algorithms

MDPI

Article

A New String Edit Distance and Applications

Taylor Petty [1,*], Jan Hannig [1,2], Tunde I. Huszar [3] and Hari Iyer [2]

[1] Department of Statistics and Operations Research, The University of North Carolina at Chapel Hill, Chapel Hill, NC 27599, USA; jan.hannig@unc.edu
[2] Information Technology Laboratory, Statistical Engineering Division, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA; hariharan.iyer@nist.gov
[3] Applied Genetics Group, Material Measurement Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA; tuende.huszar@nist.gov
* Correspondence: taylor.petty@unc.edu

Abstract: String edit distances have been used for decades in applications ranging from spelling correction and web search suggestions to DNA analysis. Most string edit distances are variations of the Levenshtein distance and consider only single-character edits. In forensic applications polymorphic genetic markers such as short tandem repeats (STRs) are used. At these repetitive motifs the DNA copying errors consist of more than just single base differences. More often the phenomenon of "stutter" is observed, where the number of repeated units differs (by whole units) from the template. To adapt the Levenshtein distance to be suitable for forensic applications where DNA sequence similarity is of interest, a generalized string edit distance is defined that accommodates the addition or deletion of whole motifs in addition to single-nucleotide edits. A dynamic programming implementation is developed for computing this distance between sequences. The novelty of this algorithm is in handling the complex interactions that arise between multiple- and single-character edits. Forensic examples illustrate the purpose and use of the Restricted Forensic Levenshtein (RFL) distance measure, but applications extend to sequence alignment and string similarity in other biological areas, as well as dynamic programming algorithms more broadly.

Keywords: string edit distance; Levenshtein distance; short tandem repeat; dynamic programming; massively parallel sequencing; next-generation sequencing; DNA identification

Citation: Petty, T.; Hannig, J.; Huszar, T.I.; Iyer, H. A New String Edit Distance and Applications. *Algorithms* **2022**, *15*, 242. https://doi.org/10.3390/a15070242

Academic Editor: Frank Werner

Received: 11 May 2022
Accepted: 11 July 2022
Published: 12 July 2022

1. Introduction

String similarity is an important tool for many applications, including web search completion suggestions [1], spelling correction [2], and ontology alignment [3]. Some of the tools developed in the field of string similarity carry over well to DNA applications, such as DNA-protein matching, sequence assembly, and local similarity searches [4], as well as Longest Common Substring and Longest Common Subsequence algorithms [5]. Determining similarity of DNA strands is important in phylogenetics, and the volume of available data continues to increase over time. Alignment-free methods have been developed, in part to account for issues that arise when traditional sequence alignment is pushed to its limit. These include the use of geometric representations in various dimensions, as well as graph theory [6].

Most string edit distances consider only single character edits and are variations of Levenshtein distance. These standard string edit distances are inadequate for analyzing similarity between DNA sequences in forensic applications where the goal is to identify the individual who is the source of the DNA in a crime sample. To fill this gap, a new string edit distance is defined, referred to as Restricted Forensic Levenshtein (RFL) distance, that accommodates the addition or deletion of one or more motifs (consecutively repeating sequences of nucleotides 2 to 6 base pairs long) as edit types separate from single-nucleotide insertion, deletion, and substitution. This makes the problem of computing the distance

much more difficult. To address this, a dynamic programming algorithm for computing the RFL distance between sequences has been developed. Forensic examples are used to illustrate the purpose and use of RFL distance but the applications extend to sequence alignment and string similarity in other biological applications. The algorithm's novel contribution is in efficiently handling the complex interactions that arise between multiple- and single-character edits.

1.1. Background

DNA-based human identification is an extremely powerful tool with many important applications such as identification of perpetrators of crimes, disaster victim identification, DNA typing of skeletal remains, and relationship testing [7–9]. The same methods are also applicable in food authenticity, poaching, and counter-bioterrorism [10]. DNA measurement technologies have made great strides recently, and now even minute quantities of DNA can be reliably measured [10].

In forensic DNA testing, genetic material present at a crime scene is extracted and amplified through the polymerase chain reaction (PCR). In principle, this technology replicates specific regions of the human genome present in the sample by a repeated process of doubling. Errors may get introduced to the products during the PCR process, and these error sequences are called *artifacts*. Traditionally, analysis of the amplified DNA products is carried out via a length-based measurement using capillary electrophoresis (CE); however, with the introduction of sequencing to the workflow, the amplified products now enter the process of massively parallel sequencing (MPS) providing a sequencing-based measurement for an allele. After the sequencing is complete, the generated data can be used as an input for bioinformatics pipelines. The data analysis provides details about the measured quantity of the expected PCR product sequences (here referred to as *parent* alleles) and numerous artifact sequences related to the range of genetic markers examined in these assays. In typing a standard single-source sample for a set of autosomal markers, the sequences observed should include one to two different parent alleles (depending on whether the sample was homozygous or heterozygous at the autosomal locus) and numerous artifact sequences that are present because of copying errors in the amplification and sequencing process. Each detected sequence is quantified by the number of sequences observed and is commonly referred to as the *depth of coverage* (DoC). Along with the parent alleles there are artifacts that are also detected and are often present at a low DoC, and therefore easily distinguishable from the parent alleles. Each of the artifacts is an error sequence of a true parent, and one can generally infer which parent it came from, but occasionally the parent sequences are similar enough that it is difficult to tell which is the actual parent sequence. For single-contributor DNA samples, this is typically not an issue, but crime scene samples motivating this work are rarely from a single individual. Crime scene samples often consist of multiple individuals, contributing different amounts of genetic information. For example, in a complex mixture (e.g., three contributors at 70%–25%–5%) it can be challenging to determine the respective parent allele of each contributor and their corresponding artifact(s).

In this light, quantifying what it means for nucleotide sequences to be similar becomes a crucial task, which has led to the notion of a generalized Levenshtein distance that allows gains and losses of multiple letters at once (here the change of multiple bases at once in the sequence string), and the development of an algorithm to calculate it. A major contribution of the proposed algorithm is in handling the exponential explosion of complexity when multiple-character edits and single-character edits interact, which requires careful techniques to resolve.

1.2. Short Tandem Repeats

The DNA identification community commonly uses short tandem repeats (STRs) as their genetic marker of choice [11,12]. STR typing methods are accurate and sensitive [13]. The work in this paper is designed for the analysis of DNA samples in forensic applications,

but the methods presented here can be applied in other areas that use STR analysis. An STR is a region of DNA that contains consecutively repeating motifs. To the left and right of this repeating region is a non-repeating section of DNA called a *flanking region*. For example:

ACTCC ATG ATG ATG ATG GGTTCTGA

Here the motif ATG is repeated four times, which is represented in a short format as [ATG]4.

As sequencing technology has developed, individual nucleotides within STR regions in the human genome can be measured increasingly economically and accurately, whereas in the past people were limited to measuring the total length of the targeted sequence. The standard length-based CE methods do not allow for the detection of intra-motif variations. For a real-data example, consider a sample with a two-allele locus with repeat regions

A: [TCTA]8 [TCTG]1 [TCTA]1

and

B: [TCTA]10.

Note A and B are identical by length. When both alleles at a locus share the same length but differ by sequence, they are known as *isometric heterozygotes*, and CE methods cannot discriminate between them. CE types both of these alleles as 10, an allele with a length of ten repeats, and classifies this sample as homozygous at this locus. By recognizing variation at the sequence level, the power to discriminate between individuals increases [14]. If an artifact sequence C = [TCTA]7 [TCTG]1 [TCTA]1 was detected in the sequencing output, it would be reasonable to infer that there is a higher likelihood of it being an error derived from A rather than B, since fewer changes are required to edit A into C than B into C. Consequently, it would make sense to infer that the artifact is related to A. With CE, A and B are viewed as identical, and no such conclusions about the artifact can be made. As crime scene samples are most often mixtures, the additional resolution of the alleles and artifacts provided by the use of sequencing can be advantageous compared to solely length-based analysis.

In STR regions, motifs often expand and contract as a unit. This phenomenon of slippage of the polymerase on the template is referred to as *stutter*. The artifacts of in vitro stutter products are consistently present in measurable quantity and therefore are accounted for in routine forensic DNA analysis [15]. Furthermore, stutter occurs in vivo in STR regions even at conservative estimates at least 3–4 orders of magnitude more often than other random point mutations [16]. To reflect this, a generalization of string edit distance was used to capture similarity between sequences in MPS output. The goal was to include the addition or deletion of a motif as an edit type with a cost that can vary independently from single-nucleotide edits, to reflect the fact that these motif losses and expansions (called *backward* and *forward* stutter, respectively) occur consistently across various in vitro applications [17]. The proposed algorithm is also applicable to in vivo methods such as rare disease diagnosis and cancer progression tracking that use STR markers including their stutter [16].

It is desirable to develop a distance that gives high-frequency artifacts a lower distance to the true alleles than low-frequency artifacts, so it is important to capture both reverse and forward stutter as low-cost events. Standard string edit distances such as the Levenshtein distance modify one character at a time, so the cost of dropping or adding a multi-character motif is necessarily equal to the sum of the costs of dropping or adding the individual characters. Furthermore, unique motifs can stutter within a single STR region, with each motif having a potentially unique cost. Beyond the issue of edit cost, stutters and single-character edits can interact in complicated ways, and algorithms like the Levenshtein distance and its relatives do not begin to address the issues that arise in that interaction.

String edit distances have been in use for decades, with many papers published on different applications. Review of the literature revealed diverse uses, but multi-character

extensions of edit distances were rarer. One paper described a spelling correction algorithm using a generalized metric that allows for generic word-to-word edits [18]. The aim and methods described in the study are different than those presented here; however, if the code and details for implementation were included in [18], direct comparison with the independent method presented here would be of interest. Another paper working with multi-character edits introduces the some theoretical ideas relevant to dictionary based searches and generalizations of the Damerau-Levenshtein distance [19], but their searches are approximate and require symmetry, which cannot be used here since insertions and deletions of both characters and motifs happen with different frequencies. There is a variation on the dynamic programming approach that allows for generalized edits from one substring to another with non-symmetric costs, but it also prohibits multi- and single-character edit interactions, and no code or software was available for comparative testing [20]. Generalized edit distances were infrequently mentioned in the literature, and attempts to find comparable algorithms described with code attached were unsuccessful. The work presented in this paper uses an algorithm readily available for public use that also allows for a large set of overlapping edits.

The ideas behind the RFL distance are applicable to any situation requiring a minimal edit distance solved via dynamic programming, notable among which is graph edit distance [21]. By extending the idea of precomputed motifs to other dynamic programming settings, one could consider a minimal graph distance that allows the addition or loss of an entire subgraph as its own edit to have an individual cost. Therefore, this idea could have implications wherever graph edit distances are used, such as handwriting recognition [22], fingerprint recognition [23], and cheminformatics [24].

The proposed algorithm to compute RFL distance has fully customizable costs, and is therefore capable of reflecting the idea that lower costs correspond to higher probability edits. This connects to the notion that more frequent events would be favored in a maximum likelihood approach over lower probability events. The Python code for the RFL algorithm is available on GitHub [25].

2. Restricted Forensic Levenshtein Distance

The proposed algorithm handles both single- and multiple-character edits, as well as their interactions. The single-character part of the algorithm is built on the original Levenshtein distance, which will be reviewed below. An introduction of the RFL distance follows.

2.1. Levenshtein Distance Overview

String edit distances are based on the minimal number of operations required to transform one string into another. This gives a way to measure dissimilarity of objects in the non-Euclidean space of strings, using the notion that two strings that are similar should differ in only a few characters, while two strings that are dissimilar would require many changes to turn one into the other. The edit distance proposed here could be viewed as a generalization of the Levenshtein distance [26], which counts the minimal number of single-character deletions, insertions, and substitutions required to transform one string into another. For example, the distance from CAT to CGT is one, from CAT to CA is one, and from CAT to CATTG is two. Other edit distances exist, with different edits allowed. Damerau-Levenshtein considers indels, substitutions, and transpositions [27], while the longest common subsequence metric only looks at indels [28]. Rather than having a common cost of 1, substitution, insertion, deletions, etc. can each have their own unique cost [29].

The calculation of minimal edit distance is typically done via dynamic programming [30]. The standard dynamic programming solution to Levenshtein will be briefly discussed below because it is crucial to understanding the novel contributions of the proposed algorithm. Examples will be presented to maximize clarity.

2.2. Forensic Distance Overview

The STR regions of DNA that are currently used for forensic identification were historically selected based on the polymorphic length variations of these markers observed in the population. Most of these are tetranucleotide repeats, but di-, tri-, penta- or hexanucleotide repeats also exist in the tandem oriented arrays. At sequence level a trinucleotide marker can be described as a string such as ACTACTACTACT, or abbreviated, as in [ACT]4. Backward stutter refers to losing a motif, or [ACT]3 in this example. Less-common forward stutter refers to gaining a motif, or [ACT]5 here. There are other possibilities, such as substitution, e.g., [ACT]4 to changing into an [ACT]2 AGT [ACT]1 via a C $\rightarrow$ G substitution, but the individual single-character edits are significantly rarer than motif stutter, and are already taken into account under the classical Levenshtein distance.

Stutter is more common than insertion or deletion, so it is modeled as a separate edit type. This enables the cost of stutter in the edit distance to be lowered to reflect its higher probability, instead of modeling it as the sum of individual nucleotide changes. In order to accomplish this, the RFL algorithm includes a step of looking at a set of substrings of various lengths with their associated costs from a precomputed dictionary. Whereas the standard dynamic programming solution looks back a single letter, the proposed approach looks back several letters. The particular dictionary of substrings is what gives the RFL algorithm the title "restricted", since the size of the dictionary limits how far back the algorithm looks.

A naive approach would be to count the difference in the number of motifs and use the classical Levenshtein distance of the flanking regions, but this method misses vast numbers of edit interactions, and further misinterprets single-character edits in the STR region as stutter, e.g., [ACT]4 editing to [ACT]3 AGT would be qualified under a naive approach as back stutter, since there is one fewer ACT motif, but inspection clearly reveals a C$\rightarrow$ G substitution. The complexity of the solution developed in this paper comes from the correct distinction of single-character edits and stutter, and the interaction of multi- and single-character edits. For example, the sequence [ACT]4 could change to [ACT]5 via forward stutter, then a C could be deleted, for a final result of [ACT]4 AT. This final string could have been attained by inserting the letters A and T from the original [ACT]4, but another option was the forward stutter and single nucleotide deletion. Different edits contribute different (customizable) costs, so one route may be more optimal than the other depending on the parameters. Extending these interactions across multiple edits becomes exponentially complex, as motifs can stutter forward and then characters can insert into the middle of that motif, for example, or a SNP could occur which turns a substring into a motif that can then stutter back, and neither of these would be detected by a naive approach. For example, if ACT were the motif, detecting the string GACGT as a three-edit insertion (one forward stutter, two single-character insertions) instead of five, or detecting TCT as a two-edit deletion instead of three, is a powerful capability, especially when considering that each edit can have vastly different costs.

The RFL algorithm handles insertions, deletions, and substitutions, the addition and deletion of prespecified character motifs, and the overlapped interactions of the multi- and single-character edit types. A comparison between classical Levenshtein, the naive approach, and the RFL algorithm is shown in Table 1.

Theoretical complexities arise when considering these edit interactions, as addressed in Example 1, Section 2.3.3, and Appendix A.1. These complexities are unbounded, so it is necessary to restrict the search space. However, the flexible parameter restrictions are required to accurately model the underlying sequence behavior.

A data-driven approach requires knowing which motifs at each locus stutter at significant rates. For the design of the RFL algorithm, sequence analysis data from 22 autosomal STRs was used from a set of 661 samples from four U.S. populations. The sequence data was previously generated using the PowerSeq Auto/Y System kit (Promega Corp., Madison, WI, USA) and was analyzed using STRait Razor v3.0 [31]. The output files were used

to identify motifs of length 2–6 that were the basis of observed stutter artifacts at each autosomal locus. This analysis is described in Section 3.1.

Table 1. Edits Accounted for in String Edit Distances.

Operations Accounted for	Classical Levenshtein	Naive Forensic	RFL
Single-character edits	✓	✓	✓
Pure motif stutter		✓	✓
Distinguishing stutter and single-character edits			✓
Multi- and single-character edit interactions			✓

2.3. Details of the Rfl Algorithm

The foundation of the RFL algorithm is built on the Levenshtein distance. In this section, a perspective on the standard dynamic programming algorithm for classical Levenshtein is discussed, then Weighted Levenshtein is discussed, and lastly the novel modifications are presented. Zero-based indexing for strings will be used, with $s[i:j]$ to represent the substring of s from the i-th to the $(j-1)$th character, inclusive. Similarly, $s[i:]$ will represent the substring composed of the ith character up to and including the last character, and $s[:j]$ will represent the first j characters, up to index $j-1$, inclusive.

2.3.1. A Perspective on Dynamic Programming

From this point on, a working knowledge of the Levenshtein distance is assumed, with the following perspective. To compute the distance between strings a and b with unit edit costs, the base case is that one of the strings is empty. If a is the empty string then insert each character of b, and the number of those insertions is the cost. If b is the empty string, delete each character of a, and the number of those deletions is the cost. If the first characters of a and b are the same, then delete $a[0]$ and $b[0]$ for no cost and the overall distance will be the same as the Levenshtein distance between the remainder of the strings $a[1:]$ and $b[1:]$ (denoted as tail(a) and tail(b), respectively).

In case the first characters of a and b do not match, the standard dynamic programming algorithm for Levenshtein distance effectively aligns the leading character of a with the leading character of b, then modifies the leading character of a to match the leading character of b. Once the leading characters of the strings match, the matching leading characters can be deleted for free.

For example, in order to compute the Levenshtein distance from AGTCT to GACT, the A must be deleted and the first T switched to an A, for a cost of two. This eyeball calculation only works for simple examples, so Figure 1 illustrates this perspective on the classical Levenshtein algorithm.

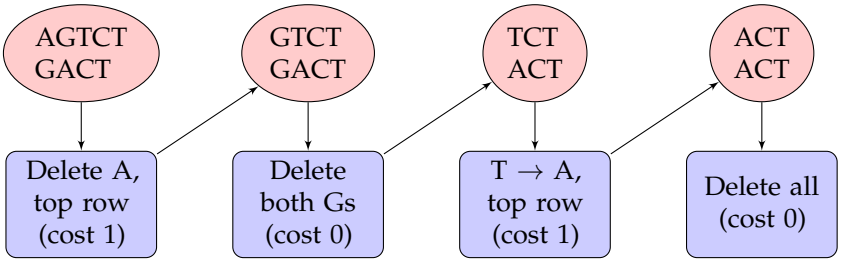


Figure 1. The steps to compute the Levenshtein distance from AGTCT to GACT.

Note that in Figure 1 there could have been an additional T concatenated to the beginning of AGTCT and then immediately deleted. This would have been a valid edit,

but not a minimal one. At each step the minimal cost must be taken, and this will be accomplished via dynamic programming. For a n_a-character string a and an n_b-character string b, the dynamic programming algorithm constructs an $(n_a + 1) \times (n_b + 1)$ matrix, and overall it is $O(n_a n_b)$ in time complexity.

2.3.2. A Note on Weighted Edit Distance

In a setting where different edits happen with different frequencies, one might want to weight edits differently from each other. This could be because DNA exhibits higher rates of changing a particular character to another, or in spelling correction because certain letters are closer to others on the keyboard. For example, inserting an A could be penalized more than any other letter, and substituting C for the letter G could be incentivized. This could be done by raising the insertion cost of an A to 2 and lowering the substitution cost of G $\rightarrow$ C to 0.5. The edit distance algorithm remains the same, but whenever those edits are performed, the cost is different. This might change the optimal edit path from one string to another. Normally, converting CAT into TGT would be two substitutions: C $\rightarrow$ T and A $\rightarrow$ G. In the case where all substitution costs are equal to 4 and insertion and deletion costs are still equal to 1, the previous edit path involving two substitutions would cost a total of 8, and a new minimal-cost path would be CAT $\rightarrow$ AT $\rightarrow$ T $\rightarrow$ GT $\rightarrow$ TGT, which would cost 4. This could be made much more complicated by changing every possible edit cost to be different than every other. The dynamic programming algorithm solution is able to take any possible assortment of costs and compute the minimal cost given those parameters.

This can easily make the distance non-commutative, i.e., it is a directed distance. In a case where the substitution cost of C $\rightarrow$ G is 3, but all other costs remain equal to 1, then the distance from CAT to GAT is 2 (delete C, insert G; this is still less costly than changing C to G directly), but the distance from GAT to CAT is 1 (changing G to C still costs 1). Therefore the distance hereafter is referred to as being from a sequence s to another sequence t, which represents that the edits begin at s and change into t. A library for the weighted Levenshtein, optimized in Cython, is available [32].

2.3.3. Restricted Forensic Levenshtein Distance

Here the full steps of the RFL algorithm are presented. To compute the distance between strings a and b, if a is the empty string, simply insert each character of b, and the number of those insertions is the cost. If b is the empty string, delete each character of a, and the number of those deletions is the cost. If the first characters of a and b are the same, then delete $a[0]$ and $b[0]$ for no cost and the overall distance will be the same as the Levenshtein distance between the remainder of the strings $a[1:]$ and $b[1:]$ (denoted as tail(a) and tail(b), respectively).

In case the first characters of a and b do not match, line the two strings on top of each other with $a[0]$ aligned to $b[0]$, with all edits being made to the parent string a (this becomes important later, when the distance becomes asymmetric). The goal of the RFL algorithm is to make the leading characters of the strings match, in which case the matching leading characters can be deleted *for free.* For instance, the distance from CAT to CGT is the same as from AT to GT, since the leading C does not require editing. Returning to the aligned strings a and b, there are three options. The first is that the leading character of a can be changed into the leading character of b, which corresponds to substitution errors in PCR. A new leading character of a can be inserted to match the leading character of b, which corresponds to insertion. Lastly, the leading character of a can be deleted so that the new leading character of a matches the leading character of b, which corresponds to deletion.

The goal is to be able to include the deletion or insertion of multiple consecutive letters as a separate edit type at a reduced cost. The classical Levenshtein solution is capable of gaining or losing multiple letters, but the cost will always be the sum total of the costs of the necessary single-character interim operations.

The classical Levenshtein algorithm looks at a character and determines what operation at that position gives the lowest cost to change a single letter at that spot when

changing one string into another. It does this iteratively, breaking the original problem into sub-problems by addressing a single letter at a time. The RFL modification further looks at multiple characters at once. This will correspond to looking back several positions within the dynamic programming matrix. Only particular substrings will be allowed as their own edit types, as opposed to any general substring. The RFL solution is to create a dictionary of multi-character substrings and their associated insertion and deletion costs. When a substring—for example CTG—needs to be inserted or deleted, the relevant cost associated with that substring in the dictionary is retrieved, and that option is included for the algorithm to consider when taking a minimum cost step at that index. This substring CTG corresponds to looking back three steps in the dynamic program.

For the sake of illustration, assume unit costs for all edits, and consider AAAGA, the motif at the Penta D locus. If AAAGA is permitted to be added or deleted all at once, then inserting AAAA will be cheaper if AAAGA is first built and then the G deleted. Inserting AAA would cost the same either way: inserting 3 As, or inserting AAAGA (+1) and then deleting an A (+1) and deleting a G (+1). Inserting AAAGTA would cost 2 (+1 to stutter and +1 to insert), as opposed to costing 6 by inserting one at a time. The RFL algorithm needs to account for this, or else it would give the string AAAGTA a cost of 6 because the motif AAAGA is "broken" by the T.

The question naturally arises: if costs are being precomputed for different substrings, which substrings should be included? Suppose that for a specific motif (e.g., AAAGA), all possible substrings and their associated costs were included. Note that *any* string with the letters AAAGA (in that order) would be cheaper to insert by first stuttering forward and then adding letters in between, instead of adding them one by one. This is shown in Example 1 to motivate the need for restricting the search space. The resolution of this example after the restrictions are in place will be shown in Example 3.

Example 1. *To build the string*

$$A(TTTT)A(TTTT)A(TTTT)G(TTTT)A$$

using edit operations starting with an empty string, 21 letters would need to be inserted, for a cost of 21. This would cost 4 less overall by inserting the motif AAAGA first via stutter (cost of 1) and then inserting all the remaining nucleotides in between (cost of 16), for a total cost of 17. This is true for an unboundedly long string, with any number of characters in between the motif characters as long as the letters AAAGA are present in that order.

Even if a string had the letters A...A...A...G...A with 2000 nucleotides in between each letter, by including it in the dictionary, the algorithm would qualify it as a modified stutter motif. Thus, not only is an infinite (or virtually infinite) dictionary impossible, it is undesirable and potentially absurd. In addition, there is no evidence to suggest that PCR results in such drastic numbers of errors. The faithfulness of the polymerase used in PCRs for STR typing has improved enough that these only err sporadically, which is supported by the fact that only small deviations were observed in artifacts in this study. Such assignment of stutter cost over excessive length of the string is undesirable, therefore a limit to the extent of the modification was incorporated.

The appendix contains more detailed discussion regarding the complex cases that arise when considering this additional edit type, but a few examples are included here. As mentioned previously, it is possible to have intra-motif substitutions and indels. The STR [ACT]4 could give rise to [ACT]2 AT [ACT]1. The difference in the number of ATG motifs here is $4 - 3 = 1$, but there was no stutter—just deletion of the letter C. Including the interaction of stutter with other edit types quickly explodes complexity in the minimal cost computation.

Thus, for every possible nucleotide sequence of length $\leq 2k - 1$, where k is the length of the motif, the classical, weighted Levenshtein distances of both inserting these strings from scratch or deleting them are computed, with the requirement that the edit path of that substring contains exactly one stutter. Specifically, the precomputed insertions contain

short sequences of letters with an associated cost, assuming the edit path begins with a motif insertion. The precomputed deletions contain the same strings as the insertion list, but with the cost of deleting, assuming the edit path ends with a motif deletion. These data are stored in a pair of dictionaries. In the GitHub resource, this function is called `lsdp`, an acronym standing for Levenshtein Stutter Dictionary Pair. Specifics are illustrated in Example 2.

Example 2. *For the motif TCTA, the stutter dictionary d has two keys: 'insert cost' and 'delete cost'. Each of these keys gives a dictionary d[key] of substrings and their associated costs. A selection is given as follows, where all edit costs equal to one—note that the insert and delete dictionaries will always contain the same entries, but different entries are shown here to demonstrate different costs:*

$$
\begin{aligned}
d = \{&\textit{insert cost:}\\
&\quad \{\textit{TCTA:}\ 1,\\
&\quad \textit{TCTAT:}\ 2,\\
&\quad \textit{A:}\ 4,\\
&\quad \textit{GGGGGGG:}\ 8,\\
&\quad \ldots\},\\
&\textit{delete cost:}\\
&\quad \{\textit{TCTATGG:}\ 4,\\
&\quad \textit{TCT:}\ 2,\\
&\quad \textit{TCTA:}\ 1,\\
&\quad \ldots\}\\
\}&
\end{aligned}
$$

Note that strings like 'GGGGGGG' and 'A' cost more to insert in this dictionary than they do in classical Levenshtein, since the RFL algorithm first inserts a motif for a cost of 1 before modifying it into the desired string. A symmetric fact holds for deletion entries. However, since RFL includes classical Levenshtein operations and takes the lowest-cost path, inserting the string 'A' will be considered both as a cost 1 edit (from the classical Levenshtein solution) and a cost 4 edit (from the dictionary), with the lower cost option being taken.

In Example 1, a sequence was shown to demonstrate that an unbounded search space would not be desirable even if it was possible. Example 3 explains how the restriction affects the answer on that use case.

Example 3. *In Example 1, it was illustrated that the string*

$$A(TTTT)A(TTTT)A(TTTT)G(TTTT)A$$

would cost 21 to insert from a blank string if no motifs were considered, but it would cost 17 if the cost dictionary for the motif AAAGA contained strings of arbitrary length. The length of the motif AAAGA leads to a bound of $2 \times 5 - 1 = 9$ *on the dictionary substrings. The RFL algorithm with this bound in place will see parts of the parent string and insert them from the dictionary when it is cheaper to do so. For example, the edit path from a blank string to the motif AAAGA costs 1 via forward stutter and the edit distance from AAAGA to the first six characters of the string, ATTTTA, costs 4 (three substitutions and one insertion), so to insert the first six characters of the string totals 5, whereas inserting each letter individually costs 6. As the RFL algorithm moves along the string, the next place it becomes cheaper to use the dictionary is when inserting ATTTTGT, which would cost 7 with single-character edits, but instead costs 6 via the dictionary (since AAAGA* $\rightarrow$ *ATTTTGT requires three substitutions and an insertion). There are no other places where the RFL algorithm finds a more optimal route. Although an unbounded search space would give a cost*

of 17 and a classical Levenshtein would give a cost of 21, the RFL algorithm with the $2k-1$ *bound gives a cost of 19.*

It must be noted that the sequence in Example 3 was given as an extreme example. Motifs stutter in the data both forward or backward with notable frequency, with vastly fewer single-character edits. It would be vanishingly rare to see as many insertions as this extreme example demonstrates. Although it is a single straightforward step to change the dictionary to include more substrings, doing so will not reflect the true relative frequency of stutter versus single-character insertions and deletions. Each dictionary key accounts for exactly one stutter, whether forward or in reverse. Once substrings of length $2k$ or more are being considered, it is much more likely that two stutters are at play instead of one, and the case of two stutters is already well-accounted for in the RFL implementation as it stands, since each motif will get treated separately. Rare, more complex theoretical exceptions are described in Appendix A.1.

In the RFL algorithm, all edits are fully customizable. Every edit from one nucleotide to another can be set with a unique cost, and every relevant motif can stutter forward and backward with unique costs.

The classical Levenshtein distance separately considers the cost of an insertion, deletion, or substitution, and takes the minimum of the three at each step to expand the matrix via dynamic programming. To account for all the substrings accessible via stutter, the RFL algorithm considers the same possibilities as classical Levenshtein, and additionally considers at each step the options of inserting or deleting any l-character portion, where l ranges from 1 to $2k-1$, and where the respective cost is defined by the pre-built dictionary.

2.3.4. How Restricted to Be?

Stopping at string length $2k-1$ is a parameter easily changed by generating a longer dictionary via the `lsdp` function. The RFL algorithm intakes the dictionary and can look back as far as the dictionary allows. Previously discussed in Section 2.3.3 was the motivation to consider only motifs that are slightly modified. Insertions were so rare in the data analysed in this study that by the time an additional k individual nucleotides have been inserted into a motif of length k, it is desirable for the proposed algorithm to be looking at a potential pair of motifs instead of one highly mutated motif. If an STR DNA sequence is gaining or losing $2k$ characters at once, it is more likely due to two separate stutters than to a single stutter that was then affected by an additional k inserts or deletes. Furthermore, in this data set, stutter and substitutions are much more common than deletions, and all are more common than insertions, giving more reason to cap the string length.

These lists of strings and their associated costs are computationally intensive to create (although still generally under a minute on a personal laptop), but once one is made (stored as a dictionary in Python for fast lookups) it can be used any number of times, as long as the motif and costs stay the same. Expanding this dictionary for completeness could be another area for further research.

2.3.5. Multiple Motifs and Time Complexity

Some forensic DNA loci have multiple motifs (e.g., [ACT]5 [AGGT]12). A pair of stutter dictionaries is computed for every motif, and at each step in the RFL algorithm the dictionaries of each motif are checked. This allows for any number and combination of preset motifs to be included in the calculation, each with fast dictionary lookups.

The proposed algorithm allows any prespecified set of motifs. For every additional motif, two inner for-loops are added at each step in the dynamic programming matrix to check the cost of inserting and deleting anywhere from 1 to $2k_i-1$ characters from the dictionary associated with that motif of length k_i. Because there are two for-loops running through the list of all motifs, and within each of them there is another for-loop running through 1 to $2k_i-1$ characters for motif i, the algorithm grows in complexity by those factors. Where the classical Levenshtein distance is $O(n_a n_b)$ for strings of length n_a and n_b,

the RFL distance theoretically requires $O(n_a n_b mk)$ time, where m is the number of motifs and k is the length of the largest motif $\max(k_i)$, assuming $O(1)$ dictionary lookup speed. However, in applications with standard autosomal STRs, motifs are all of length at most 5 ($2k_i - 1 \leq 9$), so this algorithm can be considered as $O(n_a n_b m)$, growing with the number of motifs considered as well as the lengths of the strings. Additionally, standard autosomal STRs have no more than a handful of motifs to be considered—a maximum of three, in this work—so the asymptotic growth becomes $O(n_a n_b)$ once more.

2.4. Implementation of the RFL Distance

Consider two strings, a and b, where $|a| = m$ and $|b| = n$, with zero-based indexing used throughout. The goal of the RFL algorithm is to construct an $(m+1) \times (n+1)$ matrix d, where each $d[i, j]$ is equal to the distance from the first i characters of a to the first j characters of b, resulting in the output $d[m, n]$. Recall that the distance is not commutative, so "parent" refers to a and "child" refers to b (since the string b originally began at a and went through edits to turn into b).

Throughout this section, recall the mechanics behind Example 1. The first characters of the parent and child are aligned, and the algorithm then modifies the parent. With each edit the parent is modified to make the leading characters of both strings match, which characters can then be removed for free, and thus the modifications eventually lead to a pair of empty strings and the cost of each step tallied.

2.4.1. Building the First Row

The cost of editing an empty string into an empty string is 0, so $d[0, 0]$ is always 0. The element $d[0, 1]$ is the cost of editing an empty string into the first character of the child string b. A list of cost options at this step will be made and then the minimal cost option taken. Obtaining the first character can either be done by inserting it directly or by stuttering forward a motif and then deleting characters back. For example, if the first character of the string b was G, and the motifs being considered were GGC and ATTC, then the following costs would be appended to the options list:

- The insert cost of the character G
- The cost of stuttering forward GGC plus the cost of the weighted Levenshtein distance from GGC to G.
- The cost of stuttering forward ATTC plus the cost of the weighted Levenshtein distance from ATTC to G.

Each motif can have a different stutter cost associated with it that can vary across loci, and in forensic applications forward stutter will be more expensive than backward since it is a rarer artifact. In this example, GGC might cost 3 to stutter forward and 1 to stutter backward, while ATTC might cost 2 to stutter forward and 0.5 to stutter backward.

The quantity $d[0, 1]$ is the minimum of the computed options, and the algorithm moves forward to $d[0, 2]$. Now there are several possibilities:

- $d[0, 1]$ + the cost of inserting the second character via pure insertion (i.e., the cost of inserting $b[1]$, using 0-based indexing)
- $d[0, 1]$ + the cost of inserting the second character via the stutter dictionary (for each stutter dictionary)
- $d[0, 0]$ + the cost of inserting the first two characters via the stutter dictionary (for each stutter dictionary)

Furthermore, in general, for each motif, if the motif lengths are k_i, for every length j less than or equal to $2k_i - 1$, substrings of characters from j positions back to the current can be checked, and if less costly those characters can be inserted via the stutter dictionary. It is impossible to look back further than the beginning of the string, however.

To construct the first row $d[0, :]$, pseudocode outlined in Listing 1 is followed. Building the first column $d[:, 0]$ follows similarly.

Listing 1. Building the first row of the dynamic programming matrix in the RFL algorithm.

```
parent_length = length of parent
child_length = length of child
k = length of longest motif being considered
lookback = 2k - 1
d = matrix of zeros with parent_length + 1 rows and child_length + 1 columns

for i from 1 to child_length:

# find d[0, i], the cost of modifying a blank string into the first i characters of the child string
# d[0, 0] is always 0, the cost of editing an empty string into an empty string

options = []

# d[0, i - 1] is the cost of inserting the first i - 1 characters of the child string
options.append(d[0, i - 1] + cost of inserting character i)

# look back 1, 2, ..., and 2k - 1 characters in the string (or back to the first character, whichever comes first)
for j from 1 to min(lookback, i):

# find the cost of inserting the last j characters up to and including character i, assuming a forward stutter happened first

S = the j characters immediately preceding and including character j of the child string

for every motif being considered:

# if there are motifs of unequal size, there will be substrings in the dictionary of the larger motif that are out of reach of the smaller motif, since the dictionary for a motif of length L only includes strings of length 2L - 1

if S is not too long for the forward-stutter dictionary for this motif:

options.append(d[0, i - j] + cost of inserting S from dictionary)

# the cost at this location is the smallest cost out of the options considered
d[0,i] = minimum of the options list
```

2.4.2. Filling Out the Matrix

This step combines the complexity of building the first row and column. At each point $d[i, j]$, the algorithm looks back at the previous entries $d[i-1, j]$, $d[i, j-1]$, and $d[i-1, j-1]$ and adds the costs of deletion, insertion, and substitution, respectively. Additionally, for each motif length, it looks back at $d[i-r, j]$ and $d[i, j-r]$, for reverse and forward stutter, respectively, for r from 1 to $2k_i - 1$ for each motif i of length k_i. After considering all these costs, the entry $d[i, j]$ is the minimum.

In Listing 2, $d[m-1, n-1]$ is the distance from the first $m-1$ characters of the parent to the first $n-1$ characters of the child. Because of zero-based indexing, parent$[m-1]$ is the m-th character of the parent.

To demonstrate the RFL algorithm in depth, a fully worked example is given in Appendix A.2.

2.4.3. Practical Implementation

The RFL algorithm is written in Python, compatible with NumPy and therefore Numba, which increases the speed over the same algorithm in pure Python by a factor of about 30 via an `@njit` wrapper. However, the Numba-optimized version of the algorithm only works if including the possibility of stuttering a single motif, due to limitations in the implementation with Numba's inherent dictionary types. Based on testing of the `weighted-levenshtein` package (written in Cython) [32], the RFL algorithm has potential for significant increases in speed. In preliminary testing, modifying the data structures to integer-coded arrays and rearranging the way the cost dictionaries are stored can accelerate the computations by more than an order of magnitude via Numba. Furthermore, preliminary testing of a Numba prototype for the base Levenshtein algorithm suggests that in the case of DNA, Numba can meet or exceed the speed of the generic Cython package.

Listing 2. Building the dynamic programming matrix in the RFL algorithm.

```
for every row m from 1 to parent_length:
for every column n from 1 to the child_length:

options = []

# substitution: look up and to the left a single step
options.append(d[m - 1, n - 1] + cost of changing character m of the parent into
    character n of the child)

# insertion: look back one column
options.append(d[m, n - 1] + cost of inserting character n of the child)

# deletion: look up one row
options.append(d[m - 1, n] + cost of deleting character m of the parent)

# forward stutter: look back 1, ..., 2k - 1 columns, or back to the first column
    if 2k - 1 steps would be out of range
for i in 1, ..., min(lookback, n):

S = the i characters immediately preceding and including character n of the child
    string

for every motif being considered:

if S is not too long for the forward-stutter dictionary for this motif:

options.append(d[m, n - i] + cost of inserting S (from forward-stutter dictionary)
    )

# back stutter: look back 1, ..., 2k - 1 rows, or back to the first row if 2k - 1
    steps would be out of range
for i in 1, ..., min(lookahead, m):

S = the i characters immediately preceding and including character m of the parent
     string

for every motif being considered:

if S is not too long for the back-stutter dictionary for this motif:

options.append(d[m - i, n] + cost of deleting S from back-stutter dictionary

# the distance from the first m characters of the parent to the first n characters
     of the child is attained by taking the smallest of the options at this step
d[m,n] = min(options)
```

2.5. Validation of the Algorithm

The ideas in Section 4 of [20] can be adapted to provide theoretical justification for the RFL distance. Ukkonen's proof is for when all of the string edits can be performed in parallel, i.e., no edit interactions such as a forward stutter followed by a substitution within the stuttered motif (e.g., '' → AATT → AACT). The RFL distance is specifically designed to address edit interactions that *cannot* be performed in parallel. However, with the `lsdp` cost dictionaries in place, the larger family of dictionary edits can once again be performed in parallel because the edit interactions are already built in, and Ukkonen's justification holds, although their implementation is different and no code was available for comparison testing.

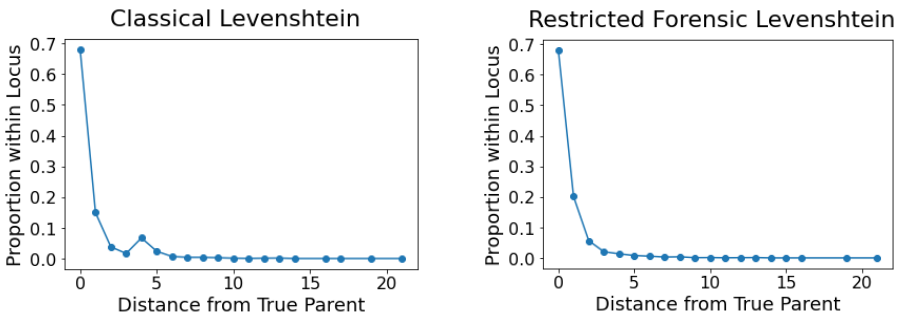
Various examples have been computed to compare the RFL distance behaves as it should with different costs when compared to the Cython package [32].

3. Results and Discussion

An overarching goal of this work was to develop a metric that was inversely proportional to sequence string frequency in MPS data analysis output. A demonstration of the effect of the RFL distance algorithm using the result from Table 2 is shown in Figure 2.

Table 2. High-stutter motifs at CODIS-20 and PENTA loci, based on 661 individuals, using a threshold of 0.167 Allele Coverage Ratio (ACR)—the ratio of the second-highest count to the highest—to determine homo- vs. heterozygosity in identifying true alleles. For each motif, LUS summary statistics are taken across all alleles. Back stutter rate is the average of all single-stutter proportions resulted from amplification and sequencing, averaged across all loci where parents with that motif were present. The last two columns describe how many alleles and unique alleles with the relevant motif at each locus were present in the data. This analysis is described further in Section 3.1.

Locus	Motif	Median LUS	Min LUS	Max LUS	Mean on-LUS Back Stutter Rate	Total Parents at Locus with Motif	Unique Parents at Locus with Motif
CSF1PO	TCTA	11	4	15	0.0219 (163)	1159	17
D10S1248	GGAA	14	8	19	0.0291 (190)	1182	13
D12S391	TAGA	12	7	19	0.0279 (115)	1287	84
D12S391	CAGA	6	3	10	0.0085 (67)	1287	84
D13S317	TATC	12	7	16	0.0183 (105)	1219	30
D13S317	AATC	2	2	3	0.0021 (56)	803	10
D16S539	GATA	11	5	14	0.0214 (123)	1207	17
D18S51	AGAA	15	9	24	0.0249 (125)	1242	27
D19S433	TCCT	13	7	18	0.0218 (133)	1194	22
D1S1656	TATC	13	9	17	0.0332 (141)	1270	30
D1S1656	AC	6	5	6	0.0059 (26)	1270	30
D21S11	TATC	12	7	15	0.0224 (96)	1264	79
D21S11	TGTC	6	4	8	0.0021 (21)	1264	79
D22S1045	ATT	12	5	16	0.0316 (233)	1179	13
D2S1338	GGAA	13	8	17	0.0221 (80)	1269	69
D2S1338	GGCA	7	3	9	0.0033 (34)	1269	69
D2S441	CTAT	11	7	14	0.0188 (114)	1205	24
D3S1358	CTAT	13	5	17	0.0309 (143)	1238	26
D3S1358	CTGT	2	2	4	0.0035 (85)	987	19
D5S818	ATCT	12	7	15	0.0257 (119)	1237	31
D7S820	CTAT	10	6	13	0.0163 (96)	1229	30
D8S1179	CTAT	12	8	15	0.0251 (115)	1237	34
D8S1179	CTGT	2	2	3	0.0013 (51)	20	6
FGA	GAAA	14	9	19	0.0211 (109)	1247	32
Penta D	GAAAA	11	5	17	0.0051 (47)	1236	18
Penta E	TTTTC	12	5	25	0.0114 (89)	1240	25
TH01	AATG	7	5	11	0.008 (63)	1165	12
TPOX	AATG	9	5	13	0.0113 (90)	1140	14
VWA	ATAG	12	3	16	0.0247 (146)	1242	32
VWA	ACAG	4	3	6	0.0023 (50)	1242	32
VWA	GATG	3	3	4	0.0019 (8)	81	2



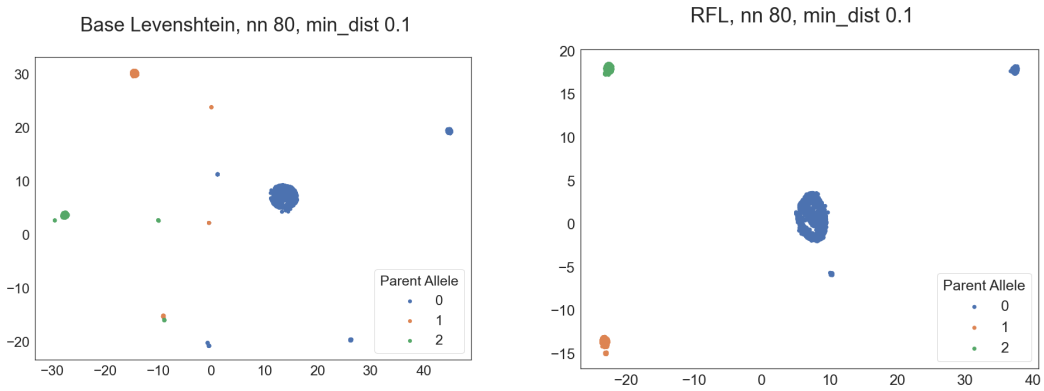
(**a**) Classical Levenshtein. (**b**) Restricted Forensic Levenshtein.

Figure 2. Proportion by distance plots at a CSF locus with unit edit costs. The RFL distance in (**b**) fixes the non-monotonicity of the classical Levenshtein in (**a**).

The non-monotonic plot in Figure 2 has a peak in frequency at a distance of 4 from the true parent sequence in the file, corresponding to stutter of an entire tetranucleotide motif being a more common occurrence than deleting 3 or 5 characters individually. The RFL distance counts this stutter with a cost of 1, thus giving monotonicity. In loci with motifs of other lengths, the frequency peak appears at that particular length. In loci exhibiting multiple motifs, there are increases in frequency at both lengths. There are smaller spikes at a distance of twice the motif length (a distance of 8 in Figure 2), corresponding to double stutter, but they are much less noticeable.

This metric also has applications in deconvolution, as shown in the following example of real data in a known mixture of two people at the locus D8S1179. According to Table 2, the RFL algorithm uses the motifs CTAT and CTGT. Two true alleles at this locus are p_1 = [CTAT]12 and p_2 = [CTAT]2 CTGT [CTAT]10. The artifact a = [CTAT]2 CTGT [CTAT]9 also appears in the results. It is desirable to capture sequence dissimilarity corresponding to decreased likelihood of an artifact being from a given allele. Using the Levenshtein distance with unit costs, the distance from p_1 to a is equal to 1, since the only edit required is changing the third A to a G. The Levenshtein distance from p_2 to a is equal to 4, since the motif CTAT must be removed letter-by-letter. However, when using the RFL distance, CTAT can be removed for a cost of 1, so then the distances between the artifact and each true allele are both equal to 1, and the deconvolution is less clear. Future work will fit costs to the data that vary by edit type to more confidently infer which parent was more likely the origin for the artifacts in cases such as this.

Another way to compare the two metrics is by visualizing the pairwise distances of output in two dimensions. Uniform Manifold Approximation and Projection (UMAP) is one method of performing this dimension reduction technique [33]. The UMAP algorithm was applied to all pairwise RFL distances and original Levenshtein distances of a 3:1 mixture at locus D1S1656. The locus is typed with three true alleles, with the higher contributor being homozygous and contributing three times as much DNA as the lower contributor, who was heterozygous for this locus. The RFL distance showed more defined clustering than classical Levenshtein across many combinations of hyperparameters, supporting the notion that the RFL distance captures PCR artifact similarity better than classical Levenshtein. Figure 3 shows a plot made with a representative combination of hyperparameters, where each artifact is colored by which parent it is closest to (there were no ties).



(**a**) Levenshtein, `nn` = 80, `min_dist` = 0.1.

(**b**) RFL, `nn` = 80, `min_dist` = 0.1.

Figure 3. UMAP plots for a ground-truth-known mixture of three parts homozygous contributor to one part heterozygous contributor at the locus D1S1656. Each point represents a sequence, and the colors represent which parent that sequence is closest to. The RFL distance gives better clustering of artifacts.

3.1. Selecting Motifs for Each Locus

The RFL algorithm requires a motif or set of motifs to include as options to lose or gain. The original motivation for the RFL algorithm was to address the high rates of stutter,

particularly back stutter, that appear in sequencing data analysis output. In order to give the algorithm the correct motifs to use, the available true allele sequences were exhaustively searched for any motif of length 2–6 that repeated at least twice consecutively.

For every locus, and for every potential motif identified at that locus, the back stutter rates were computed for all available files by taking the count of the stuttered artifact sequence divided by the sum of all reads for the locus. The results were then filtered to include only loci that showed back stutter at a mean rate of at least 0.001 across samples. Major stutter is the loss of a motif that decreases the longest uninterrupted stretch (LUS), i.e., stutter that results in the maximum number of consecutive motifs being reduced. Similarly, minor stutter is the loss of a motif that does not decrease the LUS. This differentiation was made because stutter in a LUS region was more common than stutter outside of a LUS region by several orders of magnitude in the data. Many places in the data showed stretches of a motif at locations other than the LUS region. For example, in every true allele at the locus CSF1PO, a locus with only one STR motif, there were two other locations of that motif repeating in lower numbers than the LUS in the data.

Motifs were grouped by self-permutations, e.g., if ATAG consecutively repeated at a sequence at least twice, usually TAGA and AGAT would as well. Within each group of self-permutations the motif with the highest mean LUS was chosen, using alphabetization to break ties. The final step was to take the motif at each locus with the largest mean LUS (there were no ties). This final candidate was chosen as the primary motif at the locus—in essence, the motif that had the highest mean stutter rate at that locus.

Next, every primary motif in every sequence in the data was substituted with the non-nucleotide letter Z, and the entire motif-finding process was repeated to look for secondary motifs—the motifs that stutter second-most at each locus. This mechanic of replacing the primary motifs with the letter Z ensured there was no overlap of primary and secondary motifs.

This process was iterated until no motifs with high stutter rates and LUS remained. Only VWA had candidate motifs for the tertiary round, and no loci had candidates for quaternary motifs.

Combining the three rounds of processing gives Table 2. Within each locus, the rows are sorted in order of the rounds of analysis (and thus in order of stutter frequency).

4. Conclusions

A new string dissimilarity measure is proposed with a flexible and publicly available algorithm for implementing it. It has shown promise in ongoing forensic applications thus far, but generalized Levenshtein with multi-character edits could have uses with in vivo sequences as well. Applications in graph edit distance and consequent potential uses in handwriting recognition, fingerprint recognition, and cheminformatics were also discussed. Allowing the interaction of edit types—such as a motif stuttering forward, then undergoing indels or substitutions within, before, or after that motif–is a novel addition that expands the useful applications of this work, and is accomplished via the dictionary structure in a way that is fast, feasible, and open to further speed optimization. It allows for a more accurate picture of the stutter phenomenon and how similar two STR sequences truly are when any stutter is expected, whether from PCR or in vivo polymerase slippage.

In practice, this work and the associated code available can benefit those analyzing STR sequencing data, manufacturers and developers providing software tools to accurately recognize and label sequences from these data types, thus, indirectly contributing to reliable reporting of STR sequence data, both in the forensic genetics and the medical diagnosis fields where accuracy in interpretation of STR sequence data is crucial. To count the number of edit operations to get from one sequence to another, set all edit costs to one (this will be symmetric, since every operation will have an inverse of equal cost). This is a useful first pass on sequence similarity. Optimizing the costs to reflect the frequency of the edits in the data requires a kit-specific understanding of artifact frequencies, since those would be used to determine the relative costs of each operation. Applications such as sequence

alignment may use costs motivated by traits other than pure artifact frequency, depending on the information desired.

Further work could be done to penalize the overall number of edit operations in addition to the cost of each individual operation. It would also be a simple addition to consider transpositions as a separate possible edit operation, though for the original application it was unnecessary.

The restrictions in place because of the stutter dictionary have proven helpful for several reasons, but do limit effectiveness in rare cases. A motif inserting in the middle of another motif, for example, would not be detected by the RFL algorithm, and neither would a motif spread apart by too many single-character intra-motif insertions (len(motif)-1 insertions, in particular). Although this limit is easily changeable in the code, the way the algorithm is written requires a hard cap. For forensic applications, however, when the strings are only a few edits apart and insertions are rare, the restrictions do not limit effectiveness.

5. Disclaimer

Any commercial equipment, instruments, materials, or software identified in this paper are to foster understanding only. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials, equipment, or software identified are necessarily the best available for the purpose.

Author Contributions: Conceptualization, T.P., J.H. and H.I.; methodology, T.P., J.H. and H.I.; software, T.P.; validation, T.P., J.H. and H.I.; formal analysis, T.P., J.H. and H.I.; investigation, T.P.; resources, T.I.H. and H.I.; data curation, T.P. and T.I.H.; writing—original draft preparation, T.P.; writing—review & editing, T.P., T.I.H., H.I. and J.H.; visualization, T.P.; supervision, J.H. and H.I.; project administration, J.H. and H.I.; funding acquisition, J.H., H.I. and T.I.H. All authors have read and agreed to the published version of the manuscript.

Funding: The research of T.P. and J.H. was supported in part by the National Science Foundation under Grant No. DMS-1916115 and 2113404. T.I.H. was funded by the NIST Special Programs Office (Forensic Genetics Focus Area).

Institutional Review Board Statement: All work has been reviewed and approved by the National Institute of Standards and Technology Research Protections Office. This study was determined to be "not human subjects research" (often referred to as research not involving human subjects) as defined in U. S. Department of Commerce Regulations, 15 CFR 27, also known as the Common Rule (45 CFR 46, Subpart A), for the Protection of Human Subjects by the NIST Human Research Protections Office and therefore not subject to oversight by the NIST Institutional Review Board.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are from the National Institute of Standards and Technology, but restrictions apply to the availability of these data, which were used under license for the current study, and so are not publicly available. For inquiries, please contact Hari Iyer at hariharan.iyer@nist.gov.

Acknowledgments: Thank you to Katherine B. Gettings, Peter M. Vallone, Lisa A. Borsuk, and the NIST Applied Genetics Group for sharing the data. Special thanks to Katherine B. Gettings for in-depth discussion related to the underlying biological concepts and data details.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

STR	Short Tandem Repeat
RFL	Restricted Forensic Levenshtein
PCR	Polymerase Chain Reaction
CE	Capillary Electrophoresis
MPS	Massively Parallel Sequencing
DoC	Depth of Coverage
ACR	Allele Coverage Ratio

Appendix A

In this appendix, further details of the complexities and examples of the Restricted Forensic Levenshtein algorithm are shown.

Appendix A.1. Complexities

Allowing for the possibility of both single- and multiple-character gains and losses in interaction makes the algorithm exponentially more complex. The insert costs are computed via the weighted Levenshtein distance as if the edit path was first stuttered forward, then edited from the motif (with no stutters) to the string. The delete costs are computed by forcing the string to edit back into a motif, then adding the cost for dropping a motif.

In a case where the motif is AAAGA and all edit costs are equal to 1, evidently the cost of inserting or deleting a single A or G would be 1; however, in the constructed dictionary the cost of these operations are set to 5 each, resulting from a forward stutter (1) and the removal of the remaining 4 letters (4×1). The cost of inserting or deleting a letter not part of the main motif, e.g., T, is set to 6 in the dictionary, as it is calculated with the same steps as before (5) and an additional change of the last remaining letter to T (1). Strings of all lengths up to $2k - 1$ are also included in the dictionary, i.e., costs of insertion or deletion are set for strings AAA to 3, AAAGAT to 2 and AAACGATC to 4, and so forth.

Cost can be set asymmetrically for insertions and deletions. If insertion costs were changed to 2 but deletions, substitutions, and stutters remained at a cost of 1, the dictionary entries would change. The string AAA in this dictionary would have an insert cost of 3 (blank $\rightarrow$ AAAGA (+1) $\rightarrow$ AAA (+2)), but a delete cost of 5 (AAA $\rightarrow$ AAAGA (+4 for two inserts) $\rightarrow$ blank (+1 for stutter)).

It gets more complicated when one considers adding two or three motifs at once, as shown in Example A1.

Example A1. *In this example the motif is AAAGA. Forward stutter is followed by many individual letters inserted in between, as shown:*

$$\boldsymbol{A}(TTTTAAAGATTTT)\boldsymbol{AAGA}.$$

To get to the same string, another route would be to stutter once and then insert another motif in between the letters of the motif that is already there, along with other insertions, as follows:

$$\boldsymbol{A}(TTTT)\boldsymbol{AAAGA}(TTTT)\boldsymbol{AAGA}.$$

The RFL distance does not address stutter-within-stutter, in either the forward or backward direction. Beginning an attempt to do so would require allowing the second stutter to take place at any point in the first motif, e.g., for the motif AAGG, consider A(AAGG)AGG, AA(AAGG)GG, and AAG(AAGG)GG, in addition to accounting for all the single-character operations that could take place on either motif, and once again bounding above by some maximum number of possible single-character insertions.

Additionally, the RFL distance does not include cases where more than $k - 1$ characters are inserted or deleted between an outer motif's characters, where k is the longest motif, since the maximum length of strings included in the lookup dictionary is $2k - 1$.

Appendix A.2. A Worked Example

The example below is shown with unit costs for all operations. The motif is ACG. First an original Levenshtein example is computed in detail, followed by an RFL example. The distance being computed is from ACG to ACGTCG.

Each matrix entry is equal to the distance from the parent string (up to that row) and the child string (up to that column). For example, the bold 4 in Table A1 corresponds to the distance from A to ACGTC. The value at the bottom-right of Table A1 (i.e., 3) is the least costly solution for the edits from parent to child. This follows the intuition that in the base Levenshtein distance, the necessary edits to get from ACG to ACGTCG are three insertions in sequence, each with a cost of 1.

Table A1. Standard Levenshtein.

		A	**C**	**G**	**T**	**C**	**G**
	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
C	2	1	0	1	2	3	4
G	3	2	1	0	1	2	3

The RFL distance between the same two strings with the same motif as in Table A1 is computed in Table A2.

Table A2. Restricted Forensic Levenshtein.

		A	**C**	**G**	**T**	**C**	**G**
	0	1	2	1	2	3	3
A	1	0	1	2	2	3	4
C	2	1	0	1	2	2	3
G	1	2	1	0	1	2	2

The following paths can be described from Table 2: A to ACG requires two edits, either via A → AC → ACG or A → AACG → ACG, while A to ACGTC requires three, via A → T → ACGT → ACGTC, and AC to ACGTCG also requires three, via AC → ACACG → ACTCG → ACGTCG.

The final function value is given in the bottom right corner of the matrix in Table A2, representing the RFL distance from ACG to ACGTCG. This value of 2 is obtainable via the edit path ACG → ACGACG → ACGTCG.

Changing a few individual edit costs will give different optimal paths. In the following example, the cost of inserting a motif (forward stutter) is 2, backward stutter costs 0.5, switching from A to T costs 1.5, switching from A to C costs 0.5, and inserting a C or a T costs 2. All other costs remain equal to 1. The dynamic programming matrix computed with modified costs is shown in Table A3.

Table A3. RFL with modified costs.

		A	**C**	**G**	**T**	**C**	**G**
	0	1	3	2.5	4.5	6.5	6.5
A	1	0	2	3	4	6	7
C	1.5	1	0	1	3	4	5
G	0.5	1.5	1	0	2	4	4

The previous three examples are shown below, but now with updated costs.

In the previous example, there were two paths from A to ACG that each cost 2. A → AC (+2) → ACG (+1) now costs 3, and A → AACG (+2.5) → ACG (+1) now costs 3.5. The former path cost matches the cost of 3 in Table A3, and is thus still a minimal path, but the latter is no longer minimal.

In the previous example, A to ACGTC cost 3. Now the same path A → T (+1.5) → ACGT (+2.5) → ACGTC (+2) costs 6, which matches Table A3.

For AC to ACGTCG, there was a path that cost 3, but now the cost should be 5 to match Table A3. AC → ACACG (+2.5) → ACTCG (+1.5) → ACGTCG (+1) costs 5, as desired.

Other cost combinations are possible. For example, if the cost of forward stutter was raised higher than 3 while keeping unit costs for insertion, forward stutter would never be reflected in a minimal edit path because it would always be cheaper to insert the motifs individually instead of stuttering them together.

Appendix A.3. An Algorithmic Limitation

It is possible to weight the costs unequally enough that interferes with the output of the original Levenshtein. If the insert cost of C is 5, but every other edit cost is 1, then the actual cost of inserting a C is 2, since one could insert any other letter (+1) and then substitute in a C (+1). However, the Levenshtein function in the Weighted Levenshtein package available in Python does not reflect this. It will return $lev(\text{''}, \text{C}) = 5$, not 2.

Forcing all insert costs to be the same in a given example actually avoids this issue altogether—for example, the `stringdist` package in R does not allow for individual insertions to have different costs from one another [34,35]. All deletions must cost the same, as well, as do substitutions.

If the insert cost of C is 1.5, for example, and every other edit cost is 1, then the cost of inserting a C is legitimately 1.5, since any other possible path returns at least 2. In order to have truly minimal-cost answers, then, it is necessary to enter in the individual edit costs in the standard Levenshtein function as minimal (e.g., insertion cost of any letter needs to be less than the sum of any other possible edit sequence to insert that letter indirectly).

Insertion is not the only place where this is a problem. If the cost of A → T was 5, but all other costs were 1, then A → C → T would cost 2, so A → T should cost 2. A similar example could be constructed for deletion.

The same does not hold true for stutter, however. If the forward stutter cost was raised to 15 and the backward stutter cost raised to 10 in the RFL algorithm, the matrix for the RFL distance (which is equal to the standard Levenshtein matrix) is shown in Table A4.

Table A4. RFL with high stutter penalties.

		A	**C**	**G**	**T**	**C**	**G**
	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
C	1	1	0	1	2	3	4
G	3	2	1	0	1	2	3

There are no costs of 10 or higher, since it is never cheaper to stutter than to address each nucleotide separately. Thus, the addition of motifs to the Weighted Levenshtein package does not suffer from the same weaknesses. Because it uses the package as a base, though, it still has the same problems as the original Levenshtein distance with the other edit operations.

The solution to this issue is a pre-processing step to ensure that the individual edit costs are already optimal prior to the application of the Levenshtein distance. It is more complicated than, e.g., ensuring that a two-step insert-substitute path is not cheaper than direct insertion. Pathological counterexamples exist that require more than two steps—a demonstration is shown in Example A2.

Example A2. *In a case where the following costs are set:*

- *Cost to insert a C, T, or G: 10*
- *Cost to insert an A: 1*
- *Cost of $T \rightarrow C$, $A \rightarrow C$, or $A \rightarrow G$: 10*
- *Cost of $G \rightarrow C$, $A \rightarrow T$, or $T \rightarrow G$: 1*

Then in order to insert a C, it is cheaper to edit $'' \rightarrow A \rightarrow T \rightarrow G \rightarrow C$ for a total cost of 4 than it is to insert a C directly (10), but one cannot go $'' \rightarrow A \rightarrow C$ (cost of 11) or $'' \rightarrow A \rightarrow T \rightarrow C$ (cost of 12).

In the application the RFL was designed for, however, all deletions cost the same, all insertions cost the same, and all substitutions cost the same, so this issue is avoided.

The RFL algorithm is based on the idea that the sequences undergoing the edits in PCR should take the cheapest, most frequent path. Here distance is defined to be the lowest possible cost of any edit path between the strings (regardless of whether the minimal path is itself unique), with no penalty on the number of steps. The implementation of edit count penalization to the algorithm is an area of interest for future work.

References

1. Rinartha, K.; Suryasa, W.; Kartika, L.G.S. Comparative Analysis of String Similarity on Dynamic Query Suggestions. In Proceedings of the 2018 Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS), Batu, Indonesia, 9–11 October 2018; pp. 399–404. [CrossRef]
2. Alberga, C.N. String Similarity and Misspellings. *Commun. Acm* **1967**, *10*, 302–313. [CrossRef]
3. Cheatham, M.; Hitzler, P. String Similarity Metrics for Ontology Alignment. In *Proceedings of the The Semantic Web—ISWC 2013*; Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 294–309.
4. Chang, W.I.; Lawler, E.L. Sublinear approximate string matching and biological applications. *Algorithmica* **1994**, *12*, 327–344. [CrossRef]
5. Alsmadi, I.; Nuser, M. String Matching Evaluation Methods for DNA Comparison. *Int. J. Adv. Sci. Technol.* **2012**, *47*, 13–32.
6. Qi, X.; Wu, Q.; Zhang, Y.; Fuller, E.; Zhang, C.Q. A Novel Model for DNA Sequence Similarity Analysis Based on Graph Theory. *Evol. Bioinform.* **2011**, *7*, EBO.S7364. [CrossRef] [PubMed]
7. Butler, J.M. The future of forensic DNA analysis. *Philos. Trans. R. Soc.* **2015**, *370*, 20140252. doi: doi: 10.1098/rstb.2014.0252. [CrossRef] [PubMed]
8. Clayton, T.; Whitaker, J.; Maguire, C. Identification of bodies from the scene of a mass disaster using DNA amplification of short tandem repeat (STR) loci. *Forensic Sci. Int.* **1995**, *76*, 7–15. [CrossRef]
9. Andelinović, S.; Martín, P.; Sutlović, D.; Erceg, I.; Huffine, E.; de Simón, L.F.; Albarrán, C.; Definis-Gojanović, M.; Fernández-Rodriguez, A.; García, P.; et al. DNA typing from skeletal remains: Evaluation of multiplex and megaplex STR systems. *Croat. Med. J.* **2001**, *42*, 260–266.
10. Budowle, B.; Schmedes, S.E.; Wendt, F.R. Increasing the reach of forensic genetics with massively parallel sequencing. *Forensic Sci. Med. Pathol.* **2017**, *13*, 342–349. [CrossRef]
11. Urquhart, A.; Kimpton, C.P.; Downes, T.J.; Gill, P. Variation in Short Tandem Repeat sequences—A survey of twelve microsatellite loci for use as forensic identification markers. *Int. J. Leg. Med.* **1994**, *107*, 13–20. [CrossRef]
12. Alford, R.L. Rapid and efficient resolution of parentage by amplification of short tandem repeats. *Am. J. Hum. Genet.* **1994**, *55*, 190–195.
13. Frégeau, C.; Fourney, R. DNA typing with fluorescently tagged short tandem repeats: A sensitive and accurate approach to human identification. *BioTechniques* **1993**, *15*, 100–119.
14. Gettings, K.B.; Kiesler, K.M.; Faith, S.A.; Montano, E.; Baker, C.H.; Young, B.A.; Guerrieri, R.A.; Vallone, P.M. Sequence variation of 22 autosomal str loci detected by next generation sequencing. *Forensic Sci. Int. Genet.* **2016**, *21*, 15–21. [CrossRef]
15. Brookes, C.; Bright, J.A.; Harbison, S.; Buckleton, J. Characterising stutter in forensic STR multiplexes. *Forensic Sci. Int. Genet.* **2012**, *6*, 58–63. [CrossRef]
16. Raz, O.; Biezuner, T.; Spiro, A.; Amir, S.; Milo, L.; Titelman, A.; Onn, A.; Chapal-Ilani, N.; Tao, L.; Marx, T.; et al. Short tandem repeat stutter model inferred from direct measurement of in vitro stutter noise. *Nucleic Acids Res.* **2019**, *47*, 2436–2445. [CrossRef]
17. Daunay, A.; Duval, A.; Baudrin, L.G.; Buhard, O.; Renault, V.; Deleuze, J.F.; How-Kit, A. Low temperature isothermal amplification of microsatellites drastically reduces stutter artifact formation and improves microsatellite instability detection in cancer. *Nucleic Acids Res.* **2019**, *47*, e141. [CrossRef]
18. Brill, E.; Moore, R.C. An improved error model for noisy channel spelling correction. In Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics, Hong Kong, China, 3–6 October 2000; pp. 286–293. [CrossRef]
19. Boytsov, L. Indexing methods for approximate dictionary searching. *Acm J. Exp. Algorithmics* **2011**, *16*, A8–A9. [CrossRef]

20. Ukkonen, E. Algorithms for approximate string matching. *Inf. Control* **1985**, *64*, 100–118. [CrossRef]
21. Gao, X.; Xiao, B.; Tao, D.; Li, X. A survey of graph edit distance. *Pattern Anal. Appl.* **2009**, *13*, 113–129. [CrossRef]
22. Fischer, A.; Suen, C.Y.; Frinken, V.; Riesen, K.; Bunke, H. Approximation of graph edit distance based on Hausdorff matching. *Pattern Recognit.* **2015**, *48*, 331–343. [CrossRef]
23. Neuhaus, M.; Bunke, H. Automatic learning of cost functions for graph edit distance. *Inf. Sci.* **2007**, *177*, 239–247. [CrossRef]
24. Darwiche, M.; Conte, D.; Raveaux, R.; T'Kindt, V. Graph edit distance: Accuracy of local branching from an application point of view. *Pattern Recognit. Lett.* **2020**, *134*, 20–28. [CrossRef]
25. Petty, T. restricted-forensic-levenshtein. *GitHub*, 2021. Available online: https://github.com/taylorpetty/restricted-forensic-levenshtein (accessed on 1 May 2022).
26. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
27. Zhao, C.; Sahni, S. String correction using the Damerau-Levenshtein distance. *BMC Bioinform.* **2019**, *20*, 1–28. [CrossRef]
28. Hirschberg, D.S. Algorithms for the longest common subsequence problem. *JACM* **1977**, *24*, 664–675. [CrossRef]
29. Wagner, R.A.; Lowrance, R. An Extension of the String-to-String Correction Problem. *JACM* **1975**, *22*, 177–183. doi: 10.1145/321879.321880. [CrossRef]
30. Rane, S.; Sun, W. Privacy preserving string comparisons based on Levenshtein distance. In Proceedings of the 2010 IEEE International Workshop on Information Forensics and Security, Seattle, WA, USA, 12–15 December 2010; pp. 1–6.
31. Woerner, A.E.; King, J.L.; Budowle, B. Fast STR allele identification with strait razor 3.0. *Forensic Sci. Int. Genet.* **2017**, *30*, 18–23. [CrossRef]
32. Su, D. weighted-levenshtein. *Python Software Foundation*, 2018. Available online: https://pypi.org/project/weighted-levenshtein/ (accessed on 1 May 2022).
33. McInnes, L.; Healy, J.; Melville, J. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv* **2016**, arXiv:1603.00278.
34. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2021.
35. van der Loo, M. The stringdist package for approximate string matching. *R J.* **2014**, *6*, 111–122.