Slice Distance: An Insert-only Levenshtein Distance with a Focus on Security Applications

Zeeshan Afzal, Johan Garcia, Stefan Lindskog, and Anna Brunstrom
Department of Computer Science, Karlstad University, 651 88, Karlstad, Sweden
firstname.secondname@kau.se

***Abstract*—Levenshtein distance is well known for its use in comparing two strings for similarity. However, the set of considered edit operations used when comparing can be reduced in a number of situations. In such cases, the application of the generic Levenshtein distance can result in degraded detection and computational performance. Other metrics in the literature enable limiting the considered edit operations to a smaller subset. However, the possibility where a difference can only result from deleted bytes is not yet explored. To this end, we propose an insert-only variation of the Levenshtein distance to enable comparison of two strings for the case in which differences occur only because of missing bytes. The proposed distance metric is named *slice distance* and is formally presented and its computational complexity is discussed. We also provide a discussion of the potential security applications of the *slice distance*.**

***Index Terms*—edit-distance, Levenshtein distance, intrusion detection**

I. Introduction

The problem of matching strings in the presence of errors has been around for decades. The general idea is to establish whether a given input pattern matches any text in a database or a dictionary by allowing a number of allowed errors in the matching. The notion behind matching with errors is the increasing probability of not finding exact matches in many areas. This kind of approximate matching has its applications in many diverse fields [1]. In medical science, comparison of Deoxyribonucleic acid (DNA) sequences and protein samples for similarity is frequently performed. This is analogous to string matching as DNA can be seen as long chains of strings. In information theory, modification or corruption of messages sent over a transmission channel is possible. The receiver of such messages faces the task of recovering the original message from what is received. In the field of computer science, approximate string matching is used extensively in spell checkers and optical character recognition (OCR) systems. It also finds its applications in intrusion detection [2].

Comparing strings for similarity is a core part of approximate string matching. Traditionally, edit distance is a commonly used tool to measure how similar two strings are to each other. The similarity is measured by how many operations are required to convert one string into the other. The considered operations can include insertions, deletions, transpositions, and substitutions of one or more characters in a string to transform it into the other string. There are several distance metrics to measure the edit distance between two given strings. Levenshtein distance [3], named after its inventor, is the most commonly used edit distance metric today. The general purpose Levenshtein distance works for many cases and considers insertions, deletions, and substitutions when calculating the distance. However, in some cases, depending on the application, it might be beneficial to limit the allowed operations based on the knowledge of what kind of errors can occur. This limitation of possible operations can improve the detection and computational performance. A number of simplified metrics based on Levenshtein distance have therefore been proposed in the literature [1], [4]–[6].

In this work, we propose yet another metric based on the Levenshtein distance. The proposed *slice distance* metric deals with the special case when errors can only result from the loss of one or more character(s) or byte(s). This could happen by-design or because of a lossy transmission channel. Nevertheless, the extra information about the error characteristics means that there is no need for operations such as substitutions, deletions, and transpositions. Deleted characters or bytes can only be corrected by insertions. The *slice distance* compares two strings by counting the number of inserts required in a string to match it with another string in a database or in a dictionary.

The rest of the paper is structured as follows. Section II summarizes some of the related work in the field of string matching and edit distances. Section III formally presents the *slice distance* algorithm and its variations along with a discussion on time complexity. Section IV details on the security related applications of the *slice distance*. Finally, Section V summarizes the presented work.

II. Related Work

One early edit distance metric is the Levenshtein distance [3], which is commonly used across different fields. The Levenshtein distance gives a measure of the minimal number of operations (insertions, deletions, and substitutions) necessary in order to match one string with another. In its basic form, the cost of each operation is 1 and the time complexity is O(M^2) when both strings have the same size M. This general purpose Levenshtein distance finds many applications in approximate string matching [1]. Damerau-Levenshtein distance [7] is an extension that adds transpositions to the set of allowed operations, and it has diverse applications in fields such as language processing (e.g., spell checkers) and computational biology.

978-1-5386-3662-6/18/$31.00 ©2018 IEEE

There are also restricted distance metrics that only consider a smaller subset of the edit operations. One example is the Hamming distance [4], which is defined as the number of places for which two strings are different from each other. In other words, it considers the minimum number of substitution operations needed to go from one string to the other. The hamming distance or codes are used prevalently in areas where errors cause equal-length strings to differ (e.g., coding theory).

Longest common subsequence (LCS) [1] is yet another metric that considers insertions and deletions as the only two allowed operations. It finds the longest common subsequence between the two strings to be matched. The length of the identified common substrings is then used to calculate a similarity metric.

With regards to distance metrics that limits the set of operations to inserts only, this has been proposed in the context of episode matching by Das et al. [8]. However, for the episode matching problem, inserts are considered for the pattern to match some observed text substring, whereas the proposed *slice distance* considers inserts in a text substring to match a pattern.

Petrovic and Golic [6] considered constrained edit distance calculations. They showed that by fixing the number of allowed substitutions and deletions to zero, an insert-only variant of the Levenshtein distance could be realized. However, a disadvantage of that scheme is that it can be more computationally expensive, i.e., O(M^3) or O($M^2 \ln M$) compared to the O(M^2) of the Levenshtein distance.

III. Formal and Algorithmic Descriptions

In this section, we formally define four variations of the *slice distance* and discuss implementation and complexity aspects. We use the following definitions in the rest of the section. A substring of a string X can be composed by any number of adjacent characters from string X. A subsequence of a string X is any string obtainable by removing zero or more characters from X. In the following, we consider the case of a pattern *P*, being a string with length $M = |P|$. Further, we have a text *T*, being a string with length $N = |T|$. The task is to determine the minimum number of inserts necessary to transform a substring of T to P, subject to various constraints. This is different from the episode distance [8] where the task is to determine the minimum number of inserts necessary to transform P to a substring of T. We note that the asymmetric nature of the *slice distance* makes it a quasimetric from a mathematical point of view.

A. Prefix Slice Distance

The initial distance metric has a prefix constraint on text T. What is of interest here is thus to determine the extent to which a prefix substring T_p of T, with $T_p : t_1, ..., t_u,\ u \leq M$, is a subsequence of P.

Let d_p be the proposed *prefix slice distance*. The problem is thus: given two strings, pattern P and text T, determine the minimum number of inserts that are needed to transform a prefix T_p of T to P. An inequality bounding the values of d_p can be expressed as:

$$M - u \leq d_p(T, P) \leq M \qquad (1)$$

1) Properties: From the above, we can see that the following properties hold for the proposed *prefix slice distance*.

- Regardless of N, only characters $T_1...T_M$ in T needs to be evaluated due to the prefix constraint.
- The lower bound on the distance is at least the difference in length of the pattern string and the considered prefix, i.e., $d_p(T, P) \geq M - |T_p|$.
- In the worst-case, when none of the pattern characters match the pattern (P), the distance could at most be equal to the length of pattern (P) i.e., $d_p(T, P) \leq M$.
- The distance will be zero only when a prefix of T fully matches the pattern P.

2) Algorithm: The algorithm is presented in Algorithm 1. The output of the algorithm is the minimum number of insertions necessary to transform a prefix of T into P. After execution, $insertions$ equals $d_p(T, P)$, and $insertions + matches$ equals the pattern size M.

Algorithm 1 Prefix slice distance algorithm.

```
Input: text, pattern
 1: $insertions \leftarrow 0$
 2: $matches \leftarrow 0$
 3: $M \leftarrow len(pattern)$
 4: for $i\ in\ range[M]$ do
 5:     while $pattern[i + insertions] \neq text[i]$ do
 6:         $insertions \leftarrow insertions + 1$
 7:         if $i + insertions == M$ then
 8:             return $insertions$
 9:         end if
10:     end while
11:     $matches \leftarrow matches + 1$
12:     if $matches + insertions == M$ then
13:         return $insertions$
14:     end if
15: end for
```

3) Examples: Consider an example where P = “matchme” with length $M = 7$ and T = “mate” with length $N = 4$. In this case, the outcome of $d_p(T, P)$ will be 3. Three characters “chm” can be inserted in T to match it to P. For this case, the distance is the difference in the length of two strings, i.e., $d_p(T, P) = 3 = M - N$. Now consider another example with the same P but with T = “malchme”. In this instance, $d_p(T, P) = 5$. This can be contrasted to the example T = “athmeablag” which visually appears less similar to the pattern “matchme” but has a lower distance at $d_p(T, P) = 2$.

4) Algorithmic Complexity: Time complexity is a way of describing how many operations an algorithm takes to perform its function. For Algorithm 1, it can be seen that in general, the pattern length M is the bound on time complexity. Let us now use the examples from above to further explain the time complexity. We use the Big-O notation for the description of time complexity. First we consider the identical strings scenario, where the two strings that are to be matched are exactly the same. Assume that P = "matchme" and T = "matchme". Here $M = N = 7$ and the distance $d_p(T, P)$ between them will be 0. Table I shows the time complexity for each line of the code outlined in Algorithm 1 when it runs for the given strings ***T*** and ***P***. If we count the number of times each line in the code gets executed, we get $O(3M + 5)$ or $O(aM + c)$ if we generalize. Since, in the Big-O notation, constant c can be ignored, it becomes $O(3M)$. We can also factor out constants to arrive at a complexity of $O(M)$. This shows that, for this case, the algorithm's time consumption is directly proportional to length M of pattern string P that is being matched.

TABLE I: Time complexity for Algorithm 1.

Algorithm Line	Identical strings complexity	Non-matching strings complexity
1 to 4	$O(4)$	$O(4)$
5	$O(M)$	$O(M)$
6 to 7	zero	$O(2M)$
8	zero	$O(1)$
9 to 10	n-a	n-a
11	$O(M)$	zero
12	$O(M)$	zero
13	$O(1)$	zero
14 to 15	n-a	n-a

In the non-matching strings scenario, the two strings do not match each other at the prefix for any character. Assume the same P = "matchme" as before and a T = "nolchme". Here $M = N = 7$ and the distance $d_p(T, P)$ between them will be 7. Table I again shows the time complexity for each line of the code outlined in Algorithm 1. If we count the number of times each line in the code gets executed, we get $O(3M+5)$ or $O(aM+c)$ if we generalize. Again, constant c can be ignored and as the cost of an algorithm is represented by its most costly operation and the fastest growing term is M, then $O(M)$ is the worst case time complexity. This shows that, regardless of input, the algorithm's time performance is directly proportional to length M of pattern string P that is being matched.

B. General Slice Distance

Now we consider the case in which the constraint that the *slice distance* should be computed on a prefix of the text T is removed. Formally, the problem similarly has a pattern string P, with length $M = |P|$, and text T with length $N = |T|$. Instead of working on the prefix T_p, the problem now is to find a substring T_s in T such that $T_s : t_j ... t_{j+u},\ u \leq M$, has a subsequence of P, and there is no substring T_s that has a lower $d_p(T_s, P)$.

Let d_g be the proposed function. The problem is thus: given two strings, pattern ***P*** and text ***T***, determine the minimum number of inserts that are needed to transform any subsequence of ***T*** to ***P***. The inequality governing d_g can be expressed as:

$$0 \leq d_g(T, P) \leq M \qquad (2)$$

1) Properties: From the above, the following properties can be derived:

- Only when $M > N$ is the lower bound on the distance nonzero. In that case the bound is $M - N \leq d_g(T, P)$.
- In the worst-case, when none of the pattern characters match the pattern P, the distance could be at most equal to the length of pattern P i.e., $d(T, P) \leq M$.
- The distance will be zero only when T contains a substring that fully matches the pattern P.

2) Algorithm: The algorithm is presented in Algorithm 2.

Algorithm 2 General slice distance algorithm.

```
Input: text, pattern
 1: $M \leftarrow len(pattern)$
 2: $N \leftarrow len(text)$
 3: $minimum\_d \leftarrow M$
 4: for $j$ in $range[N]$ do
 5:     $insertions \leftarrow 0$
 6:     $matches \leftarrow 0$
 7:     for $i$ in $range[M]$ do
 8:         while $pattern[i + insertions] \neq text[j + i]$ do
 9:             $insertions \leftarrow insertions + 1$
10:             if $insertions == minimum\_d$ then
11:                 continue loop 4-28
12:             end if
13:             if $i + insertions == M$ then
14:                 break loop 7-21
15:             end if
16:         end while
17:         $matches \leftarrow matches + 1$
18:         if $matches + insertions == M$ then
19:             break loop 7-21
20:         end if
21:     end for
22:     if $matches + insertions == M$ then
23:         $minimum\_d \leftarrow insertions$
24:         if $minimum\_d == 0$ then
25:             return $minimum\_d$
26:         end if
27:     end if
28: end for
29: return $minimum\_d$
```

3) Examples: Again consider an example where P = "matchme" with length $M = 7$. As the *general slice distance* does not have the prefix constraint, for T="randomtext123matcxyz" the $d_g(T, P) = 3$ while the prefix variant would find no match, thus giving $d_p(T, P) = 7 = M$. Another example is T = "malchme" with length $N = 7$. For this case, the less constrained *general slice distance* $d_g(T, P) = 3$, while the prefix variant was $d_p(T, P) = 5$.

4) Algorithmic Complexity: The complexity of the *general slice distance* can be evaluated by observing that it contains an inner loop that behaves similarly to the prefix distance calculation. If, as an initial step, the inner loop is assigned the same complexity of O(M), the outer loop can then be added. As is clear from line 4 the outer loop has complexity O(N), and thus the resulting complexity would now be O(MN). At closer inspection it can however be noted that the inner loop now has a new escape statement in lines 10-12. When searches are performed over strings with larger and larger M and/or N, the *minimum_d* value becomes increasingly likely to decrease more and more from its initial value of M during the progress of each search. Furthermore, it seems reasonable that the speed of decrease in *minimum_d* is also a function of the alphabet size A. Consequently, the average complexity would be lower than O(MN), and can be expressed as O($MN/f(A, M, N)$, where a detailed determination of f() is not considered in this paper.

C. Multi-pattern General Slice Distance

The previous subsection computed the minimal *general slice distance* between a text T and a single pattern P. This can be extended to instead compute the minimal *general slice distance* between the text T and a set of patterns, $P^S = \{P_1, ..., P_V\}$ with set size V. There is also the pattern lengths set $M^S = \{M_1, ..., M_V\}$. Let d_m be the proposed function. The problem is thus: given one string with text T, and a set of patterns P^S, determine a pattern P_x and the number of minimum inserts that are needed to transform a subsequence of T to P_x such that no pattern exists that has a lower number of minimum inserts. The inequality governing d_m can be expressed as:

$$0 \leq d_m(T, P^S) \leq \max\left(M^S\right) \qquad (3)$$

1) Properties: Since there now exists a set of patterns, the distance calculations create a set of pairs $P_x : d_g(T, P_x)$. The multi-pattern nature extends the properties of the *general slice distance* in the following way.

- The distance value of a particular pattern P_x lies within $0 \leq d_g(T, P_x) \leq M_x$.
- The distance values of the patterns in set P^S are bounded by $0 \leq d_m(T, P^s) \leq \max(M^S)$.
- The minimal distance value for a set of patterns is in the range $0 \leq d_m(T, P^S) \leq \min\left(M^S\right)$.
- Among the set of pattern-distance value pairs, one or more entries will have a distance value equal to the minimal observed distance.

Given that there now can exist multiple patterns that all have the same minimal distance value, and that patterns can vary considerably in size, some mechanism to characterize the different pattern-distance pairs can be useful. As an initial approach we consider calculating a coverage value C for each $P_x : d_g(T, P_x)$ pair such that $C_x = (M_x - d_g(T, P_x))/M_x$. Depending on the application, C_x, may be useful as further information when multiple patterns have the same minimal distance $d_g(T, P_x)$.

2) Examples: Consider an example where $P^S = \{"findme", "pinpointme"\}$ and T="abcpinpoindmex" which will give $d_m = 1$ and $C_1 = 0.83$ for "findme". Another example is T=" "xxpinoinmexx". In this case, $d_m = 2$ and thus the algorithm allows either strings to be returned as they both have $d_g(T, P) = 2$. As this example illustrates, multi-pattern matching will for many applications benefit from considering not just the distance metric but also additional metrics such as pattern coverage as they differ, here with $C_1 = 0.67$ and $C_2 = 0.8$.

3) Algorithmic Aspects: The *general slice distance* algorithm can be wrapped by a loop over all patterns, resulting in a complexity of O($\overline{M^S}NV/f(A, M^S, N)$). Such complexity may be prohibitive for many practical applications. Exact multi-pattern search (i.e. without allowing any insertions) can be naively implemented in O(NV), and speed up considerably by using algorithms such as the Aho-Corasick [9] algorithm. However, extending such an algorithm to allow insertions has, as far as we know, not been achieved.

D. Multi-pattern General Fix-constrained Slice Distance

While the three previous distance metrics have been discussed in order of increasing generality, we now instead introduce a restriction in relation to the previous metric. The fix-constraint signifies that the insertions can only take place either at the start or at the end of a substring of T. This can equivalently be stated as performing an exact match of a substring of T to the longest suffix or prefix of any P in P^S. The fix-constraint thus allows us to cast the problem as an exact pattern suffix or prefix matching problem, rather than an approximate matching problem.

1) Examples: Similarly consider an example where $P^S = \{"findme", "pinpointme"\}$ and T=" "geryointmera". In this case, $d_f = 4$ for both "findme" and "pinpointme". Also for this distance metric, coupling it with a coverage metric can be useful.

2) Algorithmic Aspects: As the problem now can be framed as an exact substring match (of all possible prefixes and suffixes of all patterns in P^S) to a text T, existing efficient algorithms can be utilized. The Aho-Corasick [9] algorithm is suitable for this problem, and it has a complexity of O($N + M + Z$), where Z is the number of matches. The total number of patterns-derived substrings used in the algorithm can be up to $\sum_{i=1}^{V}(2M_i - 1)$, although in practice it will likely be lower as a considerable number of short prefixes and suffixes with low coverage are expected not to be relevant in many application settings.

IV. Security Applications

In this section, we discuss security related applications of the *slice distance*. In general, all use-cases of the Levenshtein distance can benefit from the *slice distance* in situations in which the error can only result from missing information.

The distance metrics defined above can be useful in a variety of security-related contexts. We focus in particular on distributed intrusion detection systems (D-IDSs), which are an integral part of the efforts to secure networks. Typically, they have a database of known attack signatures that are compared with traffic flows to detect malicious activities. An abstract model of a D-IDS setting is shown in Figure 1. In

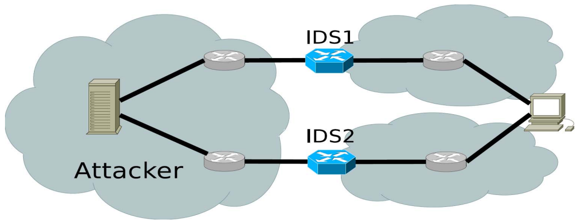

Fig. 1: Abstract attack model.

the figure, there are two separate IDSs and the traffic may flow along any or both of the two possible paths. Each of the IDSs performs its function by comparing the passing traffic with a known database of attacks using different string searching algorithms. Such a set-up has functioned in detecting intrusions over the years as the outcome of the matching is either a perfect match or no match. However, protocols such as Multipath TCP (MPTCP) [11] and CMT-SCTP [10] can create problems for such exact matchings in the D-IDSs [12]. Using MPTCP, an attacker has the possibility to manipulate which data are transferred over which path and thus distribute the application data. An IDS on one of those paths may thus only have access to partial traffic where the attacker has fragmented the data containing the signature. Recent work [13] addressed the issue by holding joint state and exchanging frequent update messages. We consider the problem in its most general formulation, without any assumptions on timely inter-IDS communication.

The problem is thus for the IDS to decide whether a string (possibly with missing characters) in the traffic fragment matches a signature string in its database. The general Levenshtein distance is not the most appropriate in this case as it does not take into consideration the fact that the string matching distance should in this case not allow for deletions, substitutions, and transpositions, but only insertions. From the point of view of an IDS inspecting incoming traffic, the splitting of traffic to several paths cannot give rise to insertions or substitutions of the in-message data that should be matched at the inspecting path. Hence, to counter the character(s) or byte(s) transferred on another path, insertions are necessary.

The proposed metrics can be used by D-IDSs in comparing input strings to the pattern strings in their databases and performing signature matching. Depending on the assumptions made regarding the attackers ability to manipulate the data transfer, and the availability of sequence number meta-data for the traffic, different *slice distance* variants are appropriate for matching. In general, it can be observed that if M is the length of a pattern string in the database and L is the number of paths among which a message can be distributed then for the IDS that sees the most of the message, at least $\lceil M/L \rceil$ bytes will match if a message containing the string in database has indeed been transferred across the paths. This is true regardless of how the traffic is distributed across the paths. This minimum detect length of bytes will be present at least at one IDS regardless of whether the input string is split into L continuous blocks or every Lth byte of the string is sent across each path. The maximum allowable *slice distance* for a match will be M-$\lceil M/L \rceil$. This minimum bound on observed traffic can be used by the D-IDSs to overcome the difficulty of detecting intrusions when matching fragmented traffic. However, this comes at the cost of increased false positives, which is an issue we aim to explore in our future work.

V. Concluding Remarks

The well known Levenshtein distance [3] is commonly used in many string matching applications today. For performance optimization in various problem domains, a number of variations to the general Levenshtein distance have been suggested [1], [4]–[6]. However, a restricted distance metric that only considers insert operations is missing. To this end, we proposed *slice distance* as a variation of the Levenshtein distance to compare strings and correct errors resulting from missing bytes. Four variations of the *slice distance* are presented. We also provided a discussion on the algorithmic aspects of these metrics and considered a particular application for the *slice distance* in the form of string matching in IDSs. The code of the prefix slice distance is freely available[1].

Acknowledgment

The work is being carried out in the High Quality Networked Services in a Mobile World project funded partly by the Knowledge Foundation of Sweden.

References

[1] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[2] S. Kumar and E. H. Spafford, "An application of pattern matching in intrusion detection," *Computer Science Technical Reports, Purdue University*, no. 94-013, 1994.

[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, pp. 707–710, 1966.

[4] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[5] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage," in *Proceedings of the Section on Survey Research*, 1990, pp. 354–359.

[6] S. V. Petrovic and J. D. Golic, "String editing under a combination of constraints," *Inf. Sci.*, vol. 74, no. 1-2, pp. 151–163, 1993.

[7] F. Damerau, "A technique for computer detection and correction of spelling errors," *Commun. ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[8] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen, "Episode matching," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 1997, pp. 12–27.

[9] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[10] J. R. Iyengar, P. D. Amer, and R. R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 951–964, 2006.

[11] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch, "TCP extensions for multipath operation with multiple addresses," IETF, Experimental RFC 6824, 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-mptcp-rfc6824bis-06

[12] Z. Afzal and S. Lindskog, "Multipath TCP IDS evasion and mitigation," in *Proceedings of the 18th International Conference on Information Security (ISC)*, September 9–11, 2015, pp. 265–282.

[13] J. Ma, F. Le, A. Russo, and J. Lobo, "Detecting distributed signature-based intrusion: The case of multi-path routing attacks," in *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26–May 1, 2015*, 2015, pp. 558–566.

[1]Source code available at https://github.com/randomsecguy/slicedistance.