Efficient Search of microDNA Reintegration in DNA Sequences

Yiming Wang, Farzad Farnoud and Pankaj Kumar

Abstract

In recent years, the study of microDNAs, a class of extrachromosomal circular DNA, has yielded promising results. MicroDNAs have been detected in human cells, with an increased level of their presence in cancer cells [1]. Biology research shows that microDNAs likely comes from genomic regions with active chromatin marks [2]. An interesting question to in microDNA research is: do microDNAs reintegrate back into the genome, and if so, can we locate the reintegration patterns? In this work, we establish mathematical definitions of microDNA reintegration and propose an efficient search algorithm. We find hundreds or thousands reintegration patterns on chromosome-Y, chromosome-22 and chromosome-21. Our simulation shows that the patterns found by the algorithm are not purely a result of inherent repetitive structure in the genome, indicating those patterns are evidence that microDNAs do reintegrate back into the genome.

1. Introduction

MicroDNAs are a class of extrachromosomal circular DNA (eccDNA), with length less than 1000 bps [2]. Research in microDNAs has yielded interesting results. For example, an increased level of microDNAs were detected in cancer cells [1]. MicroDNAs are shown to likely originate from genomic regions with active chromatin marks [2]; an interesting question in the research of microDNA is: do microDNAs reintegrate back to chromosomes, and if so, can we locate them? Tracking microDNAs and looking for reintegration pattern through biology experiments would likely be costly and time-consuming to perform. Thus, in this paper, we try to answer this question through a computational approach by modeling the reintegration process and designing a search algorithm that locates such patterns.

From a high level, we simplify the reintegration process as the following steps: (1) a segment of the genome gets copied and forms either single-stranded or double-stranded circular structure microDNA by connecting its head and tail, (2) the circular structure was split in the middle, flattened, and became linear, (3) the flattened sequence was inserted back into the genome, with same direction as the original sequence.

More specifically, let $\boldsymbol{a} = a_1a_2...a_k$ (where $a_i \in \{A, T, C, G\}\ \forall i \in [1, k]$) be a segment from original sequence and got copied outside of genome; this copy of $\boldsymbol{a}$ forms the circular structure by connecting its head $\boldsymbol{a}_1$ and tail $\boldsymbol{a}_k$; the circular structure is then cut between a_i and a_{i+1} for some $i \in [1, k-1]$, flattened into a linear string $a_{i+1}a_{i+2}...a_ka_1a_2...a_i$, and inserted back into the sequence. Note the reversion of string $a_{i+1}a_{i+2}...a_ka_1a_2...a_i$ is not considered because its direction is not the same as the original sequence. For double-stranded microDNAs, the reintegration process can similarly happen on the reversed complement strand, resulting the insertion of string $a_i^ca_{i-1}^c...a_1^ca_{k-1}^c...a_{i+1}^c$ back into the genome. The processes for these two types of reintegration are illustrated in the figure 1 and 2.

Note if we set $\boldsymbol{s}_1 = a_1...a_i$ and $\boldsymbol{s}_2 = a_{i+1}...a_k$, the reintegration would be summarized as a two-segment pattern $...\boldsymbol{s}_1\boldsymbol{s}_2...\boldsymbol{s}_2\boldsymbol{s}_1...$, or $...\boldsymbol{s}_1\boldsymbol{s}_2...\boldsymbol{s}_1'\boldsymbol{s}_2'...$ for the reintegration of reversed complement strand. One of these two segments is the part of genome that forms microDNA; the other segment is the reintegration of this microDNA after it was split into linear structure. In section 2, we will formulate a more rigorous mathematical definition of microDNA reintegration pattern that takes mismatch into account.

In this work, we propose an efficient algorithm for locating all circular repeats with the pattern described above, with a mild condition on length within a sequence. Furthermore, our algorithm allows mismatch between the repeated sequences given the fact that circular repeats in genome might have gone through mutations. The algorithm in this paper offers the following contribution: (i) it efficiently searches direct and inverted circular repeats with linear space complexity with respect to input sequence length, (ii) it offers flexibility of searching microDNAs with varying lengths, (iii) it allows dividing the search for a long sequence into a set of smaller tasks, thereby overcoming the memory bottleneck and allowing parallelization.

The rest of the paper is organized as follows. In section II, we introduce necessary notation and definitions. We give a detailed description and analysis of our reintegration pair searching algorithm in Section III. Section IV contains experiment and simulation results. We close the paper with concluding remarks in Section V.

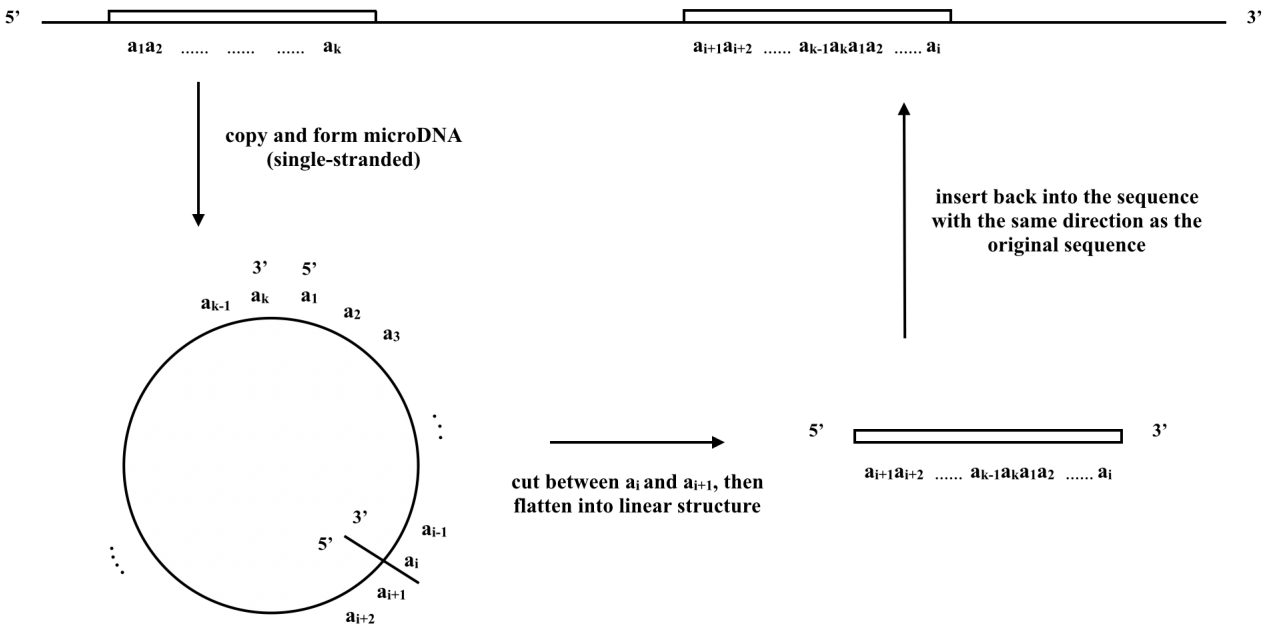

Fig. 1: direct microDNA reintegration

.

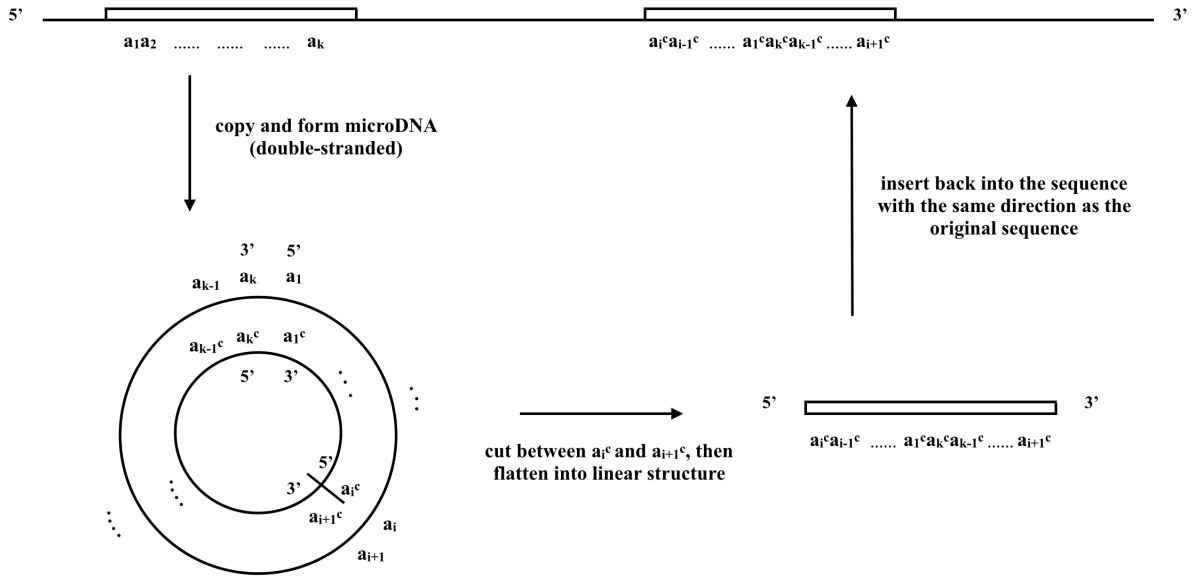

Fig. 2: inverted microDNA reintegration for reversed complement strand

.

2. Definitions and Notation

A. Preliminaries

We denote the alphabet of DNA sequences $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$ by Σ and denote the set of all sequences of length k over Σ by Σ^k. Let $\Sigma^* = \cup_{k=0}^{\infty}\Sigma^k$ be the set of all words over Σ including the empty word of length 0. Vectors and strings are denoted by boldface letters such as $\boldsymbol{x}$, while scalars and symbols by normal letters, such as x. For any $\boldsymbol{w}, \boldsymbol{w}' \in \Sigma^*$, the concatenation of $\boldsymbol{w}$ and $\boldsymbol{w}'$ is denoted $\boldsymbol{w}\boldsymbol{w}'$. The length of $\boldsymbol{w}$ is denoted $|\boldsymbol{w}|$. We shall also use the multiplicative writing $\boldsymbol{w} = w_1 \cdots w_{|\boldsymbol{w}|}$ with w_i denoting the i-th symbol in $\boldsymbol{w}$. For $1 \le i \le j \le |\boldsymbol{w}|$, $\boldsymbol{w}_i^j$ denotes the substring $w_i w_{i+1} \cdots w_j$. Denote the set of all substrings of $\boldsymbol{w}$ by $\mathcal{S}(\boldsymbol{w})$. An ℓ-(sub)string is a (sub)string of length ℓ. The Levenshtein distance [3], or edit distance, between $\boldsymbol{w}$ and $\boldsymbol{w}'$ is denoted $d(\boldsymbol{w}, \boldsymbol{w}')$.

We next review some definitions that help us to present our algorithm more efficiently.

Definition 1. *For some $x \in \Sigma$, the complement of x, denoted x^c, is given by:*

$$x^c = \begin{cases} \mathsf{T}, & \text{if } x = \mathsf{A}, \\ \mathsf{A}, & \text{if } x = \mathsf{T}, \\ \mathsf{G}, & \text{if } x = \mathsf{C}, \\ \mathsf{C}, & \text{if } x = \mathsf{G}. \end{cases}$$

Moreover, for any sequence $\boldsymbol{s} \in \Sigma^*$ with $\boldsymbol{s} = s_1 s_2 \cdots s_m$, the reverse complement of $\boldsymbol{s}$, denoted $\boldsymbol{s}'$, is the sequence $s_m^c s_{m-1}^c \cdots s_1^c$.

Definition 2. *For any $\boldsymbol{s} \in \Sigma^*$, a pair of substrings of $\boldsymbol{s}$ with the same length, we say $\boldsymbol{s}_m^n$ and $\boldsymbol{s}_p^q$, is a maximal repeated pair if $\boldsymbol{s}_m^n = \boldsymbol{s}_p^q$ and $s_{m-1} \neq s_{p-1}, s_{n+1} \neq s_{q+1}$.*

Definition 3. *For any $\boldsymbol{s} \in \Sigma^*$, a pair of substrings of $\boldsymbol{s}$ with the same length, we say $\boldsymbol{s}_m^n$ and $\boldsymbol{s}_p^q$, is a maximal inverted pair if $\boldsymbol{s}_m^n = \boldsymbol{s}_p^{q\prime}$ and $s_{m-1} \neq s_{q+1}^c, s_{n+1} \neq s_{p-1}^c$.*

B. Problem Definition

In the following, we present our model for circular repeat searching problem and formally provide the input the output of our algorithm. Fix $\boldsymbol{s}$ to be the sequence in which we want to locate circular repeats, we know that circular repeats appears in the form of either $...\boldsymbol{s}_1\boldsymbol{s}_2.....\boldsymbol{s}_2\boldsymbol{s}_1...$ or $...\boldsymbol{s}_1\boldsymbol{s}_2.....\boldsymbol{s}_1'\boldsymbol{s}_2'....$ Based on the fact that mutations are common in genomes, it is very rare to see a pair of perfect circular repeated sequences in genome. Therefore, the first s_1 (resp. s_2) and the second s_1 (resp. s_2) typically only match approximately, not exactly. Thus, we redefine the problem as locating microDNA reintegration with form of (1) $...s_{1a}s_{2a}...s_{2b}s_{1b}...$ or (2) $...s_{1a}s_{2a}...s_{1b}'s_{2b}'...$, with mismatches between (s_{1a}, s_{1b}), (s_{2a}, s_{2b}) under predefined threshold. Let p denote the maximum allowed total mismatch, then the microDNA reintegration of form (1) or (2) must also satisfy: $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) + d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b}) < p$. In practice, p are usually determined by a mismatch ratio and the length of reintegration. The following definition formally summarizes the microDNA reintegration pattern.

Definition 4. *For some non-negative number p, a pair of strings $(\boldsymbol{s}_a, \boldsymbol{s}_b) \in \Sigma^* \times \Sigma^*$ is a microDNA reintegration pattern, or circular repeated pair, within distances p_1 and p_2 if there exists $\boldsymbol{s}_{1a}, \boldsymbol{s}_{2a}, \boldsymbol{s}_{1b}, \boldsymbol{s}_{2b} \in \Sigma^*$ s.t. $\boldsymbol{s}_{1a}\boldsymbol{s}_{2a} = \boldsymbol{s}_a$, $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) + d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b}) \le p$, and one of the following conditions is satisfied:*

1) $\boldsymbol{s}_b = \boldsymbol{s}_{2b}\boldsymbol{s}_{1b}$,
2) $\boldsymbol{s}_b = \boldsymbol{s}_{1b}'\boldsymbol{s}_{2b}'$.

In addition, patterns satisfied condition (1) are called direct and patterns satisfied condition (2) are called inverted. In the rest of the paper, we sometimes use the term pattern instead of microDNA reintegration pattern for simplicity.

Example. *Let $\boldsymbol{x} = \mathsf{AGGTC}, \boldsymbol{y} = \mathsf{TCAGG}, \boldsymbol{z} = \mathsf{GACCT}$, then $(\boldsymbol{x}, \boldsymbol{y})$ is a direct pattern within distance 0 and $(\boldsymbol{y}, \boldsymbol{z})$ is a inverted pattern within distance 0, with $\boldsymbol{s}_{1a} = \boldsymbol{s}_{1b} = \mathsf{AGG}, \boldsymbol{s}_{2a} = \boldsymbol{s}_{2b} = \mathsf{TC}$.*

3. microDNA Reintegration Pattern Searching Algorithm

First let's consider the problem of searching direct pattern $...\boldsymbol{s}_{1a}s_{2a}...\boldsymbol{s}_{2b}\boldsymbol{s}_{1b}...$ without mismatch. That is, assume $\boldsymbol{s}_{1a} = \boldsymbol{s}_{1b}$ and $\boldsymbol{s}_{2a} = \boldsymbol{s}_{2b}$. This problem can be solved by searching all the maximal repeated pairs using a suffix tree, and check if there are two pairs of maximal repeated pair, $(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b})$ and $(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$, being adjacent to each other in the form of direct pattern.

The problem is more complicated when taking mismatch into account, because $(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b})$ and $(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$ may not be directly found through searching maximal repeated pairs. However, since the total mismatch is limited (assuming p are reasonably chosen), there must exist $(\boldsymbol{s}_{1a\_sub}, \boldsymbol{s}_{1b\_sub})$ and $(\boldsymbol{s}_{2a\_sub}, \boldsymbol{s}_{2b\_sub})$ that are maximal repeated pairs, where $\boldsymbol{s}_{1a\_sub}$, $\boldsymbol{s}_{1b\_sub}$, $\boldsymbol{s}_{2a\_sub}$, $\boldsymbol{s}_{2b\_sub}$ are substrings of $\boldsymbol{s}_{1a}$, $\boldsymbol{s}_{1b}$, $\boldsymbol{s}_{2a}$, $\boldsymbol{s}_{2b}$ respectively. The idea behind our algorithm, illustrated below, is to first find all these maximal repeated pairs of substrings, and see if we can extend them to form direct reintegration.

Our program has two components, *two-layer search* and *extension checking*. The two-layer search component uses suffix tree to find maximal repeated pairs in the sequence and construct a list of candidate reintegration patterns. Next, the extension checking component performs inspection on each candidate pattern to determine whether it is a valid pattern or not. Given the fact that searching for direct patterns is only of minor difference from searching for inverted patterns, we will mostly present our algorithm by illustrating the procedure of searching only direct patterns and comment on the difference at last.

A. Two-layered Search

1) Two-layered Search Outline: Let $\boldsymbol{s}$ be the given DNA sequence in which we want to locate reintegration patterns. We first find all maximal repeated pairs with length at least l_1 in $\boldsymbol{s}$ by building and searching in a suffix tree [4]. Denote the maximal repeated pairs by $(\boldsymbol{s}_{i,1}, \boldsymbol{s}_{i,2}), 1 \leq i \leq d$ (suppose $\boldsymbol{s}_{i,1}$ is to the left of $\boldsymbol{s}_{i,2}$ without loss of generality). After locating all maximal repeats, we search in their neighborhoods for candidate reintegration patterns. Let m be the specified extension size. Denote the m-substring right after $\boldsymbol{s}_{i,1}$ by $\boldsymbol{s}_{i,r}$ and denote the m-substring right before $\boldsymbol{s}_{i,2}$ by $\boldsymbol{s}_{i,l}$. We search for maximal repeated pairs $(\boldsymbol{u}, \boldsymbol{v})$ with minimum length l_2 ($\leq l_1$) in string $\boldsymbol{s}_{i,r}\#\boldsymbol{s}_{i,l}$, such that $\boldsymbol{u}$ left of $\#$ and $\boldsymbol{v}$ right of $\#$. The process of searching for maximal repeated pairs with $\boldsymbol{u}$ being left of $\boldsymbol{s}_{i,1}$, and $\boldsymbol{v}$ being right of $\boldsymbol{s}_{i,2}$ is similar and is omitted for brevity. Note that if $l_1 = l_2$, only one of the neighborhood searching is needed.

The search for inverted reintegration is performed similarly. The difference is that, instead of searching for maximal repeated pairs, we search for maximal inverted pairs. In our current implementation, the search for maximal reverse complement pairs is accomplished by first searching maximal repeated pairs $(\boldsymbol{u}, \boldsymbol{v})$ in string $\boldsymbol{s}\#\boldsymbol{s}'$ with $\boldsymbol{u}$ left of $\#$ and $\boldsymbol{v}$ right of $\#$, and then obtain location $\boldsymbol{v}'$ in $\boldsymbol{s}$ from the location of $\boldsymbol{v}$ in $\boldsymbol{s}'$.

We now have $..\boldsymbol{s}_{i,1}...\boldsymbol{u}.....\boldsymbol{v}...\boldsymbol{s}_{i,2}...$, with $\boldsymbol{s}_{i,1}$, $\boldsymbol{u}$ close to each other, and $\boldsymbol{s}_{i,2}$, $\boldsymbol{v}$ close to each other. By the definition of direct pattern, there should be no space between $\boldsymbol{s}_{i,1}$ (resp. $\boldsymbol{s}_{i,2}$) and $\boldsymbol{u}$ (resp. $\boldsymbol{v}$). Therefore, what left to check is whether we can extend $\boldsymbol{s}_{i,1}$, $\boldsymbol{s}_{i,2}$, $\boldsymbol{u}$, $\boldsymbol{v}$ such that they could have the form of direct reintegration without breaking the mismatch constraint. The details of the extension checking will be discussed in the next section.

2) Parameter Setting: A natural question to ask is: why do we need to set two different minimum lengths when searching for maximal repeated pairs? To answer this question, we need to first consider what role the minimum maximal repeated pair length plays in the algorithm. Consider a maximal repeated pair $(\boldsymbol{s}_a, \boldsymbol{s}_b)$ with length n. Adding a mutation at index $i \in (1, n)$ in $\boldsymbol{s}_b$ will destroy the maximality of the original pair, and split it into two new maximal repeated pairs. Although the exact lengths for newly generated maximal repeated pairs can vary slightly if the mutation is insertion or deletion, for simplicity we assume the two new pairs have lengths $(i-1)$ and $(n-i)$ respectively. Thus, if there are total of x mutations added in $\boldsymbol{s}_b$, a total of $x+1$ new maximal repeated pairs will be created. Let m denote the longest maximal repeated pair length in the newly created (x+1) pairs. If the mutation is evenly distributed, the minimum of $m = \lceil \frac{n-x}{x+1} \rceil$ is attained. In other words, given a maximal repeated pair of length n and total of x mutations, the length of the longest newly formed maximal repeated pair is at least $\lceil \frac{n-x}{x+1} \rceil$.

For a direct reintegration $...\boldsymbol{s}_{1a}\boldsymbol{s}_{2a}...\boldsymbol{s}_{2b}\boldsymbol{s}_{1b}...$, let m_1 (resp. m_2) denote the longest maximal repeated substring pair length in the pair $(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b})$ (resp. $(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$). We would (ideally) like the minimum length for searching maximal repeated pairs to be smaller than $min(m_1, m_2)$. However, if the lengths of $\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}$ are significantly longer (shorter) than those of $\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b}$, then m_1 will also be significantly longer (shorter) than m_2. For example, let $|\boldsymbol{s}_{1a}| = 200$, $|\boldsymbol{s}_{2a}| = 50$, and 5% mismatch ratio, i.e. $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) = 4$ and $d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b}) = 1$; then, from the lower bound formula above, we would have $m_1 = 49$ and $m_2 = 25$.

In such cases, if we only search maximal repeated pairs once, then the minimum length could be too small to be practical, because the number of maximal repeated pairs explodes when the minimum search length is too low. For example, in our experiment on chromosome-Y (after preprocessing), the number of maximal repeated pairs with length at least 40 is 544030, but this number increased to 9825805, when the minimum length is decreased to 20. Even if we are able to find all maximal repeated pairs with a low threshold, most of the result are useless because we only care about pairs that has another pair in its neighborhood.

Our two-level search overcomes this problem by first searching all the maximal repeated pair in the sequence with a larger length l_1, significantly reducing the total number of maximal repeated pairs we have. Then we perform a second-level search for maximal repeated pairs in the neighborhood of each maximal repeated pair, as described above, with a smaller length l_2. Additionally, the cost of second level search is relatively small, because the time complexity for maximal repeated pair searching is $O(m+k)$, where m is the sequence length and k is number of maximal repeated pairs with length at least l_2, and n,k are both very small compared to the entire input sequence.

B. Extension Checking

Assume we have obtained $..\boldsymbol{s}_{i,1}...\boldsymbol{u}.....\boldsymbol{v}...\boldsymbol{s}_{i,2}...$ from the two-level searching for direct reintegration. Denote the substring between $\boldsymbol{s}_{i,1}$ and $\boldsymbol{u}$ as $\mathbf{A}_2$, the substring between $\boldsymbol{v}$ and $\boldsymbol{s}_{i,2}$ as $\mathbf{B}_2$, the $|\mathbf{B}_2|$-substring immediately left of $\boldsymbol{s}_{i,1}$ as $\mathbf{A}_1$, the $|\mathbf{B}_2|$-substring immediately right of $\boldsymbol{u}$ with as $\mathbf{A}_3$, the $|\mathbf{A}_1|$-substring immediately left of $\boldsymbol{v}$ as $\mathbf{B}_1$, the $|\mathbf{A}_1|$-substring immediately right of $\boldsymbol{s}_{i,2}$ with length as $\mathbf{B}_3$. Now, we have $...\mathbf{A}_1\boldsymbol{s}_{i,1}\mathbf{A}_2\boldsymbol{u}\mathbf{A}_3...\mathbf{B}_1\boldsymbol{v}\mathbf{B}_2\boldsymbol{s}_{i,2}\mathbf{B}_3....$ Note we set the length of $\mathbf{A}_1$, the potential leftward extension of $\boldsymbol{s}_{i,1}$, to be $|\mathbf{B}_2|$ because it's the maximum number of characters $\boldsymbol{s}_{1b}$ can extend leftwards. The same

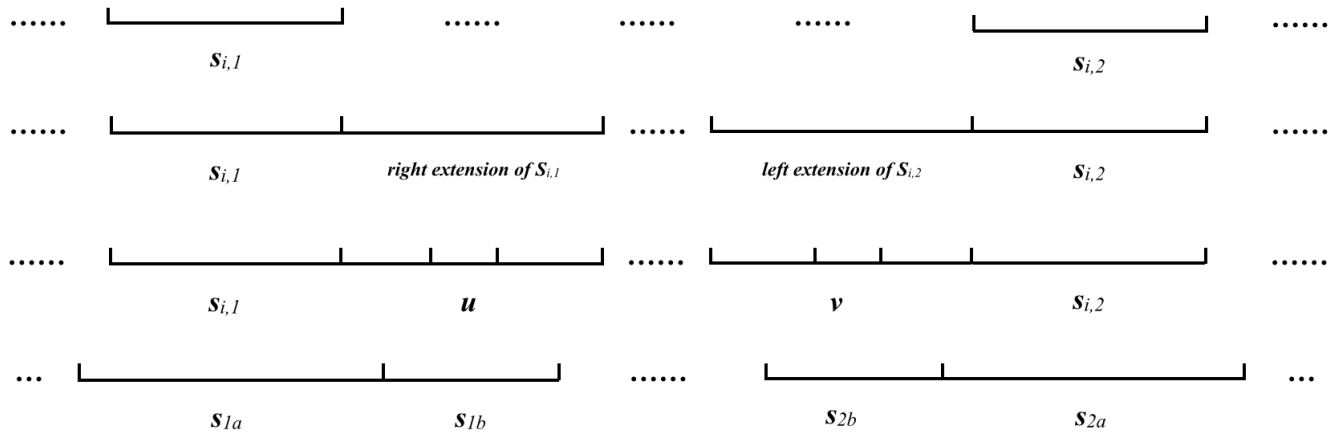

Fig. 3: for a maximal repeat pair $(\boldsymbol{s}_{i,1}, \boldsymbol{s}_{i,2})$, we find maximal repeated pair $(\boldsymbol{u}, \boldsymbol{v})$ in the extensions of $\boldsymbol{s}_{i,1}$ and $\boldsymbol{s}_{i,2}$. We extend $\boldsymbol{s}_{i,1}, \boldsymbol{s}_{i,2}, \boldsymbol{u}, \boldsymbol{v}$ to obtain a direct mircroDNA reintegration, with the sum of edit distances $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) + d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$ minimized .

reason applies to $\mathbf{A}_3$, $\mathbf{B}_1$, $\mathbf{B}_3$. Now the question is, how much should each maximal repeated pair extend such that (1) $\mathbf{A}_2$ and $\mathbf{B}_2$ would be absorbed into the extended segments, and (2) the total edit distance, $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) + d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$, is minimized, where $\boldsymbol{s}_{1a}, \boldsymbol{s}_{2a}, \boldsymbol{s}_{1b}, \boldsymbol{s}_{2b}$ denotes extended version of $\boldsymbol{s}_{i,1}, \boldsymbol{u}, \boldsymbol{s}_{i,2}, \boldsymbol{v}$ respectively.

Suppose we absorb $\mathbf{A}_2$ by extending $\boldsymbol{s}_{i,1}$ rightward for i characters and extending $\boldsymbol{u}$ leftward for $(|\mathbf{A}_2|-i)$ characters. To minimize $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b})$, $\boldsymbol{s}_{i,2}$ should be extended rightward and $\boldsymbol{v}$ should be extended leftward. To simplify computation and implementation, we assume that the right extension of $\boldsymbol{s}_{i,1}$ and left extension of $\boldsymbol{u}$ also have length i and $|\mathbf{A}_2|-i$ respectively. Even though this assumption might not minimize the incurred edit distacn (consider $d(\mathsf{TCG}, \mathsf{TTCG}) < d(\mathsf{TCG}, \mathsf{TTC})$), it would have only limited impact on the result, because for any other extension length $j \neq i$, the edit distance is lowered bounded by $|i - j|$ (number of insertions needed). Therefore, the optimal j must be close to i, and the edit distance we calculated should be close to the minimum.

With this assumption, we extend $\boldsymbol{s}_{i,1}$ rightward for i characters and extend $\boldsymbol{u}$ left for $|\mathbf{A}_2|-i$ characters. The edit distance cost incurred by choosing i for absorbing $\mathbf{A}_2$ is therefore $d(\mathbf{A}_2[:i], \mathbf{B}_3[:i]) + d(\mathbf{A}_2^{-1}[:|\mathbf{A}_2|-i], \mathbf{B}_1^{-1}[:|\mathbf{A}_2|-i])$, where $\boldsymbol{s}[:i]$ denotes the substring of s that starts at beginning and ends at ith character, and $\boldsymbol{s}^{-1}$ denotes the reverse of string s. Note the we need to take the reverse of A_2 and B_3 in the second edit distance calculation, because the extension direction is leftward. Following the same method, we can see that the cost of choosing i for absorbing $\mathbf{B}_2$ is $d(\mathbf{B}_2[:i], \mathbf{A}_3[:i]) + d(\mathbf{B}_2^{-1}[:|\mathbf{B}_2|-i], \mathbf{A}_1^{-1}[:|\mathbf{B}_2|-i])$. Note that there is no overlapping between the required inputs for calculating the cost of choosing ith position at $\mathbf{A}_2$ and $\mathbf{B}_2$, so we can minimize the sum by minimizing each term separately.

To calculate $d(\boldsymbol{s}_1, \boldsymbol{s}_2)$, we construct a 2-D array, X, using dynamic programming, and the last (bottom right) entry in X gives us the edit distance between $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ [3]. The ith diagonal elements of X therefore contains the edit-distance between the pair of substrings $(\boldsymbol{s}_1[:i], \boldsymbol{s}_2[:i])$. Let $d\_vec(\boldsymbol{s}_1, \boldsymbol{s}_2)$ denote the diagonal vector of the 2-D matrix constructed by the edit distance algorithm. Then, choosing the optimal i for $\mathbf{A}_2$ is equivalent as choosing the i that minimizes $d\_vec(\mathbf{A}_2, \mathbf{B}_3)_i + rev\_d\_vec(\mathbf{A}_2^{-1}, \mathbf{B}_1^{-1})_i$, where the prefix $rev\_$ before d_vec indicate we reverse the order of the elements in the vector. Note the reversion of $\mathbf{A}_2$ and $\mathbf{B}_1$ is required because the direction we are comparing them is from right to left.

The extension checking for direct reintegration where $\boldsymbol{s}_{2a}$ is left of $\boldsymbol{s}_{1a}$ and $\boldsymbol{s}_{2b}$ is right of $\boldsymbol{s}_{1b}$, as well as the extension checking for inverted reintegration, are similar and will be omitted for brevity.

Let $\boldsymbol{s}$ denote the input sequence, l_1 the minimum maximal repeat length for first level search, l_2 the minimum maximal repeat length for second level search, m the extension checking length, p the maximum allowed total mismatch. The algorithms for direct reintegration and inverted reintegration are summarized in the following tables.

Algorithm 1 Direct Reintegration Search

Input (string s, int l_1, int l_2, int m, double p)
Output R

t $\leftarrow$ SuffixTree(s)
S $\leftarrow$ MaximalRepeatedPairs(s,t,l_1)
R $\leftarrow \emptyset$
foreach $(s_{i,1}, s_{i,2}) \in S$ **do**
  $s_{right1} \leftarrow$ string with length m immediate right of $s_{i,1}$
  $s_{left2} \leftarrow$ string with length m immediate left of $s_{i,2}$
  $t' \leftarrow SuffixTree(s_{right1} \# s_{left2})$
  $S' \leftarrow MaximalRepeatedPairs(s_{right1} \# s_{left2}, t', l_2)$
  foreach $(u, v) \in S'$ **do**
    initialize $A_1, A_2, A_3, B_1, B_2, B_3$
    $i_1 \leftarrow argmin_i(d\_vec(A_2, B_3)[i] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i])$
    $dist_1 \leftarrow d\_vec(A_2, B_3)[i_1] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i_1]$
    $i_2 \leftarrow argmin_i(d\_vec(B_2, A_3)[i] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i])$
    $dist_2 \leftarrow d\_vec(B_2, A_3)[i_2] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i_2]$
    if $dist_1 + dist_2 < p$ **then**
      Obtain $(s_{1a}, s_{1b}, s_{2a}, s_{2b})$ from i_1, i_2
      $R \leftarrow R \cup \{(s_{1a}, s_{1b}, s_{2a}, s_{2b})\}$
    end
  end
  $s_{left1} \leftarrow$ string with length m immediate left of $s_{i,1}$
  $s_{right2} \leftarrow$ string with length m immediate right of $s_{i,2}$
  $t'' \leftarrow SuffixTree(s_{left1} \# s_{right2})$
  $S'' \leftarrow MaximalRepeatedPairs(s_{left1} \# s_{right2}, t'', l_2)$
  foreach $(u, v) \in S''$ **do**
    initialize $A_1, A_2, A_3, B_1, B_2, B_3$
    $i_1 \leftarrow argmin_i(d\_vec(A_2, B_3)[i] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i])$
    $dist_1 \leftarrow d\_vec(A_2, B_3)[i_1] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i_1]$
    $i_2 \leftarrow argmin_i(d\_vec(B_2, A_3)[i] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i])$
    $dist_2 \leftarrow d\_vec(B_2, A_3)[i_2] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i_2]$
    if $dist_1 + dist_2 < p$ **then**
      Obtain $(s_{1a}, s_{1b}, s_{2a}, s_{2b})$ from i_1, i_2
      $R \leftarrow R \cup \{(s_{1a}, s_{1b}, s_{2a}, s_{2b})\}$
    end
  end
end

Algorithm 2 Inverted Reintegration Search

Input (string s, int l_1, int l_2, int m, double p)
Output R

t ← SuffixTree(s#s')
S ← MaximalRCPairs(s#s',t,l_1)
R ← $\emptyset$
foreach $(s_{i,1}, s_{i,2}) \in S$ **do**
  $s_{right1} \leftarrow$ string with length m immediate right of $s_{i,1}$
  $s_{right2} \leftarrow$ string with length m immediate right of $s_{i,2}$

  $t' \leftarrow SuffixTree(s_{right1a}\#s_{left1b})$
  $S' \leftarrow MaximalRCPairs(s_{left}\#s_{right}, t', l_2)$
  foreach $(u, v) \in S'$ **do**
    initialize $A_1, A_2, A_3, B_1, B_2, B_3$
    $i_1 \leftarrow argmin_i(d\_vec(A_2, B_1')[i] + rev\_d\_vec(A_2', B_3)[i])$
    $dist_1 \leftarrow d\_vec(A_2, B_1')[i_1] + rev\_d\_vec(A_2', B_3)[i_1]$
    $i_2 \leftarrow argmin_i(d\_vec(B_2, A_1')[i] + rev\_d\_vec(B_2', A_3)[i])$
    $dist_2 \leftarrow d\_vec(B_2, A_1')[i_2] + rev\_d\_vec(B_2', A_3)[i_2]$
    if $dist_1 + dist_2 < p$ **then**
      Obtain $(s_{1a}, s_{1b}, s_{2a}, s_{2b})$ from i_1, i_2
      $R \leftarrow R \cup \{(s_{1a}, s_{1b}, s_{2a}, s_{2b})\}$
    end
  end
  $s_{left1} \leftarrow$ string with length m immediate left of $s_{i,1}$
  $s_{left2} \leftarrow$ string with length m immediate left of $s_{i,2}$
  $t'' \leftarrow SuffixTree(s_{left1}\#s_{left2})$
  $S'' \leftarrow MaximalRCPairs(s_{left1}\#s_{left2}, t'', l_2)$
  foreach $(u, v) \in S''$ **do**
    initialize $A_1, A_2, A_3, B_1, B_2, B_3$
    $i_1 \leftarrow argmin_i(d\_vec(A_2, B_3)[i] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i])$
    $dist_1 \leftarrow d\_vec(A_2, B_3)[i_1] + rev\_d\_vec(A_2^{-1}, B_1^{-1})[i_1]$
    $i_2 \leftarrow argmin_i(d\_vec(B_2, A_3)[i] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i])$
    $dist_2 \leftarrow d\_vec(B_2, A_3)[i_2] + rev\_d\_vec(B_2^{-1}, A_1^{-1})[i_2]$
    if $dist_1 + dist_2 < p$ **then**
      Obtain $(s_{1a}, s_{1b}, s_{2a}, s_{2b})$ from i_1, i_2
      $R \leftarrow R \cup \{(s_{1a}, s_{1b}, s_{2a}, s_{2b})\}$
    end
  end
end

4. Partition and Parallelization

The bottleneck of our algorithm is the space requirement of suffix tree used for searching maximal repeated pairs on the entire input sequence. Even though the space complexity of a suffix tree is linear with respect to the sequence length, in practice the memory requirement often becomes prohibitive due to large input size. To overcome this problem, we devise a divide and conquer approach. A helper program is first used to divide the long input sequence s into n partitions: $s = s_1 s_2 ... s_n$, where n is large enough so that the length of each s_i is manageable; we then perform microDNA reintegration searching for each pair (s_i, s_j), where $i \leq j$.

If $i = j$, we still perform first level search with the same method discussed in 3.1. If $i \neq j$, we perform first level maximal repeated (resp. inverted) pairs search on string $\boldsymbol{s}_i \# \boldsymbol{s}_j$ (resp. $\boldsymbol{s}_i \# \boldsymbol{s}'_j$), and only keep pairs $(\boldsymbol{s}_a, \boldsymbol{s}_b)$ (resp. $(\boldsymbol{s}_a, \boldsymbol{s}'_b)$) with s_a in $\boldsymbol{s}_i$ and $\boldsymbol{s}_b$ in $\boldsymbol{s}_j$ (resp. $\boldsymbol{s}'_b$ in $\boldsymbol{s}'_j$). The extension checking is performed similarly as discussed in 3.2. That the final result is then adjusted so that the locations of reintegration in the original sequence $\boldsymbol{s}$, rather than the locations in the partition sequences $\boldsymbol{s}_i$ or $\boldsymbol{s}_j$, are returned.

The effect of running the algorithm on all these pairs is equivalent to running the algorithm on a single large sequence, except when microDNA reintegration occurs at the junction of two partitions. Note that the computation cost of parallelization method is higher than without parallelization, because the same maximal repeated pairs in $\boldsymbol{s}_i$ could be visited multiple times during the first level search. For example, if we perform tasks on $(\boldsymbol{s}_1, \boldsymbol{s}_1)$,$(\boldsymbol{s}_1, \boldsymbol{s}_2)$ and $(\boldsymbol{s}_1, \boldsymbol{s}_3)$, then maximal repeated pairs with length larger than l_1 in $\boldsymbol{s}_1$ will be repeatedly searched for all three of the tasks. However, this cost is limited, because maximal repeated pairs not relevant to the task will not be chosen for second level search and extension checking, which contributed most of the computation cost.

5. Experiment

A. Preprocessing using TRF

Recall a tandem repeat is a pattern of one or more nucleotides repeated, and the repetitions are directly adjacent to each other. For example, in sequence $GGACTACTACTACTCCT$, the segment $ACTACTACTACT$ is tandem repeats since it's ACT repeated four times adjacently. It is easy to see that a tandem repeat can trivially satisfy the definition of direct reintegration.

For example, consider the sequence:

$$A\underline{GCCGCCGCC}GCCGCC\underline{GCCGCCGCC}T$$

Note the underlined part "$GCCGCCGCC$" forms a maximal repeated pair (s_{1a}, s_{1b}). The right extension of s_{1a} and the left extension of s_{1b} are both GCC, making them also a circle repeat (assuming extension check length and minimum maximal repeat length are both 3). Even though these segments are technically circle repeats, they are more a result of tandem repeats rather than direct reintegration.

In the experiment, we use Tandem Repeat Finder (TRF) to mask and remove the tandem repeat regions in the sequence as part of preprocessing. This preprocessing step offers another advantage: it significantly reduces the number of maximal repeated pairs in the sequence, thus greatly improving the algorithm speed performance. We should note that the elimination of tandem repeats does not imply that microDNA reintegration do not occur in the tandem repeats area; rather, we remove the tandem repeat regions because there is no mathematically way to distinguish them and microDNA reintegration.

B. Results on Human Chromosomes

We ran our algorithm on three sequences: chromosome-Y, chromosome-22, and chromosome-21. TRF is used during preprocessing to remove tandem repeat regions in the sequences. We set $l_1 = 40$, $l_2 = 20$, $m = 800$. The total mismatch p for each reintegration is determined by $\alpha \cdot (|\boldsymbol{s}_{1a}\boldsymbol{s}_{2a}|)$, where α, the mismatch ratio, is set to 0.1. Additionally, after obtaining the results, we impose another mismatch filter so that patterns which don't satisfy $d(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b}) < \alpha \cdot |\boldsymbol{s}_{1a}|$ or $d(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b}) < \alpha \cdot |\boldsymbol{s}_{2a}|$ are discarded. This is a stricter constraint than the one in the problem formulation, and could helps us to remove patterns where mismatch only concentrates on $(\boldsymbol{s}_{1a}, \boldsymbol{s}_{1b})$ or $(\boldsymbol{s}_{2a}, \boldsymbol{s}_{2b})$.

Note that microDNA reintegration pairs found by our algorithm could have duplicates, and may overlap with other pairs, due to the fact that maximal repeated pairs can overlap. We say two microDNA reitegration pairs (either direct or inverted) $...\mathbf{X}_1...\mathbf{Y}_1...$ and $...\mathbf{X}_2...\mathbf{Y}_2...$ overlap if $\mathbf{X}_1, \mathbf{X}_2$ overlaps and $\mathbf{Y}_1, \mathbf{Y}_2$ overlaps. The following table shows sequence length after tandem repeats removal, the number of patterns found after duplication removal, the number of patterns found after overlapping removal.

Sequence Name	Sequence Length	#Direct microDNA Reintegration Pairs	#non-overlapping Direct Pairs	direct pairs with microhomology	#Reverse microDNA Reintegration Pairs	#non-overlapping inverted Pairs	inverted pairs with microhomology
chr-Y	23974895	6137	2126	62.5%	6626	1816	17.1%
chr-22	35018961	3419	1749	53.1%	3091	1525	30.9%
chr-21	36454203	4410	846	58.2%	427	341	17.9%

One of the characteristics of microDNAs found in biology research is length 2-15 bps direct repeats of microhomology [5]. In the experiment, we define a microDNA reintegration (either direct or inverted) $...\mathbf{X}...\mathbf{Y}...$ satisfies microhomology if for some integer $k \in [2, 15]$, either (1) a prefix of $\mathbf{X}$ with length k is equal to the $k-$substring right after $\mathbf{X}$, or (2) a prefix of $\mathbf{Y}$ with length k is equal to the $k-$substring right after $\mathbf{Y}$. It is interesting that there are much more direct reintegration pairs satisfies microhomology than inverted pairs.

Finally, we performed a simple categorization of microDNAs. We define that for a reintegration pair $...\mathbf{X}...\mathbf{Y}...$, $\mathbf{X}$ and $\mathbf{Y}$ belongs to the same class. For example, in $...\mathbf{X}...\mathbf{Y}...\mathbf{Z}...$ where $(\mathbf{X}, \mathbf{Y})$, $(\mathbf{X}, \mathbf{Z})$, $(\mathbf{Y}, \mathbf{Z})$ are found as microDNA reintegration pairs, $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ are classified as the same category. Note that by definition, each class will have at least two elements. Biologically, a microDNA class that has more than two elements could imply the microDNA produces a copy (or copies) before reintegration, or the region where the microDNA is located was copied. In the experiment, a significant number of classes with two elements and classes with more than two elements are present. We then compared the break point location, measured by the length of $\boldsymbol{s}_1$ or $\boldsymbol{s}_2$, of microDNAs of the same class. By definition, the first two elements in the same class have the same break point. Interestingly, we found that in classes that have more than two elements, many (> 2), though not all of, microDNAs within the same class have the same break point location. We hypothesize that microDNAs within the same class (besides the first two elements) that has the same break point location could be generated generated by either (1) before reintegration, microDNA got copied after it is broken from the circular structure to a line, or (2) the region where the microDNA is located was copied. On the other hand, microDNAs within the same class (besides the first two elements) that has different break points locations could be generated by the microDNA getting copied before it is broken from circular structure to line structure.

C. Simulation

One interesting question to ask is: are the patterns found by our algorithm meaningful? That is, is it possible that, we are still able to find patterns like $...\boldsymbol{s}_{1a}\boldsymbol{s}_{2a}...\boldsymbol{s}_{2b}\boldsymbol{s}_{1b}...$ in a arbitrary gene sequence without microDNA reintegration? To test this theory, we simulated complete random gene sequences of length 30000000, with each letter having the same probability (i.e. $\frac{1}{4}$) to appear. In such sequences, no maximal repeated pair of length greater or equal to 40 could be found in the first step of two-layered search, let alone any reintegration patterns.

The simulation above ignores the existence of repetitive structure in gene sequences. For example, two maximal repeated pairs $(\boldsymbol{s}_{i,1}, \boldsymbol{s}_{i,2})$ and $(\boldsymbol{u}, \boldsymbol{v})$ in the two level search would likely result in a direct pattern if $\boldsymbol{s}_{i,1}$ (resp. $\boldsymbol{s}_{i,2}$) and $\boldsymbol{u}$ (resp. $\boldsymbol{v}$) are so close to each other that a direct pattern is formed regardless of extension checking. Admittedly, this alone does not prove that such patterns happen by chance and not miroDNA reintegration. However, if most of the patterns we found satisfies reintegration definition due to the closeness of maximal repeated pairs regardless of extension checking, it would be reasonable to suspect that the detected patterns is more likely a result of abundance and closeness of inherent repetitive structures rather than microDNA reintegration mechanism. We used two different simulation with chromosome-Y to check the influence of repetitive structures in the sequence on microDNAs we found.

In the first simulation, for each direct microDNA reintegration, the two maximal repeated pairs $(\boldsymbol{s}_{i,1}, \boldsymbol{s}_{i,2})$ and $(\boldsymbol{u}, \boldsymbol{v})$ found in two-layer search were recorded (note: only the pairs that correspond to valid microDNA reintegration are recorded). After removing duplicates and overlapping of the recorded pairs, we replace the regions of the sequence that are not covered by these maximal repeated pairs with randomly generated sequence of the same lengths. If the detected patterns found in the original chromosome-Y were the result of closeness of maximal repeated pairs, then we would expect to find a similar amount of patterns in the simulated sequence. In our experiment, only less than 200 pattern were found in the simulated sequence, less than 10% of the result for the original chromosome-Y.

The second simulation sequence is obtained by permuting the repeat regions (regions covered by maximal repeated pairs) and non-repeat regions in the entire sequence. More specifically, we record all maximal repeated pairs found in the first-level search; after removing duplicates and overlapping of the recorded pairs, we record all the substrings not covered by the repeats. We then randomly permute the order of all repeat regions and non-repeat regions to obtain the simulated sequence. Similar to the result of previous simulation, in average, less than 300 non-overlapping patterns were found in the simulated sequence.

Both simulations yield much less microDNA pairs than the original DNA sequence, indicating that microDNA reintegration pairs we obtained from human chromosomes is unlikely solely a product of inherent repetitive structure in gene sequences.

6. Conclusion and Future Work

In this paper, we established mathematical definitions of microDNA reintegration, and introduced an algorithm that efficiently searches such reintegration patterns in gene sequences. The algorithm first searches for candidates using two-level search, and checking the extension of each candidate to ensure mismatch constraint is satisfied. The algorithm successfully found hundreds or thousands of reintegration patterns on chromosome-Y, chromosome-22 and chromosome-21. The fact that relatively few microDNA reintegration pairs are found in the simulation indicates patterns found in human chromosomes are not solely caused by the abundance of inherent repetitive structures in gene sequences. In sum, patterns found by the algorithm are likely the evidence for microDNA reintegration into the genome.

One possible future direct is improving the efficiency of the algorithm. The suffix trees used for searching maximal repeated pairs can be replaced by more space-efficient suffix arrays, introduced by Manber and Myers [6], as it has been shown that any bottom-up traversal of a suffix tree can be simulated on a suffix array [7]. Another aspect of the algorithm that could be improved is maximal inverted pair searching. Recall that currently, to search maximal inverted pairs in $\boldsymbol{s}$, we need to construct $\boldsymbol{s}\#\boldsymbol{s}'$, search for maximal repeated pairs, adjust indices and remove duplicates. The drawback of this approach include (1) requires double memory size to construct $SuffixTree(\boldsymbol{s}\#\boldsymbol{s}')$ compared to $SuffixTree(\boldsymbol{s})$, (2) requires duplication removal. Although our divide-and-conquer approach alleviates both of problems, it is not a perfect solution. Thus, developing an algorithm that could searches all maximal inverted pairs in $\boldsymbol{s}$ with $O(|s|+k)$ (without requiring double memory size), where k is the number of maximal inverted pairs, would be an interesting research direction.

REFERENCES

[1] P. K. et al., "Normal and cancerous tissues release extrachromosomal circular dna (eccdna) into the circulation," in *Molecular Cancer Research*, vol. 15, 2017, 1197â1205.

[2] L. W. D. et al., "Production of extrachromosomal microdnas is linked to mismatch repair pathways and transcriptional activity," in *Cell Reports*, vol. 11, 2015, 1749â1759.

[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, 1966, pp. 707–710.

[4] D. Gusfield, "Finding all maximal pairs in linear time," in *Algorithms on strings, trees, and sequences: Computer science and computational biology Cambridge University Press*, 2009, pp. 147–148.

[5] Y. S. et al., "Extrachromosomal microdnas and chromosomal microdeletions in normal tissues," in *Science*, vol. 336, 2012, 82â86.

[6] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *SIAM Journal on Computing*, vol. 22, 1993, 935â948.

[7] S. K. M. I. Abouelhoda and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," in *Journal of Discrete Algorithms*, vol. 2, 2004, 52â86.