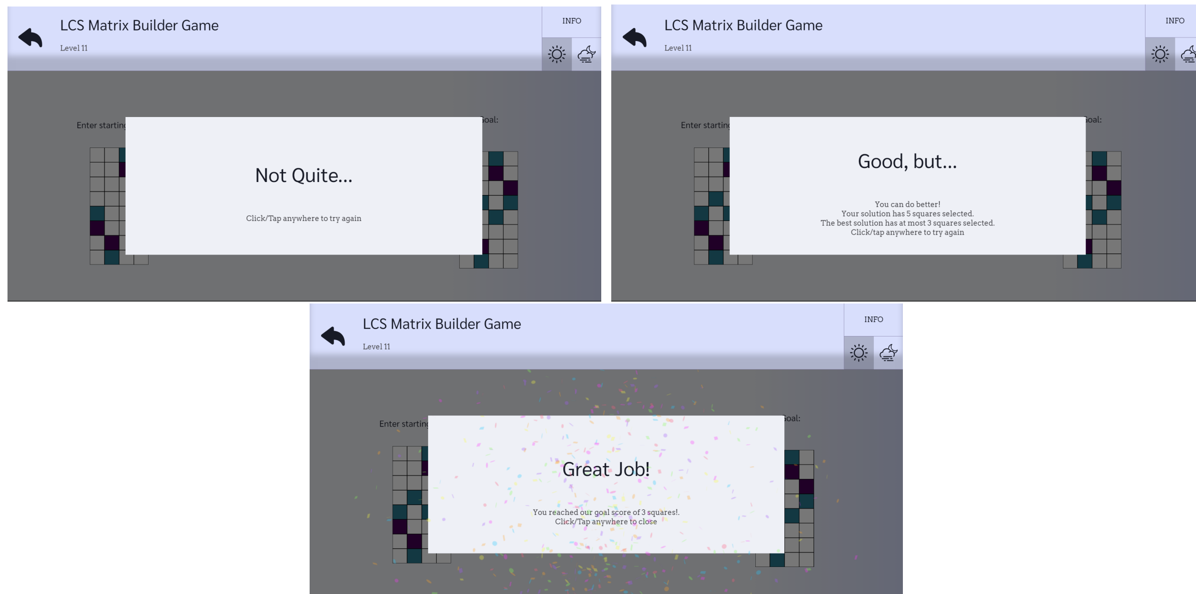


Figure 4.9: After submitting their solution, the user is informed whether their solution is incorrect, correct but suboptimal, or optimal. For those who achieve the optimal solution, a cascade of confetti adds a celebratory touch!

4.3.3 Derivation and Configuration Visualizer

The Derivation and Configuration Visualizer is an interactive tool designed to enhance user understanding of the dynamic programming algorithm for finding the longest common subsequence of two strings. Users can input two strings into the tool, and the system will automatically generate and display the memoization table. This feature enables users to visualize the intermediate values the dynamic programming algorithm uses to calculate the length of the LCS.

If a longest common subsequence exists for the given input strings, a dedicated button will appear below the table. Clicking this button initiates an animated backtracking algorithm, allowing users to dynamically observe the derivation of the LCS. Furthermore, users have the option to explore all possible configurations of the LCS within each of the input strings, additionally seeing the derivation using that configuration. The Derivation and Configuration Visualizer offers a visually intuitive representation of the derivation process, providing users with a comprehensive insight into both the dynamic programming algorithm and the underlying structure of the LCS.

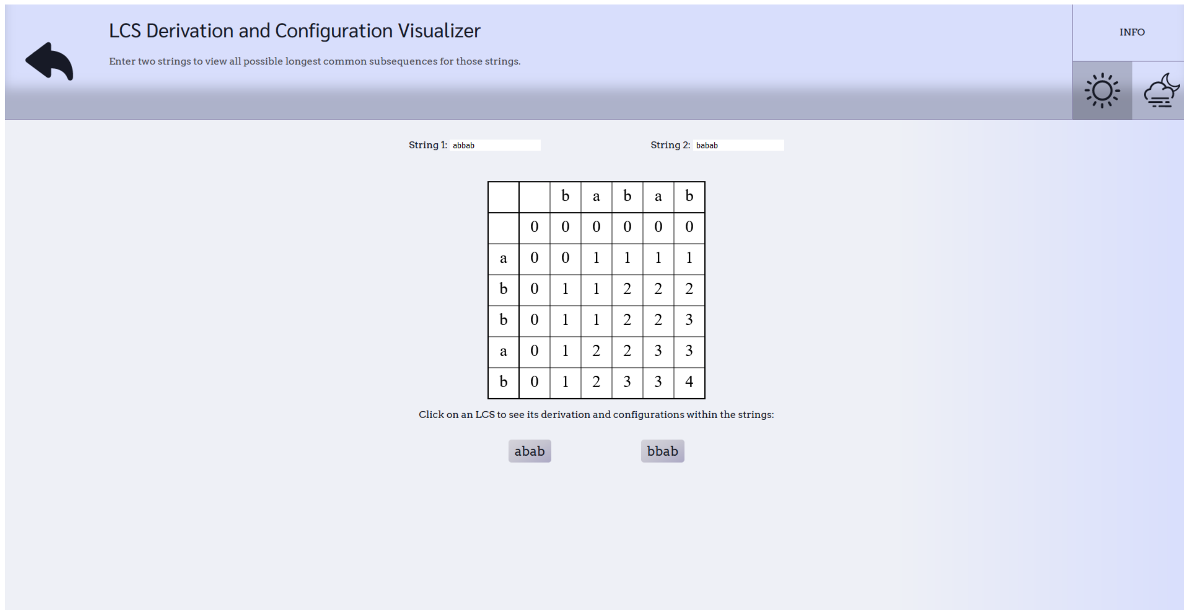


Figure 4.10: The user inputs the strings *abbab* and *babab* into the Derivation and Configuration Visualizer. The memoization table is generated and we observe that the longest common subsequences of the two strings are *abab* and *bbab*, as indicated by the buttons below the memoization table.

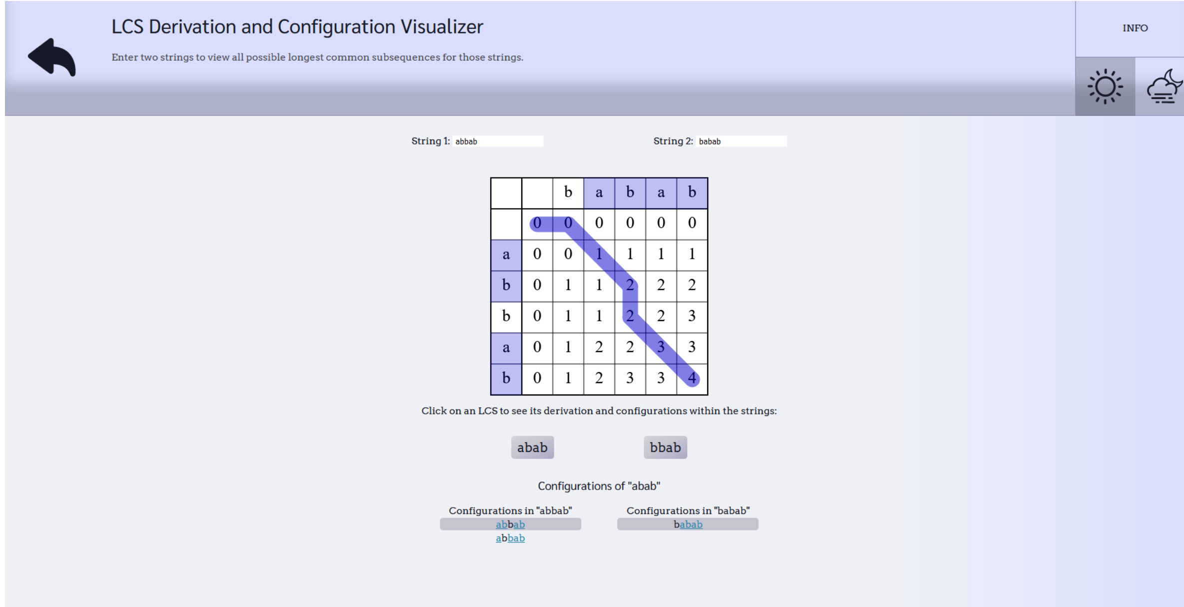


Figure 4.11: Upon clicking the button for the LCS *abab*, the tool animates the backtracking algorithm, allowing us to observe the derivation of the LCS from the memoization table. We also notice below the buttons a list of configurations of *abab* within the two input strings.

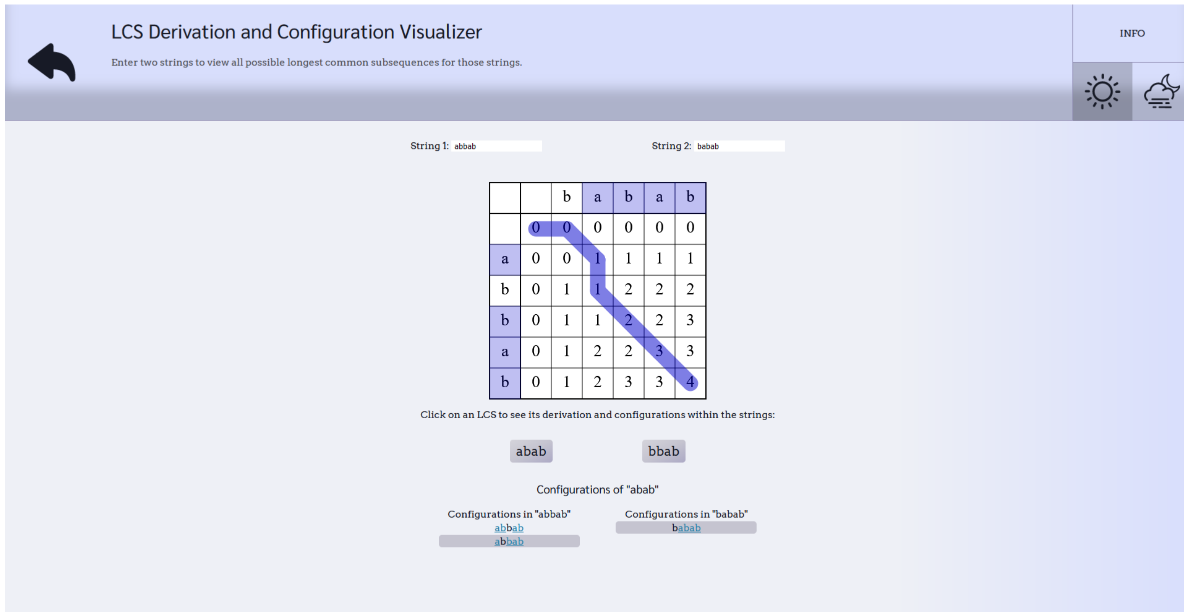


Figure 4.12: Upon selecting the second configuration of *abab* within *abbab*, the tool animates the backtracking algorithm again, this time using the second configuration as the path.

4.3.4 Distribution Explorer

The Distribution Explorer tool provides users with an interactive platform to explore the distribution of LCS strings taken from all pairs of strings of length n and m over an alphabet of size σ. The maximum values for n and m are capped at 10 and the maximum value of σ is capped at 15. This is to allow for a more comprehensive analysis of the data while still accounting for the exponential growth of the data. Upon inputting the values for n, m, and σ, users can examine a bar chart showcasing how many times each substring occurs as an LCS of the two strings. This feature provides insights into the frequency distribution of different substrings appearing as the LCS. Several edit properties can also be seen as patterns within the distribution. To make the bar chart easier to parse for large sizes of n and m, users can also apply some filters to the chart based on the distribution properties introduced earlier.

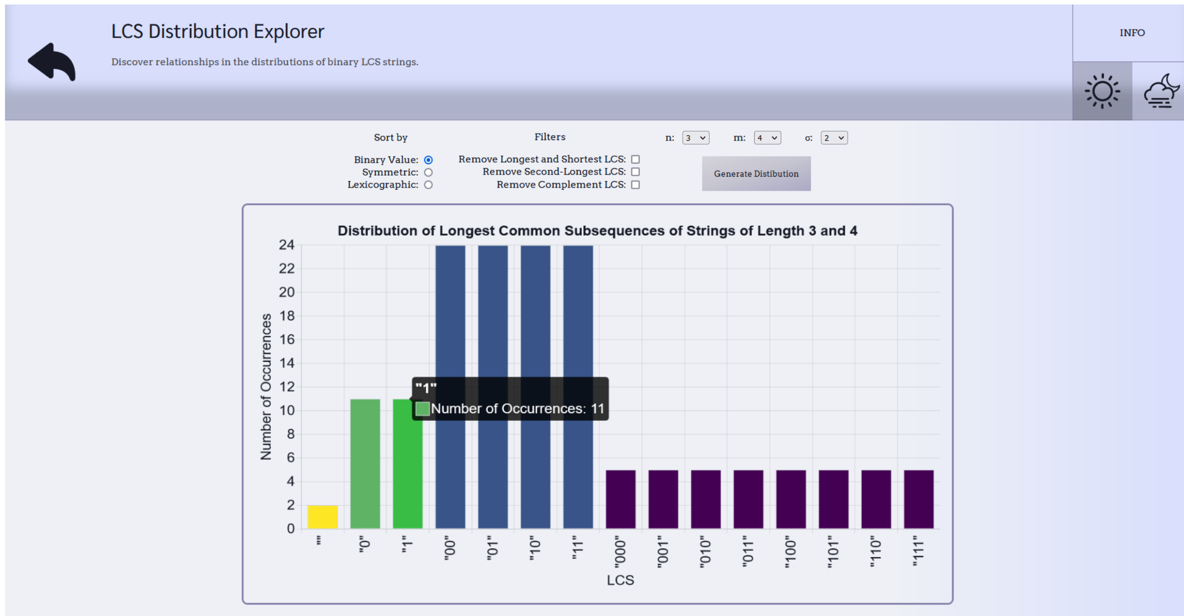


Figure 4.13: The user inputs the lengths of two strings, one of length three and the other of length four, over an alphabet of size two into the Distribution Explorer tool. The tool displays the frequency of each LCS length, as well as the range of LCS lengths. By default, the tool sorts the strings by their binary value. Upon hovering over the bar representing the substring *1*, a popup appears informing the user that it occurs as an LCS 11 times.

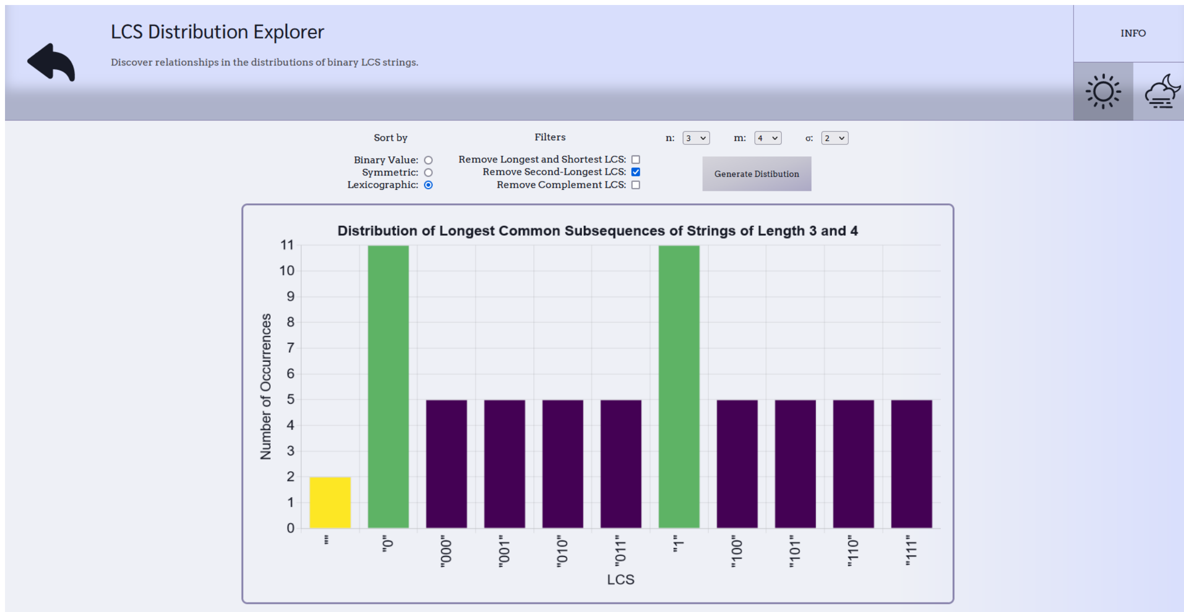


Figure 4.14: The user selects the option to sort the strings by their lexicographic order. Furthermore, the user has elected to remove the second-longest substrings, since those appear an equal number of times and thus do not provide any additional information.

4.4 Implementation Details

4.4.1 Edit Properties Explorer

Model Generator

The pre-generated models utilized within the Edit Properties Explorer are crafted through a Python script. Each model is defined within two files adhering to the Wavefront Object format: a `.obj` file containing vertex, normal, and texture coordinate data, and a `.mtl` file specifying material properties. These files, following a consistent naming convention, facilitate seamless loading and rendering within the Edit Properties Explorer. The script is structured as follows:

1. `modelZero()`: This function generates a scaled 3D representation of the number zero, used within the labels of the model. The function takes four parameters:
   - `position` – a 3D vector representing the position of the model in space
   - `scaleFactor` – the amount the label should be scaled to fit the model
   - `vertexList` – a list of 3D coordinates representing the vertices of the model
   - `nextVertexIndex` – the index of where the next vertex will be added to `vertexList`

The function begins by defining two lists: `zeroVertices`, which contains the coordinates that make up the zero model, and `vertexIndices`, whcih defines the faces of the model using indices into `zeroVertices`. Subsequently, each vertex in `zeroVertices` undergoes scaling by `scaleFactor` and translation by `position` before being appended to `vertexList`. Next, a list of faces is created and populated with `Face` objects, each defined by a list of vertex indices. Finally, `position` is updated to determine the placement of subsequent label models, and a `FaceComponents` object is returned, encapsulating the `vertexList`, `faces`, and `nextVertexIndex`, which are essential for further model construction.

2. `modelOne()`: Analogous to `modelZero`, this function generates a scaled 3D representation of the number one, contributing to label model creation.

3. `constructFace()`: This function is responsible for creating faces within a 3D model. It takes five parameters:

    - `vertices` – the vertices of the face
    - `vertexList` – the vertices already defined in the model
    - `vertexMap` – a dictionary mapping vertices to their indices in `vertexList`
    - `normalIndex` – the index of the normal for the face
    - `nextVertexIndex` – the index of where the next vertex will be added to `vertexList`

    The function starts by initializing the empty array `vertexIndices` to record the indices of the vertices of the face. Next, the function iterates over `vertices` to populate `vertexIndices`. During each iteration, it checks if the current vertex is already in `vertexMap`. If it is, the function retrieves the index of this vertex from `vertexMap` and assigns it to the corresponding position in `vertexIndices`. If the vertex is not in `vertexMap`, the function appends the vertex to `vertexList`, adds the vertex and its index to `vertexMap`, assigns the index to the corresponding position in `vertexIndices`, and increments `nextVertexIndex`. Upon completion, a `Face` object is instantiated from `vertexIndices` and `normalIndex`, and a `FaceComponents` object is returned, comprised of the newly created `Face` object, the updated `vertexList`, `vertexMap`, and `nextVertexIndex`.

4. `genObj()`: This function is responsible for creating the `.obj` file for the model. It takes three parameters: `n` and `m` are dimensions for the matrix, and `objWriter` is a file writer object that writes the generated model to a file. The function starts by writing a line to the file that specifies the material library for the model. It then constructs a matrix of size `n` by `m` which is used as the basis for the model. The base of the model is set, and `vertexMap` and `nextVertexIndex` are initialized for later use in tracking vertices. The function then initializes empty lists for vertices, UVs, and normals. It also initializes an empty list for objects, which will store the 3D objects (cubes and labels) that make up the model. The function then enters a nested loop that iterates over the matrix.

For each cell in the matrix, it creates a cube object. The cube is defined by its corners, which are calculated based on the cell's position in the matrix and its value. The cube is then added to the list of objects. After creating all the cubes, the function creates labels for the rows and columns of the matrix. These labels are also added to the list of objects.Finally, the function writes all the vertices, UVs, normals, and objects to the file. Each object is written following the format required by the `.obj` file format.

5. `genMtl()`: This function is responsible for generating the material library for a 3D model. The function takes three parameters: `n` and `m` are dimensions for the matrix, and `mtlWriter` is a file writer object that writes the generated material library to a file. The function starts by determining the number of colors needed. It then creates a gradient of colors based on a specific set of color stops. The function then enters a loop that iterates over the colors. For each color, it writes a new material to the file. The material is defined by its ambient color (`Ka`), diffuse color (`Kd`), and specular color (`Ks`), which are all set to the current color. The material also has an emissive color (`Ke`), which is set to black, a specular exponent (`Ns`), which is set to `10.0`, an optical density (`Ni`), which is set to `1.45`, a dissolve factor (`d`), which is set to `1.0`, and an illumination model (`illum`), which is set to `2`. After writing all the color materials, the function writes a material for the string labels. This material is black and has the same properties as the color materials.

Exploratory Tool

The Edit Properties Explorer was created using WebGL. WebGL is a JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins. For this project, we created a modular system that allows for the easy creation of new visualizations. This system spans over fifteen files, seventeen classes, and roughly 2000 lines of code. Leveraging this system facilitated the tool's development, debugging, and revision process. The driver code for the visualization is organized as follows:

1. `main()`: This function initializes the application, establishing global references to HTML input elements. It sets up the WebGL canvas, configures viewport, enables depth buffer and backface culling, activates shader program, initializes models, lighting, and camera, and registers event listeners for user input. Subsequently, it initiates the render loop.

2. `render()`: Serving as the main rendering loop, this function draws each frame of the 3D scene. It clears the canvas, checks if the 3D model is loaded, updates camera matrix, adjusts camera position and orientation, updates lighting parameters, model matrix, and draws the 3D model. This function continuously requests the next frame to be drawn, thus maintaining a rendering loop.

3. `changeMatrix()`: This function adjusts the matrix dimensions based on user input. It validates input values, prompts the user if larger models may lead to longer rendering times, and reinitializes the model, lighting, and camera to prepare for rendering the updated matrix.

4. `changeLCS()`: Responsible for selecting the cell in the matrix requested by the user, this function parses binary strings, calculates the index of the mesh, retrieves the length of the LCS, generates the set of LCSs, and updates the user interface accordingly before clearing edit operation inputs.

5. `performOperation()`: This function executes specified operations on provided binary strings, displaying the result in the matrix. Depending on the operation, this function either complements the kth bit of the second string, permutes the characters of the second string, computes bitwise complements, or reverses both strings. It calculates the index of the new cell, selects it, retrieves the length of the LCS, generates the set of LCSs for the new strings, and updates the user interface.

4.4.2 Matrix Builder

Exploratory Mode

The Matrix Builder is built using JavaScript and heavily relies on scalable vector graphic (SVG) elements to visualize the matrix, as they are dynamic and automatically scale to fit the content. The code is organized into several helper functions, each performing a specific task:

1. `setup()`: This function establishes global references to various HTML input elements, configures the SVG matrix, and sets up event listeners for input elements. Additionally, it generates the color gradient for the matrix and initializes the matrix with a default size of 1x1. The matrix is comprised of Cell objects, a data structure simplifying the matrix update process. The Cell object tracks the cell's location, value, and derivation method.

2. `selectCell()`: This function is used as an event handler for click events on the cells of the matrix. This function begins by calculating the row and column of the clicked cell. It then checks if the cell has existing information. If so, the function resets the cell's information, removing it from the selected cells array if applicable. If the cell has no information, the function calculates the LCS length of the cell. The cell is marked as user-selected, added to the selected cells array, and its fill color is set based on the length property. The color is obtained from the gradient map using the length as the index.

3. `fillMatrix()`: It begins by creating an empty queue and populating it with the currently selected cells. The function then enters a loop until the queue is empty. In each iteration, the function removes the first cell from the queue, retrieves its row and column, and checks four properties for each cell: commutative, complement, reverse, and concatenation. Depending on certain conditions, the function updates the corresponding cell in the matrix and adds it to the queue. The commutative property is only checked if the matrix is square, and it fills in symmetric cells with respect to the principal diagonal. The complement property fills in symmetric cells with respect to the counter

diagonal. The reverse property fills in the cell at the reverse of the coordinates in their binary representation. The concatenation property fills in cells resulting from concatenating a string with a portion of that cell. The function updates the length and derivation of the cell in the matrix, changes the fill attribute to match the current cell, and adds it to the queue.

Puzzle Mode

The Puzzle Mode of the Matrix Builder uses the same underlying structure as the Exploratory Mode, with additional functionality to support the puzzle-solving process.

1. `loadLevelData()`: This function takes a level number as input and loads the corresponding level data from a JSON file. The function starts by asychronously fetching the JSON file from the server. The fetched data is then parsed as text and stored in several variables:

    - `rows` – the number of rows in the matrix
    - `columns` – the number of columns in the matrix
    - `allowedProperties` – a map of which edit properties are enabled for the level
    - `goal` – the path to the SVG file containing the goal matrix
    - `optimalSolution` – the optimal solution for the level
    - `notes` – additional information about the level

    The function then initializes the matrix with the specified dimensions, sets the allowed properties for the level, and loads the goal matrix from the SVG file. The goal matrix is displayed to the user, alongside the notes for the level.

2. `checkSolution()`: This verification function begins by evaluating the user's solution and applying the `fillMatrix()` function from Exploratory mode to the input matrix. It then compares the input matrix with the goal matrix. If the values in both are the same, the function proceeds to check if the user's solution is optimal, meaning the user selected the fewest number of entries required. If the user's solution is optimal, a success message is displayed. In case the solution is correct but not optimal, the function indicates so with an appropriate message. If the matrices fail to match, the function notifies the user of the failure and resets the matrix.

4.4.3 Derivation and Configuration Visualizer

The Derivation and Configuration Visualizer, similar to the Matrix Builder Tool, is constructed using JavaScript and SVG elements. The code is also organized in a modular fashion, with each function serving a specific purpose:

1. `setup()`: This function establishes global references to various HTML input elements and configures the SVG table.

2. `fillTable()`: This function is responsible for populating the table with values from the input strings. It retrieves the input values, truncates strings longer than 10 characters to manage the LCS complexity, clears and appends a new SVG table, and manages the display of the configurations section based on the existence of an LCS.

3. `generateSVGTable()`: This function generates an SVG table representing the LCS problem. It calculates the dimensions, applies the dynamic programming algorithm to compute the LCS table and subsequences, creates buttons for each subsequence, sets table dimensions and outlines, populates cells with appropriate values, and appends the table to the HTML document.

4. `displayLCSInformation()`: Displays information about a specific LCS in a separate HTML element, updating the content of the configurations section and triggering animation of the backtracking process.

5. `animateBacktracking()`: Animates the backtracking process through the LCS table. It retrieves the backtracking path, disables user interaction during animation, animates the path by drawing lines and highlighting relevant cells, and re-enables user interaction after animation completion.

4.4.4 Distribution Explorer

The Distribution Explorer is primarily constructed using Chart.js, which is a third-party open source library for generating charts in JavaScript. Because there are a lot of parameters involved in generating a chart, here we give only a high-level overview of the steps taken to generate a distribution:

The distribution data is pre-generated from a Python file and stored in a JSON file named after the distribution dimensions. The `generateDistribution()` function begins by checking if a chart object already exists. If so, it clears the canvas and removes any residual styling. It retrieves the values of n, m, and σ from user input boxes and initializes empty arrays for strings and occurrences. The function determines the filename of the JSON file based on the values of n, m, and σ, ensuring the correct file retrieval. It generates a gradient map for the chart's color bars and resizes the chart area accordingly. Asynchronously, the function fetches the JSON file where the distribution data is stored. The fetched data is parsed as text and stored in a variable. The function iterates over the string occurrences property of the data object, extracting strings and occurrences and populating the respective arrays. Options for scales and zoom of the chart are then defined. Finally, a new Chart object is created, passing in the chart area element, chart type, chart data, and options. The chart object is stored in a variable for further manipulation if needed.

Chapter 5

Conclusion and Future Work

The overarching aim of this project was to develop mathematical results and an interactive tool related to the longest common subsequence problem. Based on our background research, we chose to focus mainly on the Chvátal-Sankoff constants, where we established new state-of-the-art lower bounds in the classical two binary string case as well as in the extended, general cases. We additionally explored properties of the LCS problem more generally, proving, to the best of our knowledge, several novel results concerning the effects of character edits on longest common subsequences and the distributions of longest common subsequences of certain lengths. To complement these results, we created BLISS Playground, a website which allows for deeper visualizations of the LCS problem. To our knowledge, this is the only publicly available web application designed to explore the LCS problem beyond just its dynamic programming solution, and in fact, the only website we could find that correctly visualizes the dynamic programming algorithm's ability to locate *all* the longest common subsequences of an input, instead of just one of them.

To improve theoretical bounds on the well-studied Chvátal-Sankoff constants, we took three approaches to improving a previous algorithm in the literature: parallelizing independent recurrence computations, recursively sub-chunking the problem so that memory I/O could be performed sequentially, and exploiting symmetries inherent to the LCS problem. While the techniques we employed here are specific to this particular algorithm, they demonstrate how restructuring the way sequences are represented as data away from a purely contiguous format can be of significant benefit, an idea which may be applicable in speeding up other sequence algorithms.

We created BLISS Playground initially as a tool to aid in our own exploration of the LCS problem. However, this soon grew into a project aiming to address the gap of a publicly available research and education tool for visualizing and understanding the LCS problem more holistically. The vision of BLISS was to aid in the exploration of the LCS problem through intuitive visualizations and to educate users about the many patterns inherent to the seemingly simple problem. Indeed, BLISS has already served its purpose well: we discovered Property 15 as a result of playing around with its Distribution Explorer function. In keeping with our vision of BLISS as a tool useful for education, exploration, and research alike, the repository for BLISS is publicly available for anyone to contribute to.

Combining our mathematical results and interactive website together in this project makes it both of academic interest and accessible to a wider audience. Subsequences are a fundamental concept in both computer science and math, and so long as the LCS problem sees applications in computational biology, computational linguistics, version control, or any of the other myriad domains it touches, understanding them will remain as important as ever.

5.1 Future Work

While we have made several contributions to the LCS problem, we believe that there is still room for improvement. In particular, the generalized version of the algorithm we present in Section 3.4 and Section 3.3 can be further optimized by taking advantage of the powerful symmetries present in the LCS problem. For example, an LCS of a d-tuple of strings does not depend on the order of those strings, potentially saving up to a factor of $d!$ (the number of permutations of d strings) computations. Similarly, the LCS length is independent of the labelling of each character in the string, meaning the same evaluation is performed up to $\sigma!$ times on strings that are identical up to relabelling (e.g., the length of an LCS of *bcab* and *babc* is equal to the length of an LCS of *cabc* and *cbca*). This may allow for significantly improved lower bounds in the general case, especially for larger values of σ and d.

While approximations of the value of the Chvátal-Sankoff constant for pairs of binary strings have been made (see Table 3.1), it seems that most of the work for the general case has been focused on tightening the upper and lower bounds. We believe that performing similar calculations to obtain precise approximations of different $\gamma_{\sigma,d}$ could be an avenue for future research yielding noteworthy results. Kiwi and Soto, as well as Steele, express interest in finding a relationship between different $\gamma_{\sigma,d}$ [30, 45]. We agree that such a relationship, if it exists, would be interesting, and also wonder if having precise approximations for the constants would aid in finding a relationship.

In a time where artificial intelligence is becoming increasingly popular, we are also curious if massive parallel computing power combined with techniques such as deep reinforcement learning [42] and new neural network-based sequence alignment models [29] could lead to further improvements on the lower or upper bounds on the Chvátal-Sankoff constant. If so, it would be intriguing to investigate whether these techniques may be used to solve additional combinatorial string computation problems.

We also believe BLISS Playgounrd could be expanded in a number of ways. For instance, the Matrix Builder puzzle mode could be extended in such a way as to gradually introduce the user to LCS properties of increasing complexity in a hands-on way, with accompanying diagrams and textual explanation. Further, the Derivation and Configuration Visualizer could be augmented with additional explanation of the dynamic programming algorithm beyond what this report provides. User testing should additionally be performed to identify pain points, confusing features, and additional opportunities for improvement. Ultimately, the goal could be for BLISS to find its way into classrooms as a fun, intuitive way to teach dynamic programming and explore the LCS problem more holistically.

Source code for our implementations of the algorithms described in this report and BLISS Playground can be found at https://github.com/Statistics-of-Subsequences.

References

[1] A. Abboud, A. Backurs, and V. V. Williams. "Tight Hardness Results for LCS and Other Sequence Similarity Measures". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. Ed. by Venkatesan Guruswami. IEEE Computer Society, 2015, pp. 59–78. DOI: 10.1109/FOCS.2015.14. URL: https://doi.org/10.1109/FOCS.2015.14.

[2] R. Arratia and M. S. Waterman. "An Erdös-Rényi Law with Shifts". In: *Advances in Mathematics* 55.1 (Aug. 2003), pp. 13–23. DOI: 10.1016/0001-8708(85)90003-9.

[3] R. Arratia and M. S. Waterman. "Critical Phenomena in Sequence Matching". In: *The Annals of Probability* 13.4 (Nov. 1985), pp. 1236–1249. ISSN: 00911798. URL: http://www.jstor.org/stable/2244175.

[4] K. Baba, T. Nakatoh, and T. Minami. "Plagiarism detection using document similarity based on distributed representation". In: *Procedia Computer Science* 111 (2017). The 8th International Conference on Advances in Information Technology, pp. 382–387. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.06.038. URL: https://www.sciencedirect.com/science/article/pii/S1877050917312115.

[5] R. A. Baeza-Yates et al. "Bounding the Expected Length of Longest Common Subsequences and Forests". In: *Theory of Computing Systems* 32.4 (Aug. 1999), pp. 435–452. ISSN: 1433-0490. DOI: 10.1007/s002240000125. URL: https://doi.org/10.1007/s002240000125.

[6] R. Beal et al. "A new algorithm for "The lcs problem" with application in compressing genome resequencing data". In: *BMC Genomics* 17.S4 (Aug. 2016). DOI: 10.1186/s12864-016-2793-0.

[7] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 2010.

[8] National Center for Biotechnology Information. *Basic Local Alignment Search Tool*. URL: https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[9] J. Boutet de Monvel. "Extensive simulations for longest common subsequences". In: *The European Physical Journal B - Condensed Matter and Complex Systems* 7.2 (Jan. 1999), pp. 293–308. DOI: 10.1007/s100510050616. URL: https://doi.org/10.1007/s100510050616.

[10] S. Budalakoti et al. "Anomaly detection in large sets of high-dimensional symbol sequences". In: *2006 SIAM International Conference on Data Mining*. NASA/TM-2006-214553. 2006.

[11] B. Bukh and C. Cox. "Periodic words, common subsequences and frogs". In: *The Annals of Applied Probability* 32.2 (2022), pp. 1295–1332. DOI: 10.1214/21-AAP1709. URL: https://doi.org/10.1214/21-AAP1709.

[12] R. Bundschuh. "High precision simulations of the longest common subsequence problem". In: *The European Physical Journal B - Condensed Matter and Complex Systems* 22.4 (Aug. 2001), pp. 533–541. ISSN: 1434-6036. DOI: 10.1007/s100510170102. URL: https://doi.org/10.1007/s100510170102.

[13] S. Changder, D. Ghosh, and N. C. Debnath. "LCS based text steganography through Indian Languages". In: *2010 3rd International Conference on Computer Science and Information Technology*. Vol. 8. 2010, pp. 53–57. DOI: 10.1109/ICCSIT.2010.5563974.

[14] V. Chvátal and D. Sankoff. "Longest Common Subsequences of Two Random Sequences". In: *Journal of Applied Probability* 12.2 (1975), pp. 306–315. ISSN: 00219002. URL: http://www.jstor.org/stable/3212444.

[15] T. H. Cormen and C. H. Leiserson. *Introduction to Algorithms*. en. Fourth. London, England: MIT Press, Apr. 2022.

[16] V. Dančík. "Common Subsequences and Supersequences and their Expected Length". In: *Combinatorics, Probability and Computing* 7.4 (1998), pp. 365–373. DOI: 10.1017/S096354839800368X.

[17] V. Dančík. "Expected Length of Longest Common Subsequences". PhD thesis. University of Warwick, 1994.

[18] V. Dančík and M. Paterson. "Upper bounds for the expected length of a longest common subsequence of two binary sequences". In: *Random Struct. Algorithms* 6.4 (July 1995), pp. 449–458. ISSN: 1042-9832.

[19] D. Das and B. Saha. "Approximating lcs and alignment distance over multiple sequences". In: *arXiv preprint arXiv:2110.12402* (2021).

[20] J. G. Deken. "Some limit results for longest common subsequences". In: *Discrete Mathematics* 26.1 (1979), pp. 17–31. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(79)90057-8. URL: https://www.sciencedirect.com/science/article/pii/0012365X79900578.

[21] John B Fraleigh. *A First Course in Abstract Algebra*. en. 7th ed. Upper Saddle River, NJ: Pearson, Nov. 2002.

[22] M. Hajiaghayi et al. "Approximating LCS in Linear Time: Beating the $\sqrt{n}$ Barrier". In: *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1181–1200. DOI: 10.1137/1.9781611975482.72. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975482.72. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975482.72.

[23] T. S. Han, S. Ko, and J. Kang. "Efficient Subsequence Matching Using the Longest Common Subsequence with a Dual Match Index". In: *Machine Learning and Data Mining in Pattern Recognition*. Ed. by Petra Perner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 585–600. ISBN: 978-3-540-73499-4.

[24] X. He and R. Li. "Approximating binary longest common subsequence in almost-linear time". In: (2023). arXiv: 2211.16660 [cs.DS].

[25] Erik Hirmo, Jüri Lember, and Heinrich Matzinger. *Detecting the homology of DNA-sequences based on the variety of optimal alignments: a case study*. 2012. arXiv: 1210.3771 [stat.AP].

[26] J. W. Hunt and M. D. MacIlroy. "An algorithm for differential file comparison". In: (2008).

[27] H. Huo et al. "Efficient Compression and Indexing for Highly Repetitive DNA Sequence Collections". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 18.6 (2021), pp. 2394–2408. DOI: 10.1109/TCBB.2020.2968323.

[28] S. Y. Itoga. "The string merging problem". In: *BIT* 21.1 (Mar. 1981), pp. 20–30. ISSN: 1572-9125. DOI: 10.1007/bf01934067. URL: http://dx.doi.org/10.1007/BF01934067.

[29] J. Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (2021), pp. 583–589.

[30] M. Kiwi and J. Soto. "On a Speculated Relation Between Chvátal–Sankoff Constants of Several Sequences". In: *Combinatorics, Probability and Computing* 18.4 (July 2009), pp. 517–532. ISSN: 1469-2163. DOI: 10.1017/s0963548309009900. URL: http://dx.doi.org/10.1017/S0963548309009900.

[31] Marcos Kiwi, Martin Loebl, and Jiří Matoušek. "Expected length of the longest common subsequence for large alphabets". In: *Advances in Mathematics* 197.2 (2005), pp. 480–498. ISSN: 0001-8708. DOI: https://doi.org/10.1016/j.aim.2004.10.012. URL: https://www.sciencedirect.com/science/article/pii/S0001870804003536.

[32] J. B. Kruskal and D. Sankoff. *Time warps, string edits, and macromolecules : the theory and practice of sequence comparison*. eng. Reading, Mass: Addison-Wesley Pub. Co., Advanced Book Program, 1983. ISBN: 0201078090.

[33] E. Lehman. *Mathematics for Computer Science*. Samurai Media, Mar. 2017.

[34] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet Physics Doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[35] Peter Linz and Susan H Rodger. *An introduction to formal languages and automata*. en. 7th ed. Sudbury, MA: Jones and Bartlett, Mar. 2022.

[36] G. S. Lueker. "Improved bounds on the average length of longest common subsequences". In: *J. ACM* 56.3 (May 2009). ISSN: 0004-5411. DOI: 10.1145/1516512.1516519. URL: https://doi.org/10.1145/1516512.1516519.

[37] W. J. Masek and M. S. Paterson. “A faster algorithm computing string edit distances”. In: *Journal of Computer and System Sciences* 20.1 (1980), pp. 18–31. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(80)90002-1. URL: https://www.sciencedirect.com/science/article/pii/0022000080900021.

[38] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[39] G. Navarro. “A guided tour to approximate string matching”. In: *ACM Comput. Surv.* 33.1 (Mar. 2001), pp. 31–88. ISSN: 0360-0300. DOI: 10.1145/375360.375365. URL: https://doi.org/10.1145/375360.375365.

[40] R. Roy and S. Changder. “Image realization steganography with LCS based mapping”. In: *2014 Seventh International Conference on Contemporary Computing (IC3)*. 2014, pp. 218–223. DOI: 10.1109/IC3.2014.6897176.

[41] D. Sankoff and R. J. Cedergren. “A test for nucleotide sequence homology”. In: *Journal of Molecular Biology* 77.1 (1973), pp. 159–164. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(73)90369-0. URL: https://www.sciencedirect.com/science/article/pii/0022283673903690.

[42] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[43] M. Sipser. *Introduction to the theory of computation*. 3rd ed. Belmont, CA: Wadsworth Publishing, June 2012.

[44] T. Smith and M. Waterman. “New Stratigraphic Correlation Techniques”. In: *Journal of Geology* 88 (July 1980). DOI: 10.1086/628528.

[45] J. M. Steele. “An Efron-Stein Inequality for Nonsymmetric Statistics”. In: *The Annals of Statistics* 14.2 (1986), pp. 753–758. DOI: 10.1214/aos/1176349952. URL: https://doi.org/10.1214/aos/1176349952.

[46] M. J. Steele. *Probability Theory and Combinatorial Optimization*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611970029. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970029. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970029.

[47] Alexander Tiskin. *The Chvátal-Sankoff problem: Understanding random string comparison through stochastic processes*. 2022. arXiv: 2212.01582 [math.CO].

[48] C. Wang and D. Zhang. “A novel compression tool for efficient storage of genome resequencing data”. In: *Nucleic Acids Research* 39.7 (Jan. 2011), e45–e45. ISSN: 0305-1048. DOI: 10.1093/nar/gkr009. eprint: https://academic.oup.com/nar/article-pdf/39/7/e45/16783414/gkr009.pdf. URL: https://doi.org/10.1093/nar/gkr009.

[49] M. S. Waterman. *Introduction to computational biology*. en. Chapman & Hall/CRC Interdisciplinary Statistics. Philadelphia, PA: Chapman & Hall/CRC, June 1995.

[50] M. Yulianto, R. Arifundin, and A. Alamsyah. "Autocomplete and Spell Checking Levenshtein Distance Algorithm To Getting Text Suggest Error Data Searching In Library". In: *Scientific Journal of Informatics* 5 (May 2018), p. 75. DOI: 10.15294/sji.v5i1.14148.

Appendix A

Complete Results for General $\gamma_{\sigma,d}$ Constants

Here we list out all of our results for the Feasible Triplet algorithm for all σ, d values, even for smaller values of ℓ which did not result in better lower bounds.

$\ell = 1$		σ								
		2	3	4	5	6	7	8	9	10
d	2	0.666666	0.500000	0.400000	0.333333	0.285714	0.250000	0.222222	0.200000	0.181818
	3	0.666666	0.488372	0.384615	0.317073	0.269662	0.234567	0.207547	0.186104	0.168674
	4	0.615384	0.450000	0.352583	0.289398	0.245283	0.212786	0.187869	0.168164	0.152193
	5	0.615384	0.432494	0.335517	0.273884	0.231234	0.200004			
	6	0.592592	0.421434	0.324014	0.263369					
	7	0.592592	0.413611	0.317032						
	8	0.579185	0.405539							
	9	0.579185	0.400949							
	10	0.570155								
	11	0.570155								
	12	0.563566								
	13	0.563566								
	14	0.558494								
	15	0.558494								

$\ell = 2$		σ								
		2	3	4	5	6	7	8	9	10
d	2	0.727273	0.620690	0.542373	0.480769	0.431138	0.390438	0.356545	0.327935	0.303490
	3	0.673913	0.516896	0.421518	0.356717	0.309424	0.273275	0.244710	0.221555	0.202402
	4	0.643216	0.484937	0.389008	0.324338	0.277835	0.242798			
	5	0.626506	0.461402	0.365329	0.302236					
	6	0.610925	0.445434	0.349848						
	7	0.602493	0.434514							
	8	0.594016	0.425774							
	9	0.587900								
	10	0.582349								
	11	0.578464								
	12	0.574269								
	13	0.571067								

$\ell = 3$		σ								
		2	3	4	5	6	7	8	9	10
d	2	0.747922	0.644966	0.573254	0.521091	0.479452	0.444577	0.414651	0.388537	0.365485
	3	0.687410	0.545373	0.457311	0.394945	0.347798				
	4	0.651309	0.498525	0.405702						
	5	0.632165	0.474304							
	6	0.617761								
	7	0.607261								
	8	0.598782								
	9	0.592177								

$\ell = 4$		σ								
		2	3	4	5	6	7	8	9	10
d	2	0.758576	0.657642	0.589484	0.539129	0.499229	0.466481	0.438799	0.414876	0.393811
	3	0.692950	0.556649	0.472979						
	4	0.657241	0.509237							
	5	0.636022								
	6	0.621057								

$\ell = 5$		σ			
		2	3	4	5
d	2	0.765446	0.665874	0.599248	0.549817
	3	0.697737	0.564841		
	4	0.661274			
	5	0.639248			

$\ell = 6$		σ		
		2	3	4
d	2	0.770273	0.671697	0.605786
	3	0.701317		
	4	0.664722		

$\ell = 7$		σ		
		2	3	4
d	2	0.773975	0.676041	0.610590
	3	0.704473		

$\ell = 8$		σ		
		2	3	4
d	2	0.776860	0.679441	0.614333
	3	0.707165		

$\ell = 9$		σ	
		2	3
d	2	0.779259	0.682218
	3	0.709501	

$\ell = 10$		σ
		2
d	2	0.781281
	3	0.711548

Appendix B

Proofs of LCS Properties

Proof of Property 1.

Let x and y be strings of length n and m, respectively. The length of the LCS of x and y is bounded by the number of characters in the shorter string. This is because the LCS is a subsequence of both x and y, so it cannot contain more characters than either string. Therefore, the length of the LCS is at most $\min(n, m)$. The length of the LCS is also bounded below by zero, since if x and y have no common characters, then the LCS is the empty string. ■

Proof of Property 2.

The order of characters within x and y do not change, so the set of possible subsequences for x and y stay the same. The set of LCS is a subset of the set of possible subsequences for both x and y, so it will also stay the same. ■

Proof of Property 3.

Assume we know $\text{LCS}(x, y)$ for the given strings x and y. Without loss of generality, consider some $z \in \text{LCS}(x, y)$. Additionally, we define $X = Hx$ and $Y = Hy$. We begin by considering the scenario where $|H| = 1$. Since that the first character X and Y is the same, it follows that any longest common subsequence of X and Y must begin with this character. The remaining portion of the strings are x and y, and since we already know that the LCS of x and y is z, it naturally follows that the LCS of X and Y is Hz.

As the length of H increases, we can iteratively apply the same reasoning, working backward from the end of H to its beginning. Therefore, the LCS of Hx and Hy must be Hz, for all $H \in \Sigma^*$. A similar line of thought can be extended to the situation where we append the same string to both x and y, however this time we progress forward from the start of T to its end. Therefore, the LCS of xT and yT must be zT, for all $T \in \Sigma^*$. ■

Proof of Property 4.

Assume we have two strings $A, B \in \Sigma^*$ such that $\mathrm{LCS}(A, B) = \{\lambda\}$. Let z be any LCS of Ax and By. z cannot simultaneously include characters from both A and B, since that would violate the non-commonality between A and B. Therefore, z must either include characters from exactly one of A or B or neither.

In the case that z includes characters from A but not B, it must be a longest common subsequence of Ax and y, making $z \in \mathrm{LCS}(Ax, y)$. Similarly, if z includes characters from B but not A, it must be a longest common subsequence of By and x, making $z \in \mathrm{LCS}(x, By)$. We can see that the case where z includes characters from neither A nor B is already considered in the previous two cases. Therefore, $z \in \mathrm{LCS}(Ax, y) \cup \mathrm{LCS}(x, By)$. ■

Proof of Property 5.

We know that the LCS of x and z must be at least as long as the LCS of x and y, since y is a subsequence of z. Therefore, any LCS of x and y will also be a subsequence of x and z. However, we have to take into account the possibility that the addition of the character C increase the length of the LCS. This will happen if C is critical with respect to the LCS of x and z.

C is critical when we can affix C to an LCS of x and y to get a subsequence of x and z. Let x and y have an LCS w that allows for this. In this scenario, x and z have a common subsequence one longer than w, since the subsequence is w with C inserted to it. We cannot remove C from z, as we know that affixing C causes the subsequence to be longer then the LCS of x and y, and removing C from z gives us y. Thus, C is critical.

This common subsequence must also be an LCS, as any characters other than C that are part of the LCS would already be part of w, since those characters already existed in both x and y. Therefore, when C is critical with respected to the LCS of x and z, inserting C increases the LCS length by one. ■

Proof of Property 6.

Let C denote the character $y[k]$. We know that the LCS of x and z must be at most as long as the LCS of x and y, which is true when C is not a part of at least one LCS of x and y (in which case that LCS is also an LCS for x and z). However, this is not always the case. If C is critical for every LCS of x and y, then the length of the LCS will decrease since the LCS of x and z will not include C. Since the change of length is affected by one character, the length of the LCS of x and z will be one less than the length of the LCS of x and y. ■

Proof of Property 7.

Substitution is equivalent to deleting the kth character of the string and inserting C in the same position. Therefore, we can determine the effect of a substitution by applying Properties 6 and 5, in that order. There are four possible scenarios to consider:

1. **There is some $w \in \text{LCS}(x, y)$ such that $y[k]$ is not critical and the insertion of C into w forms an LCS of x and z.** Under Property 6, deleting $y[k]$ does not change the LCS length. Under Property 5, inserting C increases the LCS length. Overall, the LCS increases in length by one.

2. **There is some $w \in \text{LCS}(x, y)$ such that $y[k]$ is not critical but the insertion of C into w does not form an LCS of x and z.** Under Property 6, deleting $y[k]$ does not change the LCS length. Under Property 5, inserting C does not change the LCS length. Overall, the LCS length remains the same.

3. **For all $w \in \text{LCS}(x, y)$, $y[k]$ is critical but the insertion of C into some w forms an LCS of x and z.** Under Property 6, deleting $y[k]$ decreases the LCS length. Under Property 5, inserting C increases the LCS length. Overall, the LCS length remains the same.

4. **For all $w \in \text{LCS}(x, y)$, $y[k]$ is critical and the insertion of C into every w does not form an LCS of x and z.** Under Property 6, deleting $y[k]$ decreases the LCS length. Under Property 5, inserting C does not change the LCS length. Overall, the LCS decreases in length by one.

Therefore, the effect of a substitution on the LCS length depends on whether $y[k]$ is critical and whether the insertion of C into the LCS of x and y forms an LCS of x and z. ■

Proof of Property 8.

Permutations of characters are equivalent to performing a series of substitutions. Since the effect of a substitution changes the LCS length by at most one, the effect of repeated substitutions is equal to the sum of the effects of each individual substitution. In the case where every substitution decreases the LCS length, the total decrease in LCS length is equal to the number of substitutions. In the case where every substitution increases the LCS length, the total increase in LCS length is equal to the number of substitutions. Since any other scenario will involve a combination of increasing and decreasing the LCS length, the total change in LCS length will be in the range $[-k, k]$. ■

Proof of Property 9.

Let $z \in \text{LCS}(x, y)$. This means z is a subsequence of both x and y, and its length is maximized among all common subsequences of x and y. We know that z can be obtained by removing some characters from x and y to form a common subsequence. Therefore, we can remove the mappings of the same characters to form a common subsequence of $\varphi(x)$ and $\varphi(y)$. As a result, the common subsequence that is formed will be $\varphi(z)$.

Suppose there exists another common subsequence w of $\varphi(x)$ and $\varphi(y)$ such that $|w| > |\varphi(z)|$. Then, by applying φ^{-1} to w, we would obtain a common subsequence of x and y with length greater than $|z|$, which contradicts the maximality of $|z|$ as an LCS of x

and y. This means $\varphi(z)$ has maximum length among all common subsequences of $\varphi(x)$ and $\varphi(y)$. Therefore, $\varphi(z) \in \mathrm{LCS}\big(\varphi(x), \varphi(y)\big)$. ■

PROOF OF PROPERTY 10.

Let $z \in \mathrm{LCS}(x, y)$. This means z is a subsequence of both x and y, and its length is maximized among all common subsequences of x and y. We know that z can be obtained by removing some characters from x and y to form a common subsequence. Therefore, we can remove the same characters, however in reverse order, to form a common subsequence of x^{R} and y^{R}. As a result, the common subsequence that is formed will be z^{R}.

Suppose there exists another common subsequence w of x^{R} and y^{R} such that $|w| > |z^{\mathsf{R}}|$. Then, by reversing w, we would obtain a common subsequence of x and y with length greater than $|z|$, which contradicts the maximality of $|z|$ as an LCS of x and y. This means z^{R} has maximum length among all common subsequences of x^{R} and y^{R}. Therefore, $z^{\mathsf{R}} \in \mathrm{LCS}(x^{\mathsf{R}}, y^{\mathsf{R}})$. ■

PROOF OF PROPERTY 11.

Using Property 3, we establish that the length of the LCS of HxT and HyT is $|H|+\mathcal{L}(x,y)+|T|$. Similarly, applying Property 3 again, we find that the length of the LCS of PxS and PyS is $|P| + \mathcal{L}(x, y) + |S|$. Since $|H| = |P|$ and $|T| = |S|$, it follows that the length of the LCS of HxT and HyT equals that of PxS and PyS. ■

PROOF OF PROPERTY 12.

Let x and y be strings of length at least k such that $\mathcal{L}\big(x \setminus x[k], y \setminus y[k]\big) = \mathcal{L}(x, y) - 2$. Without loss of generality, delete $x[k]$ from x first. Since the LCS length can only decrease by one per deletion, deleting $x[k]$ must reduce the LCS length by one. Thus, by Property 6, $x[k]$ is critical for the LCS of x and y. For the same reason, $y[k]$ must be critical for the LCS of $x \setminus x[k]$ and y. But if $y[k]$ is critical in the LCS of $x \setminus x[k]$ and y, then it must exist in that LCS. Since $x[k]$ has already been deleted, $x[k] \neq y[k]$. ■

PROOF OF PROPERTY 13.

The number of LCS is minimized when x and y have no common symbols. In this case, the only LCS of x and y is the empty string, so the set of LCS has exactly one element.

Without loss of generality, assume $n < m$. The number of LCS is maximized when x is composed of distinct symbols and y contains x^{R} as a subsequence, with the remaining symbols of y being distinct from those in x. In this case, every symbol in x is a subsequence of y. We can see that there is no subsequence of x and y that is of length greater than one. This is because if there were, then either x or y would have to contain a repeated symbol, which would violate the assumption that x and y are each composed of distinct symbols.

Therefore, the set of LCS is the set of symbols in x, which has n elements. By the same reasoning, if $m < n$, the set of LCS has m elements. Therefore, the number of LCS is at most $\min(n, m)$. ■

Proof of Property 14.

Let z be a string of length between 0 and $\min(n, m)$. Construct strings $x = zC_1 \ldots C_1$ and $y = zC_2 \ldots C_2$, where $C_1, C_2 \in \Sigma$, such that $|x| = n$ and $|y| = m$. Since exactly the first $|z|$ characters in x and y are the same, and none of the remaining characters are the same, $z \in \text{LCS}(x, y)$. ■

Proof of Property 15

Note that this property only applies to pairs of binary strings. We will later expand this property to the more general case of any number of strings with an arbitrary alphabet. We will prove the binary property first, which requires new notation.

k-Block Formulation

To facilitate this proof, we introduce the k-block representation of a string. For any binary string, we can represent it as alternating blocks of contiguous *1*s and *0*s. For example, the string *1001110* is represented as follows:

$l =$	1	2	3	1
	1	*0*	*1*	*0*

We say that a binary string s of length $n-1$ is represented by k_0 blocks of *0*s and k_1 blocks of *1*s. Furthermore, we denote the length of each block of *0*s by $l^0_1, \ldots, l^0_{k_0}$, and the length of each block of *1*s by $l^1_1, \ldots, l^1_{k_1}$. We denote the number of *0* characters as $L_0 = \sum_{i=1}^{k_0} l^0_i$, and the number of *1* characters as $L_1 = n - 1 - L_0 = \sum_{i=1}^{k_1} l^1_i$. Finally, we define p and q as follows:

$$p = \left\{ \begin{array}{ll} 0, & \text{if } s \text{ begins and ends with } \mathit{1} \\ 1, & \text{if } s \text{ begins and ends with opposite symbols} \\ 2, & \text{if } s \text{ begins and ends with } \mathit{0} \end{array} \right\} = 2 - q$$

This representation will make it easier to reason about and demonstrate several useful properties.

Placing a Character in a Single String

We are interested in finding how many times a string s of length $n-1$ appears as an LCS in all possible pairs of two strings a and b of length n. To do this, we work in reverse.

Let $s = t = u$. Clearly, s is an LCS of t and u. Our original problem is equivalent to counting the number of unique string pairs made by inserting a character to t and a character to u such that s is still an LCS of both strings. This will necessarily count all possible pairs of strings of length n whose longest common subsequence is s.

We note that, starting with two copies of s, adding a single character to only one of the strings does not immediately affect the LCS length. Thus, to count the number of unique pairs with s as an LCS, we can count the number of *unique* ways to add a character to one string, then look at the number of *unique* ways to add a character to the second string such that the LCS length does not change.

Lemma 2. *There are exactly $k_0 + L_1 - k_1 + p$ unique ways of inserting a 0 in s.*

PROOF. Firstly, note that where you place a *0* inside or next to any given *0*-block does *not* matter. That is, there is only one unique way of placing a *0* inside or next to a given *0*-block. For instance, the following placements inside *1001110* are identical:

10001110   *10001110*   *10001110*

$l =$ 1 3 3 1

1	*0*	*1*	*0*

However, it does matter *which* *0*-block you choose to place a *0* inside of or next to. Thus, there are exactly k_0 unique ways of placing a *0* inside or next to the *0*-blocks in s.

Unlike with *0*-blocks, it *does* matter where a *0* is placed inside a *1*-block. For instance, these are unique placements within the same *1*-block in *1001110*:

10010110

$l =$ 1 2 1 1 2 1

1	*0*	*1*	*0*	*1*	*0*

is different from

10011010

$l =$ 1 2 2 1 1 1

1	*0*	*1*	*0*	*1*	*0*

For a *1*-block of length l_i^1, there are $l_i^1 - 1$ ways of placing a *0* inside of it. Again, it also matters which *1*-block you place a *0* inside of. Thus, there are exactly $\sum_{i=1}^{k_1}(l_i^1 - 1) = L_1 - k_1$ unique ways of placing a *0* inside the *1*-blocks in s.

Since we have considered placing a *0* next to or inside of each *0*-block, and all *1*-blocks are surrounded by *0*-blocks, we need not consider placing *0*'s next to *1*-blocks, with one exception. If a *1*-block occurs at the start of the string, we must consider placing a *0* to the left of it as a unique possibility, and if a *1*-block occurs at the end of the string, we must consider placing a *0* to the right of it as a unique possibility.

01001110

$l =$ 1 1 2 3 1

0 1 0 1 0

The number of additional ways of placing a *0* because of this is exactly equal to q, by construction.

Combining all of these possible placements, there are exactly $k_0 + L_1 - k_1 + q$ unique ways of placing a *0* in s. ■

We can use the same argument for the number of placements for a *1* in s to yield the following Lemma:

Lemma 3. *There are exactly $k_1 + L_0 - k_0 + p$ unique ways of inserting a 1 in s.*

We can combine these two Lemmas to count all possible ways to insert a character into s:

Lemma 4. *There are exactly $n + 1$ ways of placing a 0 or 1 in s.*

PROOF. By Lemmas 2 and 3, there are $k_0 + L_1 - k_1 + q$ unique ways of placing a *0* in s and $k_1 + L_0 - k_0 + p$ unique ways of placing a *1* in s. Summing these yields

$$\begin{aligned} k_0 + L_1 - k_1 + q + k_1 + L_0 - k_0 + p &= L_1 + q + L_0 + p \\ &= L_1 + L_0 + 2 \\ &= (n-1) + 2 \\ &= n + 1 \end{aligned}$$

■

Inserting a Second Character

Let $s = t = u$. Without loss of generality, add a character to t. We now ask how many ways we can add a character to u such that $s \in \text{LCS}(t, u)$.

Lemma 5. *There are exactly $n(n+1)$ ways of placing a 0 or a 1 in t and a 0 or a 1 in u such that their longest common subsequence is s.*

PROOF. By Lemma 4, there are exactly $n + 1$ unique strings that can be created by inserting a character into t. Since the LCS of two strings of length n can only be length n if the two strings are equal, any insertion into u will not affect the LCS length except for the insertion that results in the same string as t. Thus, there are $n + 1 - 1 = n$ ways of inserting the second character that result in a different string from t, so there are $n(n + 1)$ ways in total to add a character to both strings without modifying the LCS length. ■

Generalization

We now generalize this distribution property for any number of strings with an arbitrary alphabet:

Lemma 6. *Across all possible d-tuples of strings of length n and alphabet size $\sigma \geq 2$, every possible string of length $n-1$ over that same alphabet each appears as an LCS exactly $(\sigma n - n + 1)^d - (\sigma n - n + 1)$ times in total.*

Proof. First, let us consider the case where we have only two strings but an alphabet Σ of size $\sigma \geq 2$. We use the same definitions as in the binary case, as well as one new definition. In the binary case, a *0*-block can only be next to *1*-blocks or the ends of the string, meaning we could count all insertions at the edges of a *1*-block as part of our count for an insertion inside of a *0*-block. In the general case, this is no longer true, as we could, for example, insert a *0* in between a *1*-block and a *2*-block. To account for this difference, we define a ***seam*** as a position between two blocks or at the ends of the string.

Let a character $a \in \Sigma$ be given. As before, there is exactly one way of placing a inside any given a-block, so there are k_a unique ways of placing a inside all a-blocks. Further, a can be placed inside the blocks of some other character b in

$$\sum_{i=1}^{k_b}(l_i^b - 1) = L_b - k_b$$

ways. Thus, a can be placed inside the blocks of all other characters $c \in \Sigma$ in

$$\sum_{c \in \Sigma \setminus \{a\}} \left(\sum_{i=1}^{k_c} l_i^c - 1 \right) = \sum_{c \in \Sigma \setminus \{a\}} (L_c - k_c)$$

ways.

However, so far we have considered only placing inside blocks, but not next to them at their seams. There are

$$\sum_{c \in \Sigma}(k_c) + 1$$

seams in total (number of blocks + 1).

Placing a at a seam is a unique placement if the seam does not border an a-block. So the total ways of placing a character a at seams is

$$\sum_{c \in \Sigma}(k_c) + 1 - 2k_a$$

Each block borders exactly two seams, and since no blocks of the same character may

border each other, no overcounting occurs. Thus, for placing specifically a, there are

$$\begin{aligned} k_a + \sum_{c\in\Sigma\setminus\{a\}} (L_c - k_c) + \sum_{c\in\Sigma}(k_c) + 1 - 2k_a &= k_a + \sum_{c\in\Sigma\setminus\{a\}} (L_c - k_c) + \sum_{c\in\Sigma\setminus\{a\}} (k_c) + 1 - k_a \\ &= k_a + \sum_{c\in\Sigma\setminus\{a\}} (L_c) + 1 - k_a \\ &= \sum_{c\in\Sigma\setminus\{a\}} (L_c) + 1 \\ &= n - L_a \end{aligned}$$

ways. So, for placing *any* symbol, there are

$$\begin{aligned} \sum_{c\in\Sigma}(n - L_c) &= \sum_{c\in\Sigma}(n) - \sum_{c\in\Sigma}(L_c) \\ &= \sum_{c\in\Sigma}(n) - (n-1) \\ &= \sigma n - n + 1 \end{aligned}$$

ways.

As with the binary case, the second character inserted can be added anywhere so long as the insertion results in a different string from the first insertion. So there are

$$(\sigma n - n + 1)(\sigma n - n)$$

ways of placing a symbol each in two strings such that the LCS length remains the same.

To generalize this to $d \geq 2$ strings, the idea remains the same. The placement in the first string does not matter. The only restriction is that the placements in the following $d-1$ strings cannot all be the same as the placement in the first string. Thus, there are

$$(\sigma n - n + 1)((\sigma n - n + 1)^{d-1} - 1) = (\sigma n - n + 1)^d - (\sigma n - n + 1) \tag{B.1}$$

ways of doing these placements in total.

For each of the $(\sigma n - n + 1)$ unique strings created from inserting into the first string, there are $(\sigma n - n + 1)^{d-1}$ unique combinations for the other $d-1$ strings obtained by inserting a character into each of them. Since we do not want to count the combination where every resultant string is identical, we subtract that possibility from the count, resulting in Equation B.1. ■