A Statistical Analysis of Duplication Errors in the Nanopore Sequencing Channel

Sarvin Motamen, Hao Lou, Farzad Farnoud
Electrical and Computer Engineering,
University of Virginia, Charlottesville, VA, USA,
Email: {sarvin,haolou,farzad}@virginia.edu

***Abstract*—A duplication edit, which copies a substring and inserts the copy immediately after the substring itself, is a type of error in communication systems and DNA storage. In this work, we devise an algorithm for computing the duplication distance, i.e., the minimum cost of transforming one string to another using insertions, deletions, substitutions, duplications and deduplications, where each operation is assigned a weight determining its contribution to the cost. With the help of this algorithm, we perform a statistical analysis of simulated Nanopore data to determine whether duplication and deduplication edits are prevalent in the Nanopore sequencing channel. Our results indicate a positive answer to this question.**

I. Introduction

A *tandem duplication*, or simply a duplication, is a type of string edit operation where a substring (the template) is copied and the copy is inserted immediately after the template in the string. The duplication channel, which was studied in [1] for modeling timing errors in communication systems, has received increasing attention, in part due to advances in DNA data storage. A number of works [2]–[7] have been devoted to developing codes for correcting duplication error.

As in most existing works, the prevalence of duplication errors is assumed and there is a lack of analysis to verify the presence of duplication errors based on data. In this work, we aim to address this shortcoming by performing a statistical analysis on errors that occur during DNA sequencing. Our objective is to investigate the prevalence of duplication errors. To achieve this, we analyze whether incorporating duplication and deduplication edits provides a better explanation for observed data as compared to solely allowing point edits, i.e., insertions, deletions, and substitutions.

We first devise an algorithm for computing an edit distance that allows duplication and deduplication operations, in addition to insertions, deletions, and substitutions. We refer to this distance as the duplication distance (even though other operations are also permitted). To make computing the distance feasible, however, we do not allow all duplication and point edits, as described in the next section. In particular, only duplications of length 1 are permitted.

We apply the algorithm to simulated Nanopore DNA sequencing data to determine the duplication distance between the input and output strings of the sequencing process. A statistical test is designed and performed to evaluate the hypothesis that duplication errors are present, which consists of comparing the duplication distance of actual input-output pairs with input-output pairs with the output being random but at the same Levenshtein distance from the input. Our statistical test strongly suggests that duplication edits are prevalent and that including them in the set of error types better explains edits encountered in DNA sequencing.

The rest of the paper is organized as follows. The duplication distance algorithm is presented in Section II. The data analysis and its results are presented in Section III. The proof of correctness of the algorithm is in Section IV and the conclusion in Section V. The source code of our implementation of the algorithm and data analysis is available online[1].

II. Algorithm for Computing the Duplication Distance

Edit distance is a measure of dissimilarity of two strings, obtained by evaluating the set of operations needed to transform one string into the other. Depending on what operations are permitted, different versions of edit distance can be defined. The most common edit distance, called the *Levenshtein distance*, permits substitutions, insertions, and deletions of single characters. For example, the Levenshtein distance between 'hare' and 'shark' is two, via an insertion and a substitution: hare $\rightarrow$ share $\rightarrow$ shark. Note also that 'shark' can be transformed into 'hare' with a substitution and a deletion. We denote the Levenshtein distance by $L(\cdot,\cdot)$. For our preceding example, $L(\text{hare},\text{shark}) = 2$.

Edit distance can be used to evaluate errors in a communication or storage channel. Specifically, edit distance provides a lower bound on the number of errors that have occurred to transform the input of the channel to its output. Assuming that errors are not too likely, this lower bound can also act as an approximation. Additionally, a code that can correct that many errors will be able to recover the channel input. In this paper, to evaluate the presence of duplication and deduplication errors, we study the duplication distance, which also allows duplications and deduplications, and is denoted by

This work was supported in part by the following NSF grants: CIF 1816409 and CAREER 2144974.

[1] https://github.com/SarvMotamen/Edit-Distance-Codes

$\delta(\cdot,\cdot)$. The set of permitted operations is described in detail in the next subsection.

We use capital letters to denote strings, e.g., A. For a string A, we use A_i to denote its ith character, and use A_i^j to denote the substring $A_iA_{i+1}\cdots A_{j-1}A_j$. We use lowercase letters to denote characters from the alphabet.

A. Operations and Weights

In this paper, in addition to substitutions, insertions, and deletions, we allow duplications and deduplications of single characters. Specifically, in a *duplication* operation, a character may be replaced by two copies, e.g., $abc \to abbc$, and in a *deduplication* a pair of adjacent characters may be replaced by a single occurrence, e.g., $aabc \to abc$.

In general, when defining an edit distance, each operation may be assigned a weight. The distance is then the smallest number s such that there is a sequence of operations transforming one string into the other where the sum of the weights of the operations in the sequence is s. We denote the weights of substitutions, insertions, deletions, duplications, and deduplications with $w_{sub}, w_{in}, w_{del}, w_{dup}$, and w_{ded}, respectively. To ensure that our measure of dissimilarity is symmetric (as required from any metric), we assume $w_{in} = w_{del}$ and $w_{dup} = w_{ded}$.

We also assume that $w_{del} < w_{ded} + w_{sub}$, as otherwise each deletion can be replaced with a substitution followed by a deduplication. Similarly, we need $w_{in} < w_{dup}+w_{sub}$, $w_{dup} < w_{in}$ and $w_{ded} < w_{del}$ to have insertions, duplications and deduplications respectively.

We note that in general, a duplication may copy more than one character and similarly a deduplication may remove more than one character. The number of characters added or removed is the length of the (de)duplication operation. For example, a duplication of length 3 may be as follows: $abcde \to abcd\underline{bcd}e$, where the duplicated segment is underlined. Allowing operations of length more than 1, however, significantly increases the complexity of the problem. And any algorithm capable of finding the distance with longer duplications would have a high time complexity. Hence, for simplicity, we limit our attention to duplications of length 1.

Furthermore, we assume that no operation can be performed between two duplicated or two deduplicated characters. For example, if we duplicate a to produce aa, we cannot then insert a character between the two copies of a to produce, e.g., aba. The reverse operation is also invalid, meaning we cannot take aba and delete b to get aa, and then deduplicate to obtain a. We also make the assumption that after a character is duplicated, neither of the two resulting characters can be substituted, and a substituted character cannot be deduplicated.

B. Algorithm

Similar to the edit distance algorithm introduced in [8], our duplication distance algorithm is dynamic programming-based, i.e., it computes the duplication distance based on the duplication distances between prefixes. We demonstrate the general recursion steps in the following. The proof of correctness of the algorithm can be found in Section IV.

Given strings A and B of lengths $|A|$ and $|B|$, respectively, let $d(i,j)$ be the edit distance between the prefix of A of length i, i.e., A_1^i, and the prefix of B of length j, i.e., B_1^j.

We now describe how $d(i,j)$, $1 \le i \le |A|$ and $1 \le j \le |B|$, can be found from the set of distances

$$\{d(i',j') : i' \le i, j' \le j, (i',j') \ne (i,j)\}.$$

First, we initialize $d(0,0) = 0$, since the distance between two empty strings is always 0. For (i,j) where $0 \le i \le |A|$, $0 \le j \le |B|$, and $(i,j) \ne (0,0)$, there are multiple scenarios to consider. For instance, if $A_i = B_j$, then the distance between A_1^i and B_1^j can be the same as the distance between A_1^{i-1} and B_1^{j-1}, i.e., $d(i,j) = d(i-1,j-1)$. Or, when $A_i \ne B_j$, A_i may be replaced by B_j and $d(i,j)$ can be equal to $d(i-1,j-1)+w_{sub}$. These and other possibilities are considered below, and the optimal distance $d(i,j)$ is the minimum of all these values:

$$d(i,j) = \min \begin{cases} d(i-1,j-1) & i,j>0, A_i = B_j \\ d(i-1,j-1)+w_{sub} & i,j>0, A_i \ne B_j \\ d(i,j-1)+w_{dup} & j>1, B_{j-1} = B_j \\ d(i-1,j)+w_{ded} & i>1, A_{i-1} = A_i \\ d(i,j-1)+w_{in} & j>0 \\ d(i-1,j)+w_{del} & i>0 \end{cases} \tag{1}$$

Therefore, the algorithm consists of two for loops, meaning, for $0 \le i \le |A|$, for $0 \le j \le |B|$ where $(i,j) \ne (0,0)$, $d(i,j)$ is computed. We denote the duplication distance between two strings A, B as $\delta(A,B) = d(|A|,|B|)$.

Under the assumption that accessing the data structure takes constant time, updating each $d(i,j)$ has time complexity of $O(1)$. Therefore, the algorithm has time complexity $O(|A||B|)$.

III. Data Analysis

In this section, we evaluate the presence of duplications among the errors observed in Nanopore sequencing [9] in DNA data storage. To do so, we analyze a dataset consisting of 1000 input-output pairs. The input represents the (true) base sequence of a DNA molecule (we call this string A). Each input string is of length 200, with bases randomly generated from $\{A,C,G,T\}$. The output (string B) represents the output of sequencing this molecule using a Nanopore sequencer, as simulated by the Nanopore Deep Simulator [10] and the Guppy basecaller [11].

Our goal is to statistically test whether duplications and deduplications are a significant source of errors in Nanopore sequencing. We do so by comparing the duplication distance $\delta(A,B)$ of the input-output pairs (A,B) with the duplication distance $\delta(A,B')$ for a randomly generated string B' such that $L(A,B') = L(A,B)$. The idea behind this statistical analysis is that in the absense of duplication and deduplication errors, the output B behaves no differently in terms of the duplication

distance from A compared to a randomly-generated string with the same Levenshtein edit distance from the input A.

The first step is to generate a random string (denoted B') for each input string A with length $|A|$, where $L(A,B) = L(A,B')$. In addition, we require the number of operations of each type (insertion, deletion, and substitution) needed to transform A to B be the same as those needed to transform A to B'. In order to generate such a string, we first find $\ell = L(A,B)$ and n_{in}, n_{del}, n_{sub}, i.e., the number of insertions, deletions, and substitutions associated with that distance ($\ell = n_{in} + n_{del} + n_{sub}$). Next, we produce a sequence of edits $S = (s_1, s_2, ..., s_\ell)$ with n_{in} insertions, n_{del} deletions, and n_{sub} substitutions, where the order of the operations is random. For example, if $n_{in} = 2$, $n_{del} = 1$, $n_{sub} = 2$, a possible sequence of edits would be $S = (sub, del, in, in, sub)$.

We then generate a set of ℓ random positions $P = \{p_1, p_2, ..., p_\ell\}$ where for $1 \le i \le \ell$, $p_i \le |A|$. After sorting P in descending order, we perform edit s_i in position p_i for all $1 \le i \le \ell$ to obtain a candidate string B'. Let $\ell' = L(A,B')$ and let the number of edits of each type corresponding to ℓ' be n'_{in}, n'_{del}, and n'_{sub}. If the equality $(\ell', n'_{in}, n'_{del}, n'_{sub}) = (\ell, n_{in}, n_{del}, n_{sub})$ is not satisfied, we discard this candidate for B' and generate another candidate, repeating the process until the equality is satisfied.

With this process, B' could be found for 973 out of 1000 input strings in a reasonable time (at most 200 strings were generated for each input A).

The next step is to calculate the duplication distance between the input A and the two outputs B, B'. We denote $\delta_t = \delta(A,B)$ and $\delta_r = \delta(A,B')$. Our null hypothesis is that the events $\delta_r > \delta_t$ and $\delta_r < \delta_t$ have equal probability, i.e., $p = 0.5$ (assuming that in the case of $\delta_r = \delta_t$, the string is put in one of the other two groups with probability 0.5).

For the 973 triples (A, B, B'), we compute δ_r and δ_t, and categorized them into three groups: strings where $\delta_r > \delta_t$, $\delta_r = \delta_t$, and $\delta_r < \delta_t$. For calculating the duplication distance, we consider the weights to satisfy $w_{in} = w_{del} = w_{sub} = 1, w_{dup} = w_{ded} = w < 1$, and conduct the test for 3 values of $w = 0.1, 0.5, 0.9$. The results are shown in Table I.

TABLE I
δ_r VS. δ_t FOR 973 GUPPY SEQUENCES. THE TEST WAS CONDUCTED FOR 3 DIFFERENT VALUES OF w, AS WRITTEN IN THE FIRST COLUMN.

w	$\delta_r > \delta_t$	$\delta_r = \delta_t$	$\delta_r < \delta_t$
0.1	655	35	283
0.5	636	108	229
0.9	629	112	232

We observe for all values of w, in the majority of the cases, the duplication distance for the output B is smaller than the random string B'. This suggests that duplications and deduplications played a part in producing B.

More formally, based on the null hypothesis, the number of strings in each of the two groups is a Binomial random variable with $n = 973$ and $p = 0.5$. As a result, under the null hypothesis, one would expect to see roughly equal numbers for $\delta_t < \delta_r$ and $\delta_t > \delta_r$. The null hypothesis can be rejected with p-value $< 10^{-8}$. This implies that the fact that $\delta_t < \delta_r$ appears more often is not the result of random chance. Our alternative explanation is that some of the insertions and deletions needed to achieve the Levenshtein distance are in fact duplications and deduplications, resulting in reduced distance when (de)duplications are allowed with a cost lower than that of insertions and deletions. In other words, our results suggest that in the Nanopore dataset, the number of duplication and deduplication edits are more than a randomly-generated string, which implies the significance of duplication and deduplication edits in the Nanopore sequencing channel.

IV. ALGORITHM'S PROOF OF CORRECTNESS

In this section, we prove the correctness of the algorithm for computing the duplication distance. At a high level, our proof follows that of Wagner and Fischer [8] for the Levenshtein edit distance algorithm. However, the introduction of duplications and deduplications substantially increases the complexity of the problems, including requiring modifying the *trace* structure and its properties as well as a comprehensive analysis of various cases that may occur in the execution of the algorithm.

In subsection IV-A, the notation used in the proof is provided. Subsection IV-B defines the *trace* structure, which is then used to prove the correctness of (1) in subsection IV-C.

A. Notation

For a sequence of edits $S = s_1 s_2 ... s_m$, let $\gamma(S) = \sum_{i=1}^{m} w(s_i)$ be the sum of the weights of all edits in S, with possible edits being point edits and (de)duplications respecting the conditions in II-A. For two strings A and B, define

$$\delta(A,B) = \min\{\gamma(S) : S \text{ is an edit sequence taking } A \text{ to } B\}.$$

For $1 \le i \le |A|$ and $1 \le j \le |B|$, let $d(i,j) = \delta(A_1^i, B_1^j)$. Also, we define $d(0,j) = \delta(\varnothing, B_1^j)$, $d(i,0) = \delta(A_1^i, \varnothing)$, and $d(0,0) = \delta(\varnothing, \varnothing)$, with $\varnothing$ being the empty string.

B. Trace

Consider a graph G with a vertex set consisting of $|A|+|B|$ vertices, namely, the characters of A and B, arranged in two lines, as shown in Fig. 1. The edge set of this graph is denoted by $T = (T_{AB}, T_A, T_B)$, where T_{AB} is the set of edges between A and B, while the edges in T_A are of the form (A_i, A_{i+1}), and similarly for T_B. Let $G_A = (A, T_A)$ and $G_B = (B, T_B)$ denote subgraphs of G. We refer to the set T as a *trace from string A to B* if it satisfies the following properties:

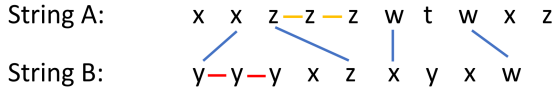

Fig. 1. An example of a trace for $A = xxzzzwtwxz$ and $B = yyyxzxyxw$. Edges in T_{AB} are blue, T_A yellow, and T_B red.

1) No two edges of T_{AB} share an endpoint. Furthermore, no two edges of T_{AB} cross each other, i.e., if $(A_{i_1}, B_{j_1}), (A_{i_2}, B_{j_2}) \in T_{AB}$ and $i_1 < i_2$, then $j_1 < j_2$.

2) Endpoints of edges in G_A are identical characters, i.e., $A_i = A_{i+1}$. Let V_C be the vertices of a component in G_A. There is at most one edge in T_{AB} with an endpoint in V_C, and that endpoint can only be the left-most vertex of V_C. The same properties hold for G_B.

Let I be the set of components in G_A where no vertex in the component has an edge in T_{AB}, and J be the sets of components in G_B where, again, no vertex in the component has an edge in T_{AB}. We define the cost of T as below:

$$\text{cost}(T) = \sum_{(i,j)\in T, A_i \neq B_j} w_{sub} + \sum_{(i,i+1)\in T_A} w_{dup} + \sum_{(j,j+1)\in T_B} w_{ded} + \sum_{i\in I} w_{del} + \sum_{j\in J} w_{in}$$

Theorem 1: For any two strings A and B,

$$\delta(A,B) = \min\{\text{cost}(T) | T \text{ is a trace from } A \text{ to } B\}. \quad (2)$$

Proof: We show that traces have the properties:

(I) For every trace T from A to B, there is an edit sequence S taking A to B such that $\gamma(S) = \text{cost}(T)$.

(II) For every edit sequence S taking A to B, there is a trace T from A to B such that $\text{cost}(T) \leq \gamma(S)$.

In order to prove property (I), we build the edit sequence S such that $\gamma(S) = \text{cost}(T)$ as follows:

Deduplicate A_{i+1} for all $(A_i, A_{i+1}) \in T_A$, delete A_i for all $C \in I$ with left-most vertex A_i, substitute A_i with B_j for $(A_i, B_j) \in T_{AB}$ where $A_i \neq B_j$, insert B_j for every component $C \in J$ with left-most vertex B_j, and duplicate B_j to get B_jB_{j+1} for $(B_j, B_{j+1}) \in T_B$.

Induction is used to build trace T and prove property (II). Assume we have strings A and B and edit sequence $S = s_1, s_2, ..., s_n$ that transforms A into B. Let $A^{(k)}$ be the string after performing edits $S^{(k)} = s_1, ..., s_k$ on A, and $A^{(0)} = A$. The trace $T^{(k)}$ refers to the trace from A to $A^{(k)}$ where $\text{cost}(T^{(k)}) \leq \gamma(S^{(k)})$

As the base case, the trace $T^{(0)}$ from A to $A^{(0)}$ is built as follows: $T^{(0)}_{AB} = \{(A_i, A^{(0)}_j) | i = j, 1 \leq i \leq |A|\}$, $T^{(0)}_A = T^{(0)}_B = \varnothing$. The graph of this trace is a bipartite graph with $|A|$ vertices in each set and an edge between vertices corresponding to the characters in the same position in A and $A^{(0)}$. In this case, $\gamma(S^{(0)}) = \text{cost}(T^{(0)}) = 0$, since for all $(A_i, A^{(0)}_j) \in T^{(0)}_{AB}$, $A_i = A^{(0)}_j$.

Next, we assume that for trace $T^{(k-1)}$ from A to $A^{(k-1)}$, we have $\text{cost}(T^{(k-1)}) \leq \gamma(S^{(k-1)})$, and we will build trace $T^{(k)}$ from A to $A^{(k)}$ where $\text{cost}(T^{(k)}) \leq \gamma(S^{(k)})$.

We first introduce two shift operations on trace T from A to B. The result of these operations are also traces.

Let $\mathcal{S}_{right}(T, A, B, t) = (T'_{AB}, T_A, T'_B)$, where:

$$\begin{aligned} T'_{AB} =& T_{AB} \setminus \{(A_i, B_j) | (A_i, B_j) \in T_{AB}, j > t\} \\ & \cup \{(A_i, B_{j+1}) | (A_i, B_j) \in T_{AB}, j > t\} \\ T'_B =& T_B \setminus \{(B_j, B_{j+1}) | (B_j, B_{j+1}) \in T_B, j > t\} \\ & \cup \{(B_{j+1}, B_{j+2}) | (B_j, B_{j+1}) \in T_B, j > t\} \end{aligned}$$

The $\mathcal{S}_{right}$ operation shifts all edges corresponding to a character in B after B_t to the right by one position.

Let $\mathcal{S}_{left}(T, A, B, t) = (T'_{AB}, T_A, T'_B)$, where:

$$\begin{aligned} T'_{AB} =& T_{AB} \setminus \{(A_i, B_j) | (A_i, B_j) \in T_{AB}, j > t\} \\ & \cup \{(A_i, B_{j-1}) | (A_i, B_j) \in T_{AB}, j > t\} \\ T'_B =& T_B \setminus \{(B_j, B_{j+1}) | (B_j, B_{j+1}) \in T_B, j > t\} \\ & \cup \{(B_{j-1}, B_j) | (B_j, B_{j+1}) \in T_B, j > t\} \end{aligned}$$

Similarly, the $\mathcal{S}_{left}$ operation shifts all edges corresponding to a character in B after B_t to the left by one position.

Edges are added or deleted based on what edit operation s_k is as follows:

1) substitution: $A^{(k-1)}_i$ is being substituted with $A^{(k)}_i$, therefore $\gamma(S^{(k)}) = \gamma(S^{(k-1)}) + w_{sub}$. Note that $A^{(k-1)}_i$ cannot have an edge in $T^{(k-1)}_B$ since no duplicated character can be substituted.
   - $A^{(k-1)}_i$ does not have an edge in $T^{(k-1)}$: $A^{(k-1)}_i$ is associated with an insertion in $T^{(k-1)}$, therefore replacing it with $A^{(k)}_i$ can be considered as an insertion of $A^{(k)}_i$ instead of $A^{(k-1)}_i$ in A, meaning no additional operation is performed, $T^{(k)} = T^{(k-1)}$ and $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) \leq \gamma(S^{(k)})$.
   - $A^{(k-1)}_i$ has an edge $(A_l, A^{(k-1)}_i) \in T^{(k-1)}_{AB}$: Since duplicated characters cannot be substituted, $A^{(k-1)}_i$ does not have an edge in $T^{(k-1)}_B$. We replace $A^{(k-1)}_i$ with $A^{(k)}_i$ and keep the edge $(A_l, A^{(k)}_i) \in T^{(k)}_{AB}$. $\text{cost}(T^{(k-1)})$ increases by at most w_{sub}, which means $\text{cost}(T^{(k)}) \leq \gamma(S^{(k)})$.
2) insertion: $A^{(k)}_{i+1}$ is inserted right after $A^{(k-1)}_i$, and $\gamma(S^{(k)}) = \gamma(S^{(k-1)}) + w_{in}$: in this case, $T^{(k)} = \mathcal{S}_{right}(T^{(k-1)}, A, A^{(k-1)}, i)$. We have no insertion between two duplicated characters, therefore all edges in $T^{(k)}_B$ are still valid, and $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) + w_{in} \leq \gamma(S^{(k)})$.
3) deletion: $A^{(k-1)}_i$ is deleted, and $\gamma(S^{(k)}) = \gamma(S^{(k-1)}) + w_{del}$:
   - $A^{(k-1)}_i$ does not have an edge in $T^{(k-1)}$: $A^{(k-1)}_i$ is associated with an insertion in $T^{(k-1)}$, therefore by deleting it, $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) - w_{in} \leq \gamma(S^{(k)})$.
   - $A^{(k-1)}_i$ has an edge $(A_l, A^{(k-1)}_i) \in T^{(k-1)}_{AB}$ but no edge in $T^{(k-1)}_B$: We delete this edge, which implies the deletion of A_l, and $\text{cost}(T^{(k-1)})$ increases by at most w_{del}, meaning $\text{cost}(T^{(k)}) \leq \gamma(S^{(k)})$.
   - $A^{(k-1)}_i$ has one or both of the edges $(A^{(k-1)}_{i-1}, A^{(k-1)}_i) \in T^{(k-1)}_B$ and $(A^{(k-1)}_i, A^{(k-1)}_{i+1}) \in T^{(k-1)}_B$ but no edge in $T^{(k-1)}_{AB}$: The edge(s) in $T^{(k-1)}_B$ are deleted. If $A^{(k-1)}_i$ has both edges, we add the edge $(A^{(k-1)}_{i-1}, A^{(k-1)}_{i+1})$ to $T^{(k-1)}_B$. In any case, we have one less duplication, therefore $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) - w_{dup} \leq \gamma(S^{(k)})$.
   - $A^{(k-1)}_i$ has edges $(A_l, A^{(k-1)}_i) \in T^{(k-1)}_{AB}$ and $(A^{(k-1)}_i, A^{(k-1)}_{i+1}) \in T^{(k-1)}_B$: Both edges are deleted

and $(A_l, A_{i+1}^{(k+1)})$ is added to $T_{AB}^{(k-1)}$, which, again, implies deleting a duplication, and $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) - w_{dup} \leq \gamma(S^{(k)})$. Note that after deleting $A_i^{(k-1)}$, character $A_{i+1}^{(k-1)}$ becomes the left-most vertex of its component in G_B.

Lastly $T^{(k)} = \mathcal{S}_{left}(T^{(k-1)}, A, A^{(k-1)}, i))$.

4) duplication: $A_i^{(k-1)}$ is duplicated to give us $A_i^{(k)} A_{i+1}^{(k)}$, and $\gamma(S^{(k)}) = \gamma(S^{(k-1)}) + w_{dup}$. First, $T^{(k)} = \mathcal{S}_{right}(T^{(k-1)}, A, A^{(k-1)}, i)$. Then, the edge $(A_i^{(k)}, A_{i+1}^{(k)})$ is added to $T_B^{(k)}$, and we have $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) + w_{dup} \leq \gamma(S^{(k)})$.
5) deduplication: $A_i^{(k-1)} A_{i+1}^{(k-1)}$ is deduplicated to give us $A_i^{(k)}$, and $\gamma(S^{(k)}) = \gamma(S^{(k-1)}) + w_{ded}$.
   - At most one of $A_i^{(k-1)}$ and $A_{i+1}^{(k-1)}$ has an edge in $T_{AB}^{(k-1)}$: At least one character is inserted, we delete that character and all its edges in $T_B^{(k-1)}$ (if it has two edges to $A_{i-1}^{(k-1)}$ and $A_{i+1}^{(k-1)}$, those edges are deleted and the edge $(A_{i-1}^{(k-1)}, A_{i+1}^{(k-1)})$ is added to $T_B^{(k-1)}$), therefore we have $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) - w_{in} \leq \gamma(S^{(k)})$.
   - Both $A_i^{(k-1)}$ and $A_{i+1}^{(k-1)}$ have an edge in $T_{AB}^{(k-1)}$: Since no deletion can happen between two deduplicated characters, $(A_l, A_i^{(k-1)}) \in T^{(k-1)}$ and $(A_{l+1}, A_{i+1}^{(k-1)}) \in T^{(k-1)}$, and since no substituted character can be deduplicated, $A_l = A_i^{(k-1)}$ and $A_{l+1} = A_{i+1}^{(k-1)}$. Here, $(A_{l+1}, A_{i+1}^{(k-1)})$ is deleted from $T_{AB}^{(k-1)}$ and (A_l, A_{l+1}) is added to $T_A^{(k-1)}$. If $A_{i+1}^{(k-1)}$ has an edge in $T_B^{(k-1)}$, it is not with $A_i^{(k-1)}$, since two characters connected with an edge in T_B cannot both have edges in T_{AB}. Therefore, the edge is $(A_{i+1}^{(k-1)}, A_{i+2}^{(k-1)})$, which is deleted and $(A_i^{(k-1)}, A_{i+2}^{(k-1)})$ is added to $T_B^{(k-1)}$. In this case, $\text{cost}(T^{(k)}) = \text{cost}(T^{(k-1)}) + w_{ded} \leq \gamma(S^{(k)})$.

   Lastly, $T^{(k)} = \mathcal{S}_{left}(T^{(k-1)}, A, A^{(k-1)}, i+1))$

This completes the proof of property (II). From properties (I) and (II) of traces, equation 2 follows. ∎

C. Proof of Correctness

Theorem 2: Consider two string A and B and let $d(0,0) = 0$. Furthermore, for $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$ where $(i,j) \neq (0,0)$, let $d(i,j)$ be given by (1). Then, $d(i,j) = \delta(A_1^i, B_1^j)$.

Proof: Let T be a least cost trace from A_1^i to B_1^j. If A_i and B_j both have an edge T_{AB}, they must both be connected to the same edge, since otherwise we have edges $(A_{i_1}, B_{j_1}), (A_{i_2}, B_{j_2}) \in T_{AB}$ where $i_1 < i_2$ and $j_1 > j_2$, which contradicts the definition of trace. Then, at least one of the following cases must hold:

Case 1: $(A_i, B_j) \in T_{AB}$ and $A_i = B_j$: $m_1 = d(i-1, j-1)$.

Case 2: $(A_i, B_j) \in T_{AB}$ and $A_i \neq B_j$: we have a substitution and $m_2 = d(i-1, j-1) + w_{sub}$.

Case 3: $(B_{j-1}, B_j) \in T_B$: then we have a duplication and $m_3 = d(i, j-1) + w_{dup}$.

Case 4: $(A_{i-1}, A_i) \in T_A$: then we have a deduplication and $m_4 = d(i-1, j) + w_{ded}$.

Case 5: $C \in J$ where the left-most vertex is B_j: then we have an insertion and $m_5 = d(i, j-1) + w_{in}$

Case 6: $C \in I$ where the left-most vertex is A_i: then we have a deletion and $m_6 = d(i-1, j) + w_{del}$

Since one of the six cases above must hold, $d(i,j) = \min(m_{1-6})$. ∎

The preceding theorem implies that $\delta(A, B) = d(|A|, |B|)$, thereby establishing the correctness of the proposed algorithm.

V. Conclusion

In this paper, we defined duplication distance as the distance between two strings where duplication and deduplication edits of a single character subject to specified conditions are allowed along with insertion, deletion, and substitution. Using dynamic programming, we developed an algorithm for computing this distance, and used it to show that duplication and deduplication edits are prevalent in the Nanopore sequencing channel.

Directions of interest for future work include developing an algorithm for computing the distance when (de)duplications of length larger than 1 are allowed, as well as eliminating other limitations that we enforced for the sake of simplicity. The analysis can also be expanded from Nanopore sequencing to other channels including DNA synthesis and channels with timing errors, and to real data compared to simulated data. Such analyses can enable designing more capable error-correcting codes with reduced redundancy.

References

[1] L. Dolecek and V. Anantharam, "Repetition error correcting sets: Explicit constructions and prefixing methods," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 4, pp. 2120–2146, 2010.

[2] S. Jain, F. Farnoud, M. Schwartz, and J. Bruck, "Duplication-correcting codes for data storage in the dna of living organisms," *IEEE Transactions on Information Theory*, vol. 63, no. 8, pp. 4996–5010, 2017.

[3] Y. M. Chee, J. Chrisnata, H. M. Kiah, and T. T. Nguyen, "Deciding the confusability of words under tandem repeats in linear time," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 3, pp. 1–22, 2019.

[4] M. Kovačević and V. Y. Tan, "Asymptotically optimal codes correcting fixed-length duplication errors in dna storage systems," *IEEE Communications Letters*, vol. 22, no. 11, pp. 2194–2197, 2018.

[5] A. Lenz, A. Wachter-Zeh, and E. Yaakobi, "Duplication-correcting codes," *Designs, Codes and Cryptography*, vol. 87, pp. 277–298, 2019.

[6] Y. Tang, Y. Yehezkeally, M. Schwartz, and F. Farnoud, "Single-error detection and correction for duplication and substitution channels," *IEEE Transactions on Information Theory*, vol. 66, no. 11, pp. 6908–6919, 2020.

[7] Y. Tang and F. Farnoud, "Error-correcting codes for noisy duplication channels," *IEEE Transactions on Information Theory*, vol. 67, no. 6, pp. 3452–3463, 2021.

[8] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[9] D. Deamer, M. Akeson, and D. Branton, "Three decades of nanopore sequencing," *Nature Biotechnology*, vol. 34, no. 5, pp. 518–524, May 2016.

[10] Y. Li, S. Wang, C. Bi, Z. Qiu, M. Li, and X. Gao, "Deepsimulator1. 5: a more powerful, quicker and lighter simulator for nanopore sequencing," *Bioinformatics*, vol. 36, no. 8, pp. 2578–2580, 2020.

[11] "Guppy software overview," https://community.nanoporetech.com/docs/prepare/library_prep_protocols/Guppy-protocol/v/gpb_2003_v1_revax_14dec2018/guppy-software-overview, accessed: 12/5/2023.