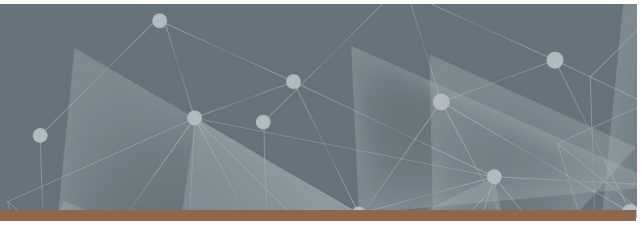
CHALMERS
UNIVERSITY OF TECHNOLOGY



Clustering for DNA Storage

An Efficient Trie-based Method

Degree project report in Communication Engineering

YOUJUN LUO

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Degree project report 2023

Clustering for DNA Storage

An Efficient Trie-based Method

YOUJUN LUO

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2023

Clustering for DNA Storage
An Efficient Trie-based Method
YOUJUN LUO

Supervisor: Alexandre Graell i Amat, Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden
Co-supervisor: Eirik Rosnes, Simula UiB, Bergen, Norway
Examiner: Alexandre Graell i Amat, Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden

Degree project report 2023
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Clustering for DNA Storage
An Efficient Trie-based Method
YOUJUN LUO
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Deoxyribonucleic acid (DNA) has emerged as a potential storage medium due to its high information density and durability. Playing a critical role in the DNA storage process, the clustering partitions similar sequenced reads into groups for the decoder. However, the synthesis, storing, and sequencing of DNA introduce insertion, deletion and substitution (IDS) errors, making the clustering of reads harder. And because of the large numbers of reads in DNA storage, traditional clustering methods in biological domains become time-consuming. Recently, a trie-based algorithm called Clover is proposed to accelerate the clustering process in DNA storage by fuzzy searching the input reads on a trie structure. However, it only considers substitutions during the search, while deletions and insertions are addressed through multiple tests on different regions of input reads afterwards. In this thesis, we proposed efficient clustering algorithms that optimize the trie searching by considering the IDS channel. In our algorithm, discrete IDS errors are corrected with a depth-limited strategy. And a cluster merging method is developed to improve the success rate of searching. We validate the proposed methods on three real-world DNA storage datasets, achieving the lowest runtime and comparable accuracy compared to state-of-the-art DNA clustering tools.

Keywords: DNA storage, Indexing, Clustering, Trie, Levenshtein distance, Poucet search, Depth-limited search, Cluster merging.

Acknowledgements

I would like to express my gratitude to Professor Alexandre Graell i Amat for providing me with the opportunity on this thesis and an interesting topic, as well as his course on channel coding that provided me with the necessary background to delve into the theories and concepts explored in this thesis.
And I would also like to extend my appreciation to Eirik Rosnes and Christian Häger for their helpful suggestions on the design of methods and experiments during the thesis.
Finally, I would like to thank my parents for their unwavering support and encouragement throughout my studies at Chalmers University of Technology. Their belief in me and their constant motivation has been a driving force behind my accomplishments.

Youjun Luo, Gothenburg, May 2023

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

DNA storage technology employs DNA molecules as a medium for data storage, leveraging its high density and longevity to potentially address future storage density and reliability demands [1]. A general working pipeline [2] of DNA storage is shown in Figure 1.1. With files in binary bits, the data can be split into bit sequences of the same length. Then these binary sequences are encoded into codewords by adding redundant bits, for error detection and correction. The codewords in binary code are then converted to quaternary codes for mapping to four DNA nucleotides named Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). And these codes are synthesized into DNA strands and stored in a pool. Retrieval of information involves sequencing the stored DNA strands. Since each strand can be amplified and sampled several times, reads are unordered and there are multiple repeats for each synthesized strand, which makes decoding difficult. So, clustering algorithms are introduced to group reads from the same original sequence, and decoding is performed on the resulting clusters to improve the success rate of decoding. Finally, binary data can be retrieved by the decoder from the obtained clusters. Many methods for DNA

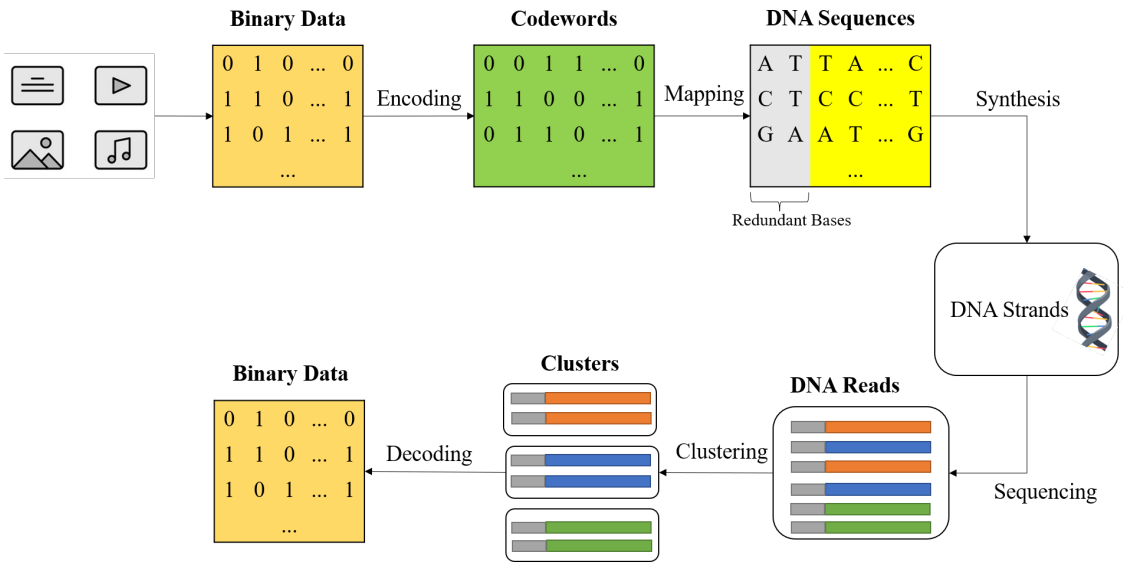


Figure 1.1: Principle of DNA storage.

clustering have existed for a long time, such as CD-HIT [3], UCLUST [4], VSEARCH

[5] and MeShClust [6]. They are frequently used for biological domains, but several challenges remain in their practical use for DNA storage.

The large number of clusters and reads: Shorter sequences of 60-300 nucleotides are preferred in DNA storage due to lower error rates and ease of synthesis and amplification. However, the use of shorter sequences leads to a larger number of clusters and reads, ranging from millions to billions, posing a challenge for traditional clustering algorithms. This can negatively impact the efficiency of the DNA storage reading process.
IDS errors: DNA sequences may undergo mutations during synthesis, storage, and sequencing, resulting in errors in the retrieved reads. All possible mutations can be represented with several numbers of IDSs. These operations change the bases of original sequences and length, so sequenced reads contain errors. To address this issue, additional error correction or fuzzy matching has to be taken on reads, which can increase the complexity of the algorithm.
Similarity metrics for clustering: Finding an appropriate similarity metric for efficient and accurate DNA clustering is a challenging task. Such a metric can be categorized as alignment-based or alignment-free. Alignment-based metrics compare the identity between each pair of sequences, which takes a long time for large-scale data. Alignment-free metrics convert DNA sequences into feature vectors in a new space, such as k-mer, which does not rely on pairwise calculation. However, these metrics may not well separate reads when there are numerous closed clustering centers.
Runtime for clustering: Efficient clustering is critical for DNA storage as data retrieval often relies on multi-read error correction. Faster clustering contributes to quicker data retrieval, facilitating the usage of large-scale DNA sequencing datasets.

Despite the complexities and time consumption associated with DNA clustering, existing information within DNA sequences can be leveraged to improve clustering algorithms in DNA storage. A key component is the address area or indexes, which are added to binary or DNA sequences to indicate the order of segments during file restoration. Although these areas are typically short, they are designed to be independent of each other to facilitate the reordering of reads in the decoder, which also provides well-separated intervals for DNA clustering. So, investigating the use of indexes in DNA clustering is an interesting and open topic for improving clustering performance in DNA storage.

1.2 Thesis Aim and Scope

This thesis aimed to develop efficient DNA clustering methods for the retrieval of DNA storage data. Leveraging the potential benefits of using indexes, we sought to reduce computational complexity and improve clustering accuracy while ensuring well-separated clusters. Our focus was on optimizing both the accuracy and runtime of DNA clustering with index information to efficiently cluster millions of real-world DNA reads with high accuracy and fast speed.

1.3 Thesis Structure

This thesis begins with an introduction to DNA storage and the challenges associated with DNA clustering. In Chapter 2, we provide an overview of the mainstream DNA storage model, clustering methods, and related theory utilized in this thesis. In Chapter 4, a clustering method using the depth-limited search on trie is presented. And it is further improved by a combined search strategy. In Chapter 3 the metrics for comparison are introduced, and then in Chapter 5, the comparison of the proposed algorithms with three state-of-the-art clustering methods for DNA storage, using three real-world sequencing datasets is presented. In Chapter 6, we further discuss the possibility to optimize the indexing scheme for improving clustering results, and the parameter selection for the proposed methods. Finally, Chapter 7 concludes the work in this thesis.

2
Literature Review

2.1 DNA Storage Model

2.1.1 DNA Storage Scenarios using Indexing

In recent years, several schemes for DNA storage have been proposed and validated by researchers. In this study we referred to three different scenarios that utilize indexes for DNA storage, their specifications are shown in Table 2.1.

Name	Erlich et al.[7]	Song et al.[8]	Organick et al.[9]
Strands synthesized	72,000	187,973	210,000
Strands length	200 nt	200 nt	150 nt
Outer code	Fountain code	Fountain code	Reed-Solomon
Inner code	Reed-Solomon	CRC	Rotating Code
Index method	Random index	Random index	Block index
Clustering method	-	-	Starcode
Synthesis method	Twist Bioscience	Twist Bioscience	Twist Bioscience
Sequencing method	Illumina	Illumina	Illumina

Table 2.1: DNA storage scenarios using indexing are considered in this thesis.

DNA fountain storage Erlich et al. report a DNA storage strategy that employs fountain codes for reliable data retrieval. Fountain codes work by sending droplets of data information like a fountain. The encoder generates random linear combinations of data splits using random seeds, and stores both the encoded data and seeds in the nucleotides. Thus, the receiver can collect a sufficient number of payloads and seeds and uses decoding algorithms to reconstruct the original sequences. As illustrated in Figure 2.1, for droplet generation, firstly the data is split into 32-byte segments. Then, these segments are drawn randomly with a random seed generated from a robust soliton distribution and then combined with exclusive-or. Then the generated payloads and seeds are concatenated as outer codes, and then they are encoded with Reed-Solomon code as inner code. The inner codes are then sorted according to GC-content and homopolymer, and the adapter is added to both ends for sequencing. Thus, each code is constructed with a random seed in 16 nucleotides as a header that indicates its position in segmentation, followed by 128 nucleotides of data payload and 8 nucleotides of Reed-Solomon code for erroneous reads detection. Since the random seeds are generated by a pseudo-random number generator and

a screening monitor, while the seed is received, the splits used for coding can be known and can be retrieved by exclusive-or.
The clustering is not included in the decoding process. Instead, erroneous sequences that are detected with error correction codes are excluded, and each new read is compared with all previously processed reads. Repeated inputs are discarded to ensure the accuracy of the decoded data.

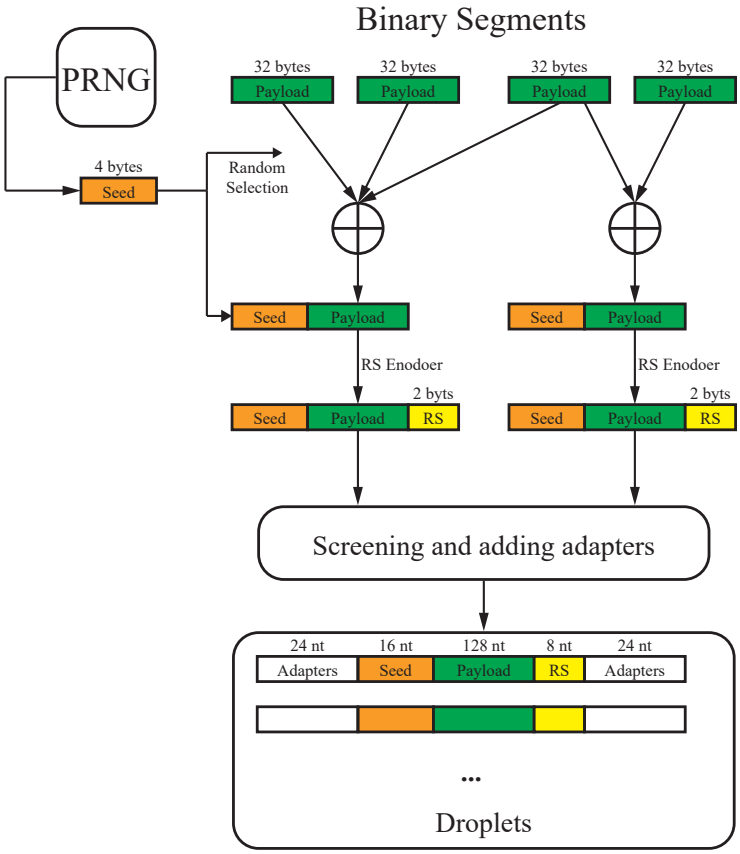


Figure 2.1: Fountain encoder for DNA storage.

De Bruijn graph-based storage Song et al. propose a DNA reconstruction strategy based on the De Bruijn graph (DBG), for handling the handling breaks and rearrangements errors in DNA storage. In this scheme, indexes are designed as indicators and reads with the same indexes are gathered to construct a graph for trace reconstruction. Similar to the previous scenario, this DNA storage scheme utilizes fountain codes as the outer code. However, in this scenario, the Cyclic Redundancy Check (CRC) code instead of the RS code is used as the inner code. The DNA strand designed in this scenario contains 16 nucleotides index from pseudo-random number generator, 140 nucleotides data payload and 8 nucleotides CRC code, flanked by landing sites for sequencing primers of length 18 nucleotides.
Random access storage For large-scale DNA storage, Microsoft and Organick's group report a random-access scheme. Based on the Grass' encoding and decoding method reported in 2015[10], they design unique primers of length 20 nucleotides for addressing, which satisfy several design criteria, including the GC-content, sequence-complementarities, homopolymers and minimum hamming distance. The encoder

first splits the digital file into blocks, each block composition is represented by a matrix in the shape of 10 rows and 55,000 columns, where each cell carries a 16-bit value. Then each row is encoded with Reed-Solomon code to have 15% of redundant bits. After that, every column is converted into a DNA sequence of length 110 nucleotides by adding index information (block index and column index) and converted to quaternary code by applying rotating coding [11]. Finally, the designed addressing PCR primer is added to both ends to construct the final strands, as shown in Figure 2.2.

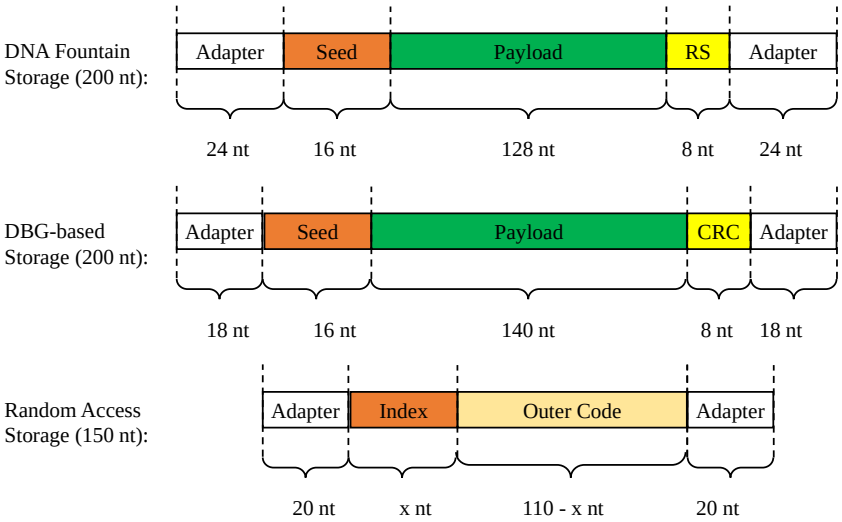


Figure 2.2: The structure of the designed strands in the three DNA storage models.

2.1.2 Channel Characterization and Model

Mutations of DNA may occur during the process of synthesizing, storing, and sequencing DNA for data storage[12]. Systematic errors may occur when molecules cannot be successfully synthesized or sequenced, or when there is incomplete sampling. Intrinsic errors, on the other hand, typically result from the loss of molecules due to DNA decay, and errors that occur during synthesis and sequencing. Intrinsic errors can be represented by several IDSs, and studies have shown that substitution error rates range from 0.0015 to 0.0004 errors per base, while insertion and deletion errors are significantly less likely, on the order of 10^{-6}.
The Markov chain-based channel[13] proposed by Davey and Mackay is considered in this thesis for analysis and creating virtual sequencing data in DNA storage. During the synthesis and sequencing, the quaternary codes with the alphabet $\Sigma = \{A, T, G, C\}$ are transmitted over parallel channels that introduce IDS errors. And the IDS channels are defined with three transition probabilities p_i, p_d and p_s. The sequence of length L to be transmitted can be viewed as a queue of L symbols. While we have the sequence to transmit $\mathbf{x} = (x_1, x_2, ...x_L)$, the received sequences

$\mathbf{y} = (y_1, y_2, ...y_{L'})$ are generated state by state with the following process. Assuming that x_i is queued as shown in Fig. 2.3. All possible transitions can be:

- With probability p_i, a symbol a is draw from the alphabet Σ and added to the received sequence before x_i is transmitted.
- With probability p_d, the symbol x_i is deleted and not transmitted.
- The symbol is transmitted with probability $p_t = 1 - p_i - p_d$. With probability p_s, the transmission of x_i is replaced by the transmission of x_i^*, which is from Σ but not the same as x_i. Otherwise, the symbol x_i is transmitted without error and added to the received sequence $\mathbf{y}$.

This process finishes after all the symbols are queued and processed. By observing the sequence of emitted symbols from the IDS channel, the explicit emission probability of this Markov process can be concluded with Table 2.2.

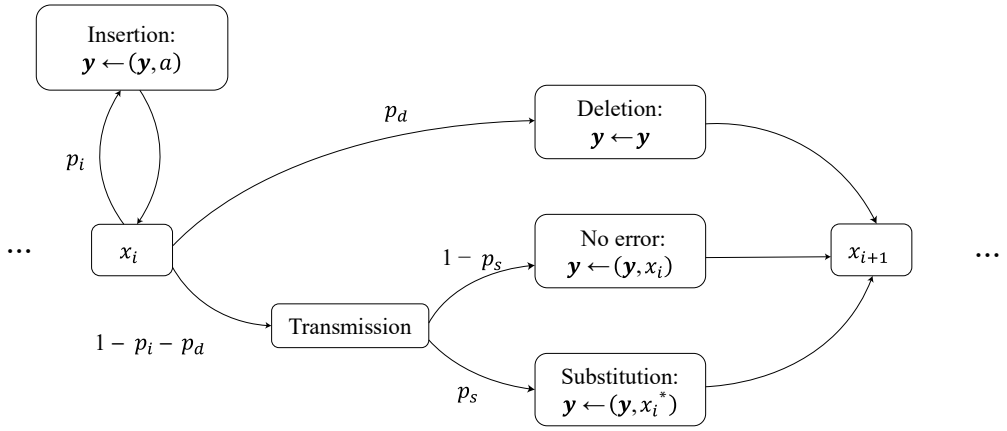


Figure 2.3: IDS channel model with probabilities p_i, p_d and p_s of insertions, deletions, and substitutions.

Table 2.2: Explicit emission probability for IDS channel.

Insertions	Deletions	Substitutions	Probability
0	0	0	$p_t(1-p_s)$
0	1	-	p_d
0	0	1	$p_t p_s$
1	0	0	$p_i p_t(1-p_s)$
1	1	-	$p_i p_d$
1	0	1	$p_i p_t p_s$
⋮	⋮	⋮	⋮
I	0	0	$p_i^I p_t(1-p_s)$
I	1	-	$p_i^I p_d$
I	0	1	$p_i^I p_t p_s$
⋮	⋮	⋮	⋮

2.2 Problem Statement

The generation of sequencing data can be described based on the previous channel assumption. Firstly, with a given number of synthesis sequences k and the sequence length L, the set to synthesised $D = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k\}$ is created. For a random dataset, each of the k sequences is obtained by randomly sampling L times from the alphabet $\Sigma = \{A, T, G, C\}$. Secondly, each sequence $\mathbf{x}_i$ is amplified by d_i times, in which d_i is a random number uniformly distributed within a preset range. Then, the amplified sequences are transmitted over the IDS channel with transition probabilities p_i, p_d and p_s. A symbol $\mathbf{S} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$ is used to represent the set of received N reads.

Thus, the clustering is the algorithm for finding the optimal mapping from received reads to a set of clusters $f : \mathbf{S} \mapsto \widetilde{\mathbf{C}}$, $\widetilde{\mathbf{C}} = \{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_{k'}\}$, where entries $\mathbf{c}_i$ are clusters containing received sequences and k' is the number of clusters that algorithm outputs.

2.3 Clustering Methods for DNA Storage

2.3.1 Incremental Clustering in Biological Domain

DNA clustering has long been used in biological domains, such as grouping protein families and finding motifs in DNA proteins[6], [14]. Traditional methods work by studying similarity measures between protein sequences. But for the clustering, calculating these metrics needs to compare every two sequences in the datasets, so typically the complexity of the clustering is $O(N^2)$, where N is the number of sequences.

With the increasing size of the DNA dataset, the need for efficient clustering has arisen. CD-HIT[3] provides an incremental method for DNA clustering, which is a well-known solution for clustering large-scale datasets. UCLUST and VSEARCH are another two popular DNA clustering tools[4], [5], which use a similar clustering scheme but the sequences are compared with a different distance metric.

The incremental clustering treats the sequences to be clustered as a stream, and it performs a greedy strategy, which can be concluded as Algorithm 1. The clustering is initialized with an empty set for recording representatives. Each input read will be grouped with the existing representatives in the set that has high similarity, otherwise, the input reads will be added to the set as a new representative. Since every read is only compared with representatives, the time complexity of incremental clustering is $O(kN)$, where k is the number of representatives and N is the number of reads.

Incremental clustering requires accurate similarity comparison for sequences. If the similarity metrics are too loose, the input reads may be clustered with the wrong representatives. On the other hand, if the similarity metrics are too strict, the input reads may not find a matching representative and create new clusters, resulting in more clusters than expected.

2.3.2 Trie-based Incremental Clustering

As an enhancement to traditional incremental clustering methods, Guanjin et al. report the Clover clustering method that compares DNA reads directly in a trie (also referred to as prefix tree), instead of using similarity metrics [15]. An example of a trie with 4 layers is shown in Figure 2.4. The main idea behind this data structure is that DNA reads with the same prefix share the branch in a trie structure and the branch splits at the position of different symbols. Thus, a representative can be represented with a leaf in the trie. For DNA reads, only the symbols A, T, C and G are considered, and each node represents a symbol and links up to 1 parent node and 4 children nodes, resulting in a quadtree to store the reads. Clustering labels are stored on the leaves of the trie to indicate the label of the cluster if the search reaches the end of the trie.

The incremental clustering with trie is a process of constructing trie from reads, as shown in Algorithm 2. An empty trie is first initialized, and then for every sequence to construct the trie, the algorithm searches a path from the root to the leaves that have exactly the same symbols in the sequence. And if this path cannot be found, a new branch will be created. Note that the number of representatives that the trie can store is 4^n where n is the depth of the trie.

To search for an exact input sequence with a trie, the algorithm begins by comparing the first symbol of the input with the children of the root and continues to compare all symbols consecutively. Once the search reaches the end of the trie, the label on the leaf is used to determine the cluster in which the input sequence should be grouped.

For the presence of channel errors, in the Clover algorithm, a certain permissible number of substitution errors can be ignored during the traversal. Also, the algorithm shifts the interval for comparison, to maximize the number of matched nodes when there are insertions and deletions. Due to the fact that the search time in a trie is directly related to the depth of the trie, the algorithm has a linear time complexity of $O(N)$, where N is the size of the input dataset.

But the Clover has a higher probability of failure when clustering due to its strict traversal strategy. Insertion and deletion errors can often result in a burst of mis-

Algorithm 1 Incremental clustering

Input: Reads $\mathbf{S} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$.
Output: Representatives set $\mathbf{C}$ and mapping $\pi : \mathbf{S} \mapsto \mathbf{C}$.

Initialize an empty set $\mathbf{C} \leftarrow \emptyset$ and empty mapping $\pi \leftarrow \emptyset$.
for $i = 1, 2, \ldots, N$ **do**
    Matching input sequence $\mathbf{y}_i$ with current set $\mathbf{C}$.
    for $j = 1, 2, \ldots, |\mathbf{C}|$ **do**,
        if $c_j \in \mathbf{C}$ matches $\mathbf{y}_i$ within a metric threshold **then**
            Add mapping $\mathbf{y}_i \mapsto c_j$ to π.
        else
            Add $\mathbf{y}_i$ to $\mathbf{C}$ as a new representative.
            Add mapping $\mathbf{y}_i \mapsto c_{|\mathbf{C}|}$ to π.

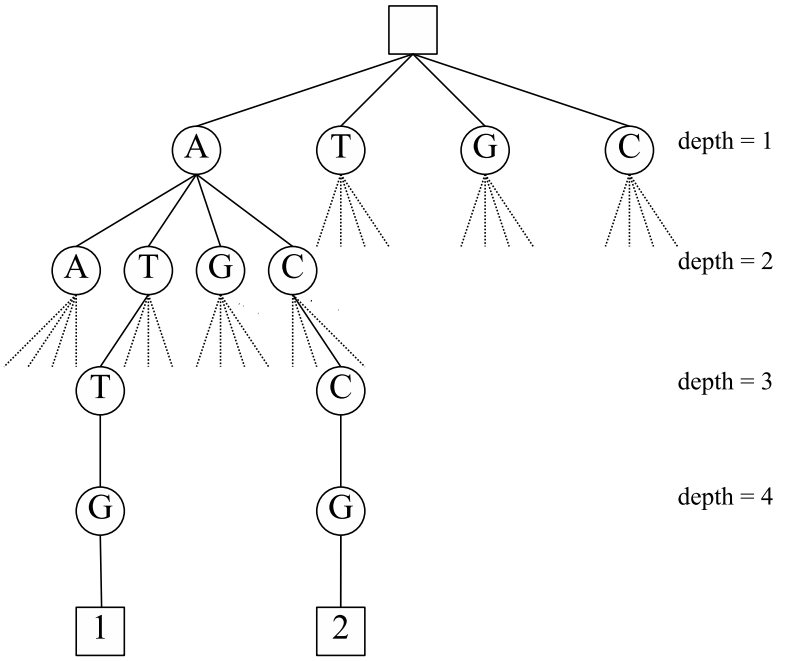


Figure 2.4: An example of trie structure for DNA reads in Clover clustering.

matches, making clustering failures more likely.

2.3.3 Trie-based Search for IDS Channel

During the Trie-based search, the Levenshtein distance between an input read and all representatives in the trie can be calculated. This allows for a distance-based comparison method except for direct symbol comparison.

Algorithm 2 Construct trie

Input: Reads $\mathbf{S} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$ and depth of trie n.
Output: The root of the created trie $\mathbf{R}$.

Initialize an empty trie $R \leftarrow \emptyset$.
for $i = 1, 2, \ldots, N$ **do**
    Get current input read $\mathbf{y}_i = \{y_1, y_2, \ldots, y_n\}$.
    Set current node to root $\mathbf{T} \leftarrow \mathbf{R}$.
    for $j = 1, 2, \ldots, n$ **do**
        if symbol y_j exists in children of $\mathbf{T}$ **then**
            Move current node $\mathbf{T} \leftarrow \mathbf{T}$.children$[y_j]$.
        else
            Construct node $\mathbf{T}$.children$[y_j]$ from $\mathbf{T}$.
            Move current node $\mathbf{T} \leftarrow \mathbf{T}$.children$[y_j]$.

	/	C	A	T	G	A
/	0	1	2	3	4	5
C	1	0	1	2	3	4
A	2	1	0	1	2	3
G	3	2	1	1	1	2
A	4	3	2	2	2	1
T	5	4	3	2	3	2

	/	C	A	T	G	A
/	0	1	2	3		
C	1	0	1	2	3	
A	2	1	0	1	2	3
G	3	2	1	1	1	2
A		3	2	2	2	1
T			3	2	3	2

Figure 2.5: An example of the Wagner–Fischer algorithm (left) and reduced method for clustering applications for a given threshold of 3 (right).

2.3.3.1 Levenshtein Distance and Calculation

The Levenshtein distance also referred to as the edit distance, is defined as the minimum number of IDS operations required to transform one sequence into another. The Wagner-Fischer algorithm is the dynamic programming algorithm for computing the Levenshtein distance efficiently. And it computes with a recurrence relation as in the Equation 2.1[16] when comparing sequence p and q, where $\mathrm{LD}(i, j)$ denotes the Levenshtein distance between $p[1, ..., i]$ and $q[1, ..., j]$. And $t(i, j)$ is 1 if $p[i]$ is equal to $q[j]$, otherwise, it is 0.

$$\mathrm{LD}(i, j) = \min\left[\mathrm{LD}(i-1, j) + 1, \mathrm{LD}(i, j-1) + 1, \mathrm{LD}(i-1, j-I) + t(i, j)\right]. \quad (2.1)$$

As shown in Figure 2.5, the Wagner-Fischer algorithm employs a dynamic programming approach to build a matrix of distances between pairs of substrings from the two input strings. The matrix is initialized with values that correspond to the distances between each substring in one string and the empty string, and vice versa. For each subsequent cell in the matrix, the minimum of three possible IDS operations is calculated, and the resulting distance is stored in the cell. At the end of the algorithm, the value in the bottom right corner of the matrix represents the minimum Levenshtein distance between the two strings. The Wagner-Fischer algorithm has a time complexity of $O(nm)$, where n and m are the lengths of the two input strings. However, the Levenshtein distance is a pair-wise metric and computationally expensive to calculate, especially for large datasets.

For clustering applications, the need for sequence comparison is mainly to determine if two sequences lie in a certain threshold. Eduard et al. [17] proposed a method for reducing the computational cost of calculating the Levenshtein distance in clustering. Their approach involves excluding cells in the distance matrix whose values would exceed a pre-defined threshold, as illustrated in Figure 2.5. And for the matrix update, the program can only maintain a flip L-shape cache, and the search is aborted when all the cells of the cache are large than the threshold.

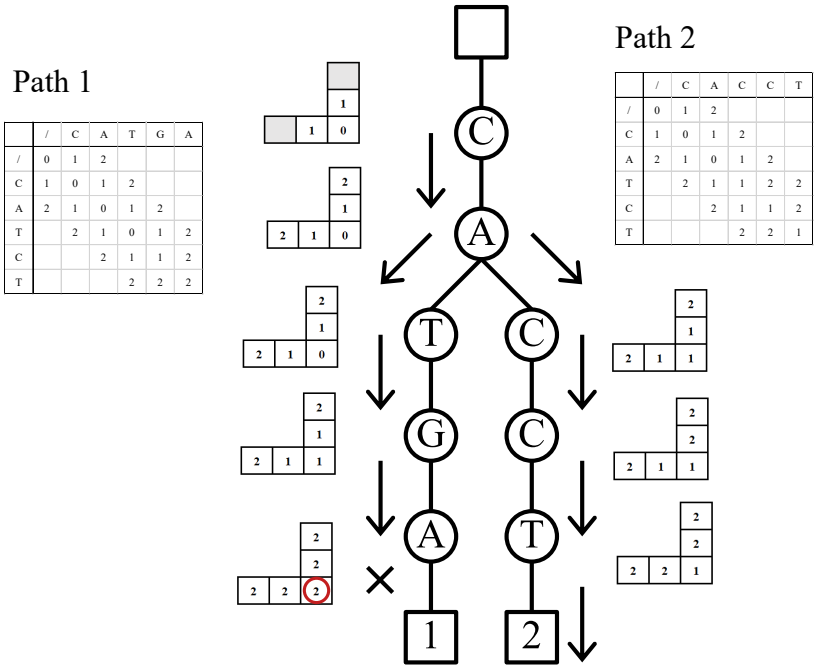


Figure 2.6: An example of the poucet search for an input sequence "CATCT", with Levenshtein distance threshold of 1.

2.3.3.2 Poucet Search on Trie

Poucet search[17] is a method working on trie and finding which pairs of input sequences and representatives lie within a given Levenshtein distance threshold. As an example shown in Figure 2.6, we assume an input sequence "CATCT" is entered into the trie with a Levenshtein distance threshold of 1. During the traversal, the flip L-shape cache is kept updated and the current minimum Levenshtein distance is equal to the value of the center cell. When the search reaches the branch "CATGA", the minimum Levenshtein distance exceeds the threshold, so the search turns to another branch from node "A" at depth 2. As nodes that match the input sequence are visited first, poucet search behaves as a deep-first searching process.
Poucet search is an algorithm that enables efficient comparison of all the sequences in the trie by utilizing a fixed Levenshtein distance threshold. Firstly, it optimizes the search path with a deep-first strategy. Secondly, it automatically terminates the search process in a route when the Levenshtein distance exceeds the given threshold.

3

Clustering Quality Metrics

For reliable decoding in DNA storage, clustering is necessary to preserve the purity of the sequences within the same cluster, while retaining the number of noisy copies amplified from the synthesized sequences as much as possible. Thus, in this study, the purity and accuracy metrics are considered to evaluate the quality of clustering.

3.1 Purity

The purity indicates the degree to which each cluster is mixed. A predicted cluster with labels only from one synthesized sequence has high purity, indicating a high quality of clustering. The purity is calculated by

$$Purity\left(\mathbf{C},\widetilde{\mathbf{C}}\right)=\frac{1}{N}\sum_{i}\max_{\pi}\left\{\left|\widetilde{\mathbf{C}}_{\pi(i)}\subseteq\mathbf{C_i}\right|\right\}, \tag{3.1}$$

where the max is over all mappings $\pi:\left\{1,2,...,|\widetilde{\mathbf{C}}|\right\}\longrightarrow\left\{1,2,...,\max\left(|\mathbf{C}|,|\widetilde{\mathbf{C}}|\right)\right\}$, the $\mathbf{C}$ is true clustering and $\widetilde{\mathbf{C}}$ is the output result of the algorithm. A shortcut of the purity metric is if a true cluster has been split into many pieces by algorithms, the purity is also high but the actual clustering quality for decoding purposes is bad, as the number of sequences for decoding is reduced.

3.2 Accuracy

Another measure evaluates the percentage of successfully retrieved clusters from the input reads. This metric, proposed by Cyrus et al. [18], is calculated by,

$$Accuracy(\mathbf{C},\widetilde{\mathbf{C}})=\max_{\pi}\frac{1}{|\mathbf{C}|}\sum_{i}1\{|\widetilde{\mathbf{C}}_{\pi(i)}\subseteq\mathbf{C}_i|\text{ and }|\widetilde{\mathbf{C}}_{\pi(i)}\cap\mathbf{C}_i|\geq\gamma|\mathbf{C}_i|\}. \tag{3.2}$$

The accuracy requires that every counted cluster $\widetilde{C}_{\pi(i)}$ must be a subset of one of the true clusters and at least has γ-fraction of elements. And it indicates how many clusters satisfy a fraction threshold compared with the true clustering.

4

Methods

As trie traversal can easily be corrupted when input sequences contain IDS errors, to ensure successful clustering using trie-based clustering as described in Chapter 2, detecting and correcting errors is crucial. In this chapter, the trie-based clustering algorithm is optimized for the scenarios that IDS errors are present in reads.

4.1 Depth-Limited Search

In this section, the depth-limited search (DLS) is presented to find possible branches for unmatched reads without exhaustive searching of all branches in the trie structure when there is corruption. The algorithm performs a deep-first search from an intermediate node and investigates the context of symbols within a limited depth to estimate an IDS error.

Algorithm 3 outlines the DLS approach. Initially, the input read $\mathbf{y}$ is added to a queue, and the traversal starts with the original input $\mathbf{y}$. During the traversal, an unmatched event occurs when there is not a matching node with the next symbol of the read in the children of the current node. And in response, the DLS searches for possible sub-branches within the depth d from the current node. And if the other d symbols of the sequence exactly match the consequent d nodes while assuming there is an insertion, deletion or substitution at the next symbol, the assumpted IDS error is then corrected. An error-corrected input $\tilde{\mathbf{y}}$ is then added to the queue as a new trial, along with the accumulated number of errors. This number enables early termination of searching when it is larger than a threshold θ, reducing paths to search.

To illustrate the DLS method, we consider the example in Figure 4.1. Suppose there are three representative sequences in the trie and an input read of "CAGAT" is given with one deletion error at the third position. During the search, the algorithm cannot find "G" in node "T"'s children. However, when comparing the subsequence "GA" of the input with branch "TGA", it can be inferred that there is a deletion error. The corrected input "CATGA" is then used to search in the trie again. This time, the input sequence will be matched to the branch with label 2 since the corrected input "CATGA" matches this branch within the error threshold.

Algorithm 3 Depth-limited search (DLS)

Input: Read $\mathbf{y}$, a trie's root $\mathbf{R}$, the trie depth n, depth limit d, threshold θ.
Output: The mapping $\pi : \mathbf{y} \mapsto c_k$.

Set current node to root $\mathbf{T} \leftarrow \mathbf{R}$.
Initialize queue $\mathbf{q} \leftarrow \emptyset$.
Initialize an initial queue item $[\mathbf{y}, 0]$. △ The second element will record errors.
$\mathbf{q} \leftarrow \emptyset$
while $\mathbf{q}$ is not empty **do**...
    Get reads and error numbers $\mathbf{y}, e$ from $\mathbf{q}$.
    if e>θ **then** Break △ Terminate try when the error number exceeds the threshold.
    for $i = 1, 2, \ldots, n$ **do**
        if $\mathbf{y}_i$ exists in $\mathbf{T}$.children **then**
            Move current node $\mathbf{T} \leftarrow \mathbf{T}$.children$[\mathbf{y}_i]$.
        else
            for Branch $\{\mathbf{T}_1, \ldots, \mathbf{T}_{d+1}\}$ under current node **do**
                if $\mathbf{T}_1$.children$[y_{i+1}]$, ..., $\mathbf{T}_d$.children$[y_{i+d}]$ exist **then**
                    Fix the insertion error $\widetilde{\mathbf{y}} \leftarrow f(\mathbf{y})$.
                if $\mathbf{T}_1$.children$[y_i]$, ..., $\mathbf{T}_d$.children$[y_{i+d}]$ exist **then**
                    Fix the deletion error $\widetilde{\mathbf{y}} \leftarrow f(\mathbf{y})$.
                if $\mathbf{T}_2$.children$[y_{i+1}]$, ..., $\mathbf{T}_{d+1}$.children$[y_{i+d+1}]$ exist **then**
                    Fix the substitution error $\widetilde{\mathbf{y}} \leftarrow f(\mathbf{y})$.
                Update $\mathbf{q}$ with $\mathbf{q} \leftarrow [\widetilde{\mathbf{y}}, e + 1]$
    if $i == n$ **then** Return mapping $\pi : \mathbf{y} \mapsto \widetilde{c_k}$ that extracted from current $\mathbf{T}$.
Return mapping $\pi : \mathbf{y} \mapsto \mathbf{y}$. △ Not found a matched representative.

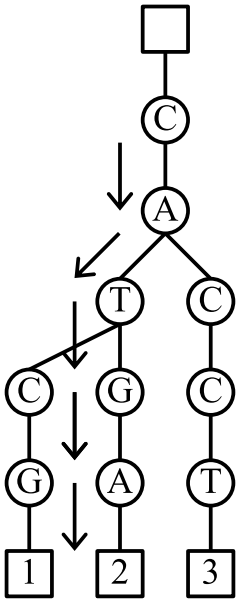


Figure 4.1: An example of DLS algorithm that assumes an input sequence "CAGAT", with a depth limitation of 2.

4.2 Cluster Merging using Poucet Search

In the proposed DLS method, continuous IDS errors will cause the clustering of the current sequence to fail. Additionally, interference can occur between branches that share a certain length of the prefix. Figure 4.2(a) illustrates an example of interference, where an input sequence "CATCT" has a substitution error at the third symbol from "C" to "T". With DLS, the search fails because of two continuous unmated nodes after the prefix "CAT", and node "T" at a depth of 3 becomes the interference node. To address this issue, we first use the poucet search described previously in Section 2.3.3.2 to identify the interference node with a threshold of 1, as shown in Figure 4.2(b). Then, we add a new branch from the interference node to eliminate interference as shown in Figure 4.2(c), therefore for future input "CAGAT", the search will be successful during the DLS.

As shown in Algorithm 4, to merge clusters, we implemented a secondary trie $\mathbf{R_s}$ specifically for storing representatives used in cluster merging, and the main trie is extended by additional branches learned from the secondary trie. The poucet search algorithm was employed to identify potential matches for not-clustered sequences in the secondary trie based on a specified Levenshtein threshold θ_s. The identified sequence was subsequently added to the main trie $\mathbf{R_m}$, with the same label on the leaf that had been detected in the secondary trie. This approach ensures that even if an input read cannot be matched to a branch in the current main trie with DLS, it can still be grouped into the correct cluster and has the potential to merge with other similar input sequences to the correct cluster.

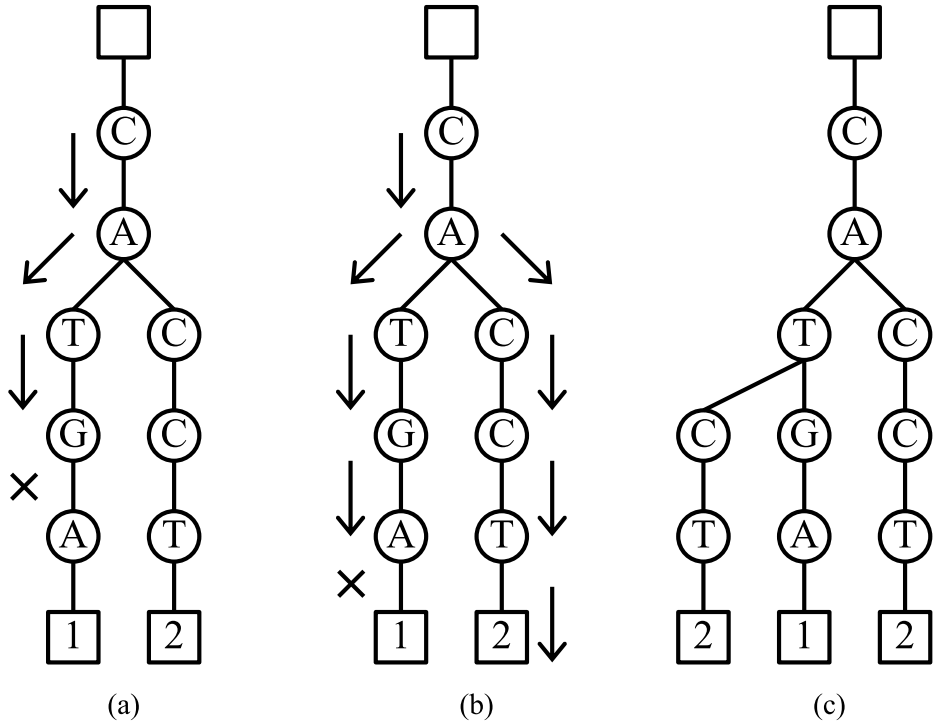


Figure 4.2: An example of proposed DLSM algorithm that assumes an input sequence "CATCT". (a) DLS on the main trie. (b) Poucet search on the secondary trie. (c) Update the main trie to eliminate interference.

Algorithm 4 Depth-limited search with cluster merging (DLSM)

Input: Read $\mathbf{y}$, main trie's root $\mathbf{R_m}$, secondary trie's root $\mathbf{R_s}$, the trie depth n, depth limit d, main trie's threshold θ_m, secondary trie's threshold θ_s.

Output: The mapping $\pi : \mathbf{y} \mapsto c_k$.

Compare $\mathbf{y}$ with the trie $\mathbf{R_m}$ with DLS as in Algorithm 3.
if A matching branch is found in the main trie $\mathbf{R_m}$ **then**
    Update and return the mapping π as in Algorithm 3.
else
    Compare $\mathbf{y}$ with the trie $\mathbf{R_s}$ with poucet search.
    if A matched branch is found in the secondary trie $\mathbf{R_s}$ within the threshold θ_s **then**
        Add $\mathbf{y}$ to the main trie $\mathbf{R_m}$ with the label found in $\mathbf{R_s}$.
        Update and return the mapping π.
    else
        Add $\mathbf{y}$ to both trie $\mathbf{R_m}$ and $\mathbf{R_s}$ with the new label.
        Update and return the mapping π.

4.3 Methods Verification

We took indexing methods from DNA fountain storage scenarios mentioned in Chapter 2 to verify our DLS and DLSM methods. We generated 10,000 sequences consisting of 16 nucleotides seed and 10 random payloads for Monte Carlo simulations, the flow chart is illustrated in Figure 4.3. First of all, the index area is designed to contain a 32-bit random value going through the interval $[1, 2^{32} - 1]$. A Galois linear-feedback shift register(GLFSR) that uses the primitive polynomial $x^{32} + x^{30} + x^{26} + x^{25} + 1$ is initialized and generates 32-bit values. And the cycle size of this GLFSR is $2^{32} - 1$ [19]. These values are monitored and those that have more than 3 homopolymers or GC-content large than 60% or lower than 40% are discarded, while other values are mapped to 16 nucleotides to be the index of the sequences. Since the reads should be truncated to 16 nucleotides for clustering, we also generated extra 10 random nucleotides as payload for each sequence to prevent the length of reads is under 16 nucleotides. In Figure 4.4, DLS and DLSM are

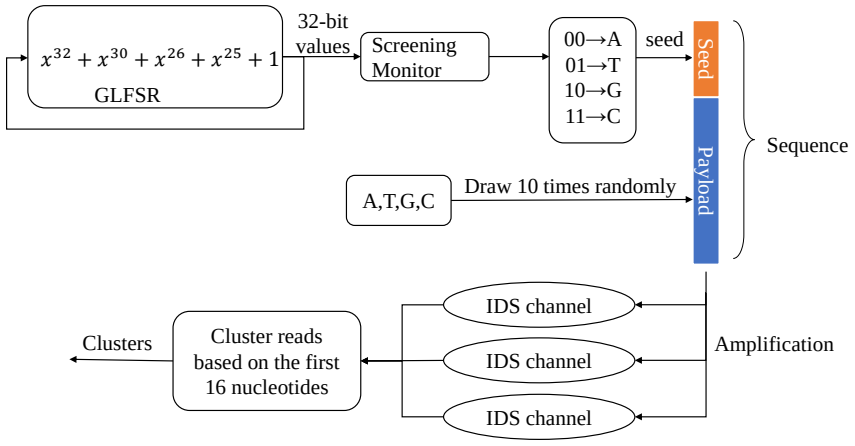


Figure 4.3: Schematic of coding and simulations.

compared in terms of the number of clusters generated, using reads generated from parallel IDS channels with parameter $p_i = p_d = p_s = 0.01$. The input sequences are generated and amplified with a factor of 20 before being transmitted over the channels, resulting in 200,000 reads. And different secondary trie thresholds θ_s are used for DLSM. The number of clusters increases with the number of input reads for both DLS and DLSM. However, DLSM generally generates fewer clusters than DLS for the same number of input reads, as it uses Levenshtein distance to merge similar clusters and eliminate interference. Figure 4.5 compares the accuracy of DLS and DLSM with varying secondary trie thresholds using the accuracy metric described in Chapter 3. When γ is high, DLSM has higher accuracy than DLS since it can cluster more sequences to the correct cluster set that DLS cannot identify. However,

when γ is lower, DLSM has lower accuracy than DLS as it introduces more error sequences to the clusters, reducing the accuracy value. Hence, our algorithms can cluster the index generated by DNA fountain's scheme, but the threshold for the secondary trie should be chosen carefully to minimize the effect of wrong clustering.

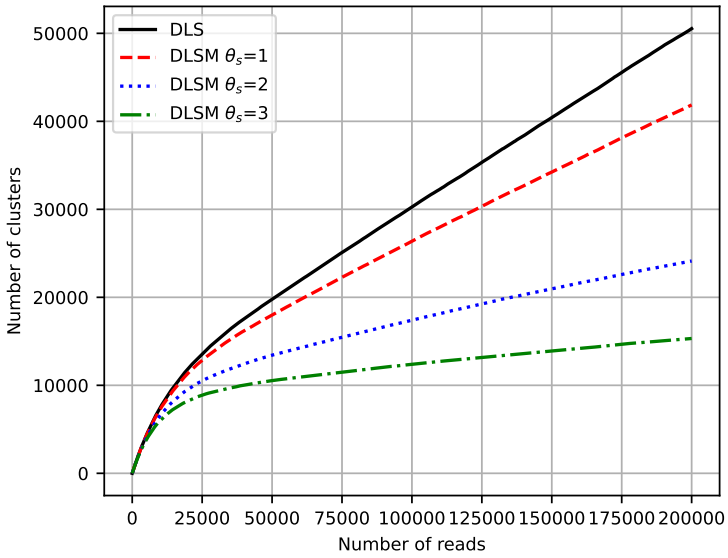


Figure 4.4: Comparison of the number of generated clusters against the number of input reads for DLS and DLSM. The main trie's threshold is set to 4 and depths are set to 16.

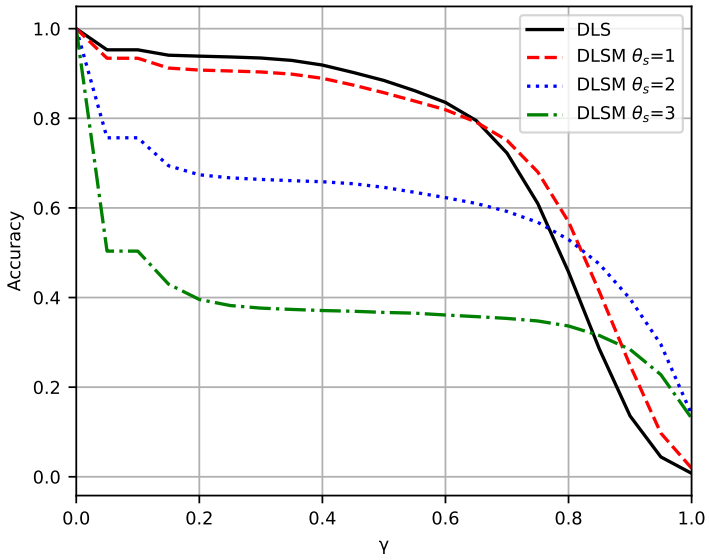


Figure 4.5: Comparison of the accuracy between DLS and DLSM. The main trie's threshold is set to 4 and depths are set to 16.

4.4 Theoretical Analysis

4.4.1 Time Complexity

The time complexity of the proposed DLS is proportional to the depth of the trie. For an input of N reads, the tries loop N times to process all the reads. For an unmatched node, there can be at most 3 children linked to this node. The search can find at most 5 corrected inputs (3 for possible substitutions, 1 for insertion and 1 for deletion) under these children. Taking an extreme case, where θ is the threshold for the DLS, d is the depth limitation of the DLS, and n is the maximum depth of the trie, IDS can take place θ times with an interval of at least d. The search visits at most 5^θ new branches or $5^\theta n$ nodes. As the maximum number of visited nodes is not affected by an increase in N, the overall time complexity is still $O(N)$.
For the DLSM, two tries are maintained at the same time, the time-consuming grows while sequences are not successfully grouped into one of the clusters and the poucet search is functional in the reference trie. After a number of reads are input, most of the representatives are added to the trie, so the time-consuming will decrease as the poucet search is functional less frequently. Considering the worst case, each input read compares all the k branches in the trie, so the time complexity of DLSM is $O(kN)$.

4.4.2 Space Complexity

For DLS and DLSM, the extreme case is that the tries are filled by n^4 branches, where n is the maximum depth of the trie, so the space complexity can be up to $O(n^4)$. In practice, for N reads, there are almost N branches in the trie considering that every read is an independent cluster, so the actual number of nodes is lower than n^4.

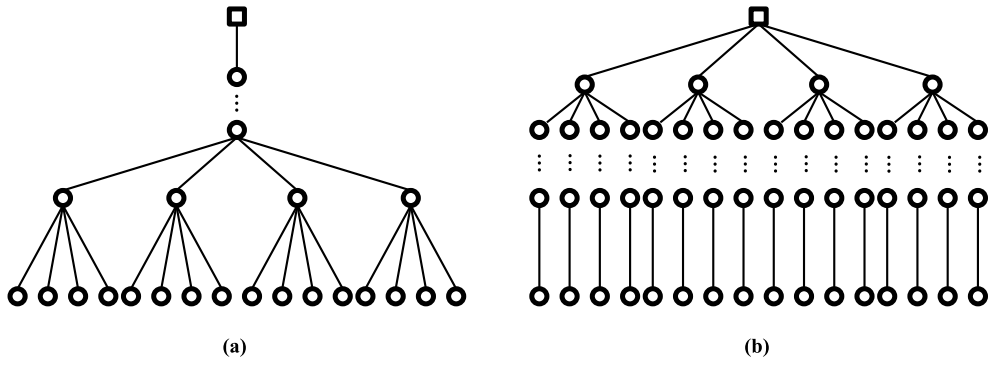


Figure 4.6: An example of tries that minimizes the number of nodes (a) and maximizes the number of nodes (b), assuming there are 16 representatives.

The maximum and minimum number of nodes after N reads input can be calculated [15]. Considering the case for filling in the trie with the minimum number of nodes,

as shown in Figure 4.6(a). The first x symbols of all the sequences can share x node at the first x layers of the trie, and the other layers of the trie should be that $(n-x)^4 \geq N$. So the first x layers can contain at least $n - \lceil \log_4 N \rceil$ nodes in total. For other $\lceil \log_4 N \rceil$ layers in the trie, the total number of nodes in this layer is the sum of geometric progression minus the number of redundant nodes, which is

$$\frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - 1 - \left(4^{\lceil \log_4 N \rceil} - N\right). \tag{4.1}$$

Thus, the minimum number of nodes can be calculated by

$$\frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - \left(4^{\lceil \log_4 N \rceil} - N\right) + n - \lceil \log_4 N \rceil - 1. \tag{4.2}$$

Considering the maximum number of nodes shown in Figure 4.6(b). In this case, we require the trie to split the branches as much as possible at the first x-th layers satisfying $x^4 \geq N$. Thus, x can be expressed as $x = \lceil \log_4 N \rceil$. Thus, the number of nodes in the first x layers is

$$\frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - \left(4^{\lceil \log_4 N \rceil} - N\right). \tag{4.3}$$

For the other $n - x$ layers, each brand consists of N nodes. Thus, the maximum number of nodes produced by the n-th sequence is

$$\frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - 4^{\lceil \log_4 N \rceil} + (n - \lceil \log_4 N \rceil)\, N. \tag{4.4}$$

Therefore, the actual number of nodes Q in the trie is bounded by

$$\begin{aligned} &\frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - \left(4^{\lceil \log_4 N \rceil} - N\right) + n - \lceil \log_4 N \rceil - 1 \leq Q \\ &\leq \frac{4^{\lceil \log_4 N \rceil + 1} - 1}{3} - 4^{\lceil \log_4 N \rceil} + (n - \lceil \log_4 N \rceil)\, N. \end{aligned} \tag{4.5}$$

With the bound of the number of nodes, memory usage is expected to increase with a growing number of reads, while the impact of the trie depth is less pronounced.

5

Results

In this chapter, we benchmark proposed algorithms on real-world DNA storage scenarios. The code implementation is available at `https://github.com/youjunl/DNA-Tree-based-Clustering`. The DLS and DLSM were implemented in C++ and bound to Python API.

5.1 Pre-processing

For benchmarking, reliable labeled datasets were extracted from the reads and synthesis references. In this section, we introduced the process of creating a labeled dataset from sequencing data.

Pair-end reads merging Paired-end reads are a type of sequencing data in which both ends of a DNA fragment are sequenced to produce two separate reads for each fragment. Merging paired-end reads can improve the completeness of the sequencing data. The process of merging paired-end reads involves aligning the two reads from each fragment and then merging them together. From each sequencing dataset, forward reads and reversed reads are split into two different files. And we use the software tool PEAR [20] for merging these reads. And after that, we removed unpaired reads from the sequencing dataset.

Sequences alignment Alignment is for finding the best matching reference sequences for each read in a dataset, and determining where each read aligns with the reference sequence(s). This is done by comparing the sequence of each read to the reference sequence(s) using a similarity score and identifying the best alignment(s) based on this score. Alignment is processed based on the paired reads with bowtie2 method [21]. The reference sequence in the FASTA file is first labeled by bowtie2, and the paired reads are then aligned and output the SAM file consisting of alignment results. Finally, matched sequences and their labels are converted into a text file for the clustering tools.

Labeled datasets In the alignment results, each retained read can be matched to one of the sequences in the references. So, all the reads that point to the same indexed sequences can be treated as a cluster. We split the alignment results into the reads file and label file for clustering and benchmarking respectively, while the reads file contains the indexes of sequences and the symbols of the sequences, the label file contains the indexes of sequences and the clustering label of the sequences. Also, we removed the cluster with size 1 from the dataset, which was not treated as a cluster.

5.2 Sequencing Data

Three DNA sequencing datasets from different DNA storage scenarios mentioned in Chapter 2 are used for benchmarking. Their specifications are shown in table 5.1. Also, we show the characterization of the labeled datasets extracted from these datasets in Figure 5.1. All of these datasets have a different distribution of amplified numbers, but all exhibit an approximately normal distribution. We also count the fraction of reads that contain errors in each underlying cluster. It is shown that most of the clusters contain more than 10% of erroneous reads. For the ERR1816980 and ID20 dataset, all the cluster contains erroneous reads, which make it harder for identifying the representatives for many algorithms during the clustering.

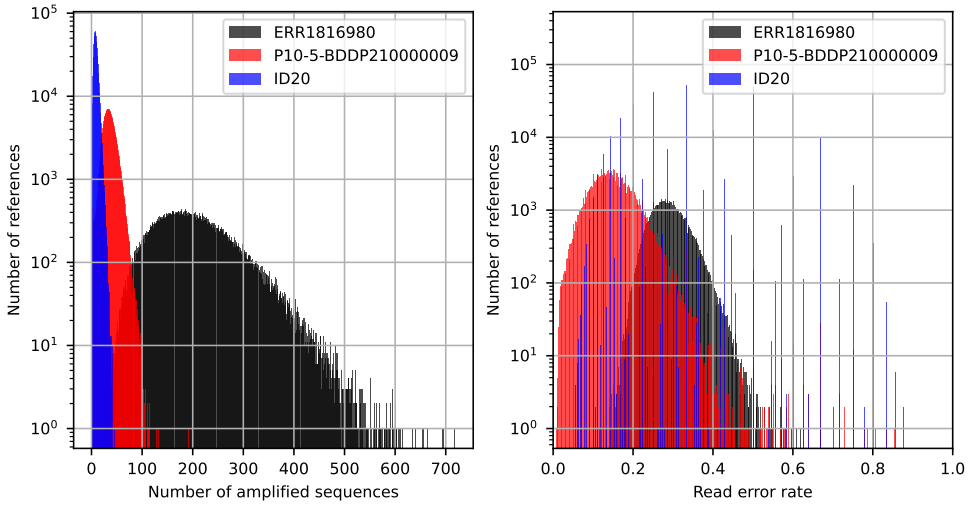


Figure 5.1: Characterization of labeled datasets, (left) reference amplification distribution, (right) error rate distribution.

Table 5.1: An overview of DNA sequencing data for benchmark.

Datasets	ERR1816980	P10-5-BDDP210000009	ID20
File size	7.5 MB	6.8 MB	9.5 MB
Strands synthesized	72,000	210,000	607,150
Aligned clusters	72,000	208,950	487,225
Reads number	15,787,115	16,605,182	15,719,903
Aligned reads number	14,654,646	16,253,050	2,128,653
Synthesis Length	200	200	150
Sources	Erlich et al. [7]	Song et al. [8]	Oganick et al. [9]

5.3 Parameters

For the benchmark, the parameters for DLS and DLSM we used are shown in Table 5.2. The depth of the trie is set to 20 to achieve higher accuracy, which can include the full index area and part of the payload. And the depth limitation is set to 4 to reduce wrong clustering. The main trie's threshold is set to 4, and the secondary trie's threshold is set to 1 to reduce time-consuming. For ERR1816980 and ID20 dataset, the starting symbol for the traversal is the first symbol of each reads because the primers have been removed in advance. For the P10-5-BDDP210000009 dataset, in which primers of length 24 nucleotides in the head have not been removed, the traversal started from the 25th nucleotides at each read. For Starcode, we chose built-in message-passing clustering methods and a Levenshtein distance threshold of 4 in order to get the best accuracy while avoiding overflow of memory. For MMSeqs2 [22], we used the default setup.

Table 5.2: Parameters for DLS and DLSM.

Datasets	ERR1816980	P10-5-BDDP210000009	ID20
Main trie threshold	4	4	4
Secondary trie threshold	1	1	1
Trie depth	20	20	20
Depth limitation	4	4	4

5.4 Purity Results

The purity results for the three datasets are compared in Table 5.3. All the algorithms achieve very high purity (more than 98%) for all scenarios except Clover fails on the P10-5-BDDP210000009 dataset. We also found that the purity of the DLSM method is consistently slightly lower than that of the DLS method. This suggests that many reads are incorrectly assigned to the wrong cluster. However, the frequency of such errors is negligible and does not significantly affect the overall clustering performance. Notes that our methods use only 20 nucleotides of each read. Thus, this result evaluates that our method is able to well separate sequences with limited area for these DNA storage scenarios.

Table 5.3: Comparison of purity on three sequencing datasets.

Datasets	ERR1816980	P10-5-BDDP210000009	ID20
Proposed DLS	0.9999	0.9982	0.9880
Proposed DLSM	0.9993	0.9979	0.9849
Clover	0.9999	0.2901	0.9887
Starcode	**1**	**0.9991**	**0.9999**
MMseqs2	**1**	0.9901	**0.9999**

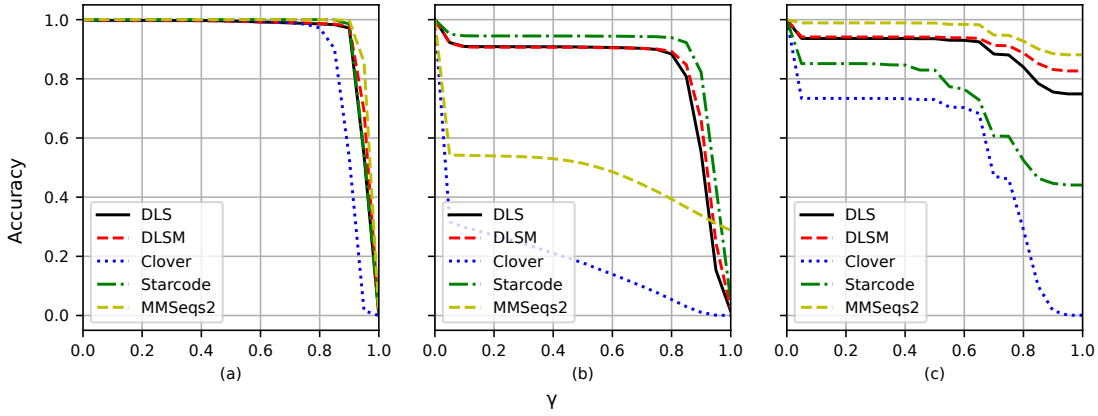


Figure 5.2: Accuracy comparison between Clover and proposed methods on three sequencing datasets (a) ERR1816980, (b) P10-5-BDDP210000009 and (c) ID20.

5.5 Accuracy Results

In Figure 5.2, we compared the accuracy for γ ranges from 0 to 1. Our proposed algorithms showed similar accuracy to other clustering algorithms for the ERR1816980 dataset. However, as an enhancement method based on Clover, our algorithms were able to retain more sequences when the γ value was greater than 0.8. For the P10-5-BDDP210000009 dataset, Starcode and our algorithms were able to cluster more sequences correctly compared to the other algorithms. In the case of the ID20 dataset, the proposed algorithms were also in second and third place for the highest accuracy. Overall, our proposed methods have demonstrated their ability to group more than 80% amplified sequences for most of the synthesis data in all three datasets. Furthermore, the proposed methods have consistently achieved high accuracy, ranking in second and third place for the highest accuracy among the compared clustering algorithms.

5.6 Runtime and Memory Usage

The runtime of the proposed clustering algorithms for three datasets is presented in Table 5.4. The proposed methods exhibit the highest speed in clustering these datasets. The relationship between the runtime of the proposed algorithms and the number of reads is also benchmarked and presented in Figure 5.3. The time complexity of DLS is linear with the number of reads as expected. The DLSM would take much more compared with DLS, and time-consuming is not growing linearly as DLS. This is because DLSM identifies most of the clusters when a large number of reads are input, which increases the success rate of DLS due to cluster merging and interference elimination performed by DLSM in advance.
The memory usage for our proposed algorithms is shown in Figure 5.4, plotted against the number of reads sampled from ERR1816980. The graph indicates that memory usage grows rapidly at first, until about 10,000 reads, as new clusters are frequently formed during this interval. However, afterwards, the memory usage

increases linearly, as the number of unclustered reads grows and is appended to the representative set.
The runtime and memory tests conducted in this section demonstrate that our proposed algorithm is efficient in terms of time, as it implements linear time complexity for large-scale datasets. However, in terms of memory usage, the algorithm becomes less effective as the number of representatives grows.

Table 5.4: Runtime comparison for three sequencing datasets (in seconds).

Datasets	ERR1816980	P10-5-BDDP210000009	ID20
Proposed DLS	**87**	**99**	**12**
Proposed DLSM	122	158	41
Clover	151	252	64
Starcode	238	382	259
MMseqs2	246	291	123

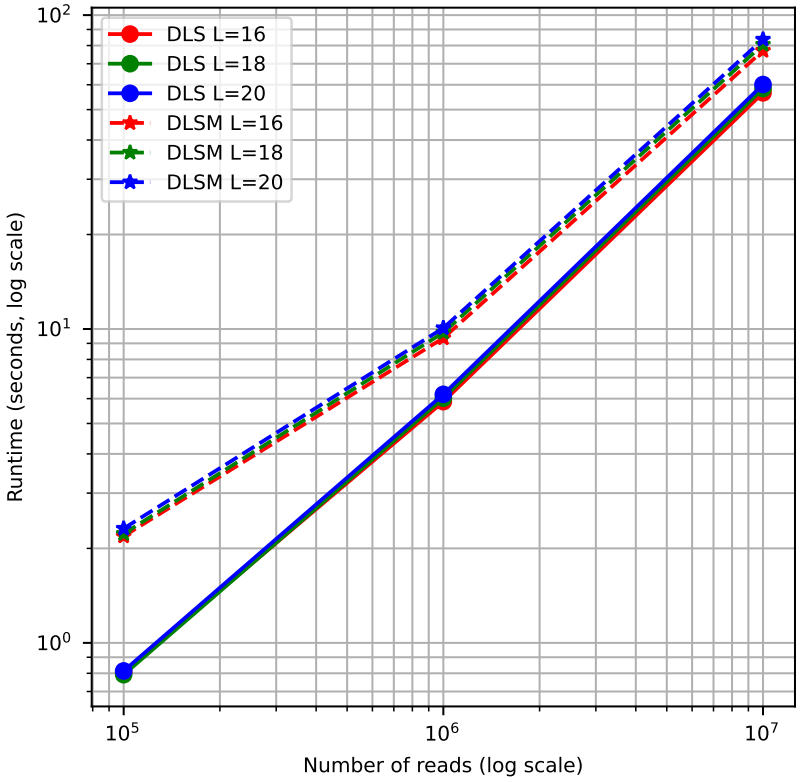


Figure 5.3: The time-consuming for proposed DLS and DLSM on different numbers of reads sampling from ERR1816980.

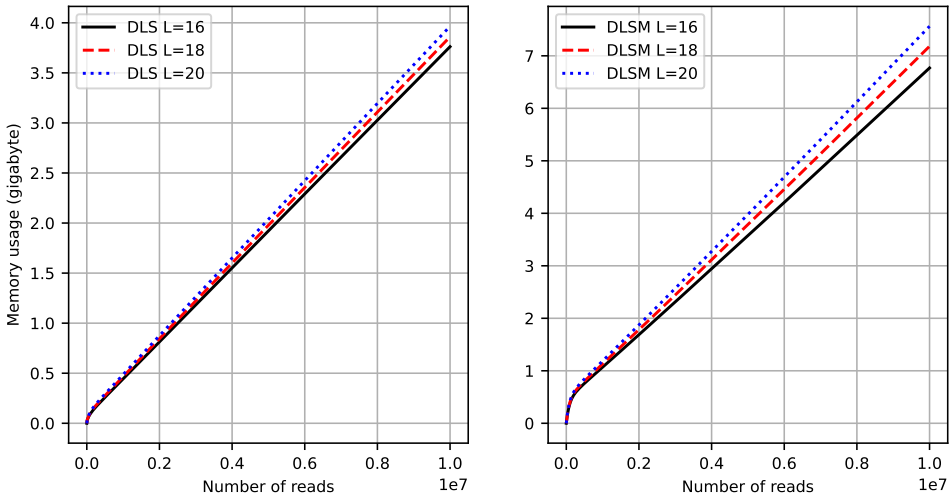


Figure 5.4: The memory usage for proposed DLS and DLSM on different numbers of reads sampling from ERR1816980.

6

Discussion

6.1 Indexes Optimization for Clustering

The Levenshtein distance among the indexes affects clustering results. In Chapter 5, the results show that there was not a significant improvement observed from using the DLSM algorithm compared to the DLS algorithm. This can be attributed to the fact that the Levenstein distance is not typically considered in the design of the indexes in these works, because of the computational complexity of finding a large index table with a certain minimum Levenshtein distance. Consequently, when DLSM uses Levenshtein distance for searching, wrong clustering reduces both the purity and accuracy. As depicted in Figure 4.5, it can be observed that in the low γ region, the accuracy decreases when the search threshold θ_s is high.

Algorithm 5 Create index table with a minimum Levenshtein distance of τ

Input: Number of indexes k, the trie's root $\mathbf{R_s}$ and the trie depth n.
Output: The index table $\mathbf{I}$.

1: Initialize an empty set $\mathbf{I} \leftarrow \emptyset$.
2: Initialize GLFSR initial state s_0.
3: **for** $i = 1, 2, \ldots, k$ **do**
4: Get output $\mathbf{y}$ from the GLFSR.
5: **while** poucet search with $\mathbf{y}$ and $\mathbf{R_s}$ success with threshold τ **do**...
6: Move the GLFSR to the next state.
7: Get output $\mathbf{y}$ from the GLFSR.
8: Add $\mathbf{y}$ to trie $\mathbf{R_s}$.
9: Add $\mathbf{y}$ to index table $\mathbf{I}$.

However, the trie structure and poucet search discussed in Section 2.3.3.2 can provide an efficient method to generate an index table with a certain minimum Levenshtein distance. With the previous simulation method in Figure 4.3, we screen the output of the Galois Linear Feedback Shift Register (GLFSR). As shown in Algorithm 5, the index table is initialized as an empty set for storing the indexes and a trie is created for the index comparison. As the Galois Linear Feedback Shift Register (GLFSR) generates indexes, each index is searched in the trie. If there is no existing index within the distance threshold that has a Levenshtein distance with the generated index, the output from the current state is added to the index table and trie. Otherwise, the GLFSR moves to the next state and generates new indexes. This process continues until a new index is found that meets the distance thresh-

old. Finally, the index table is effectively determined with the desired minimum Levenshtein distance.

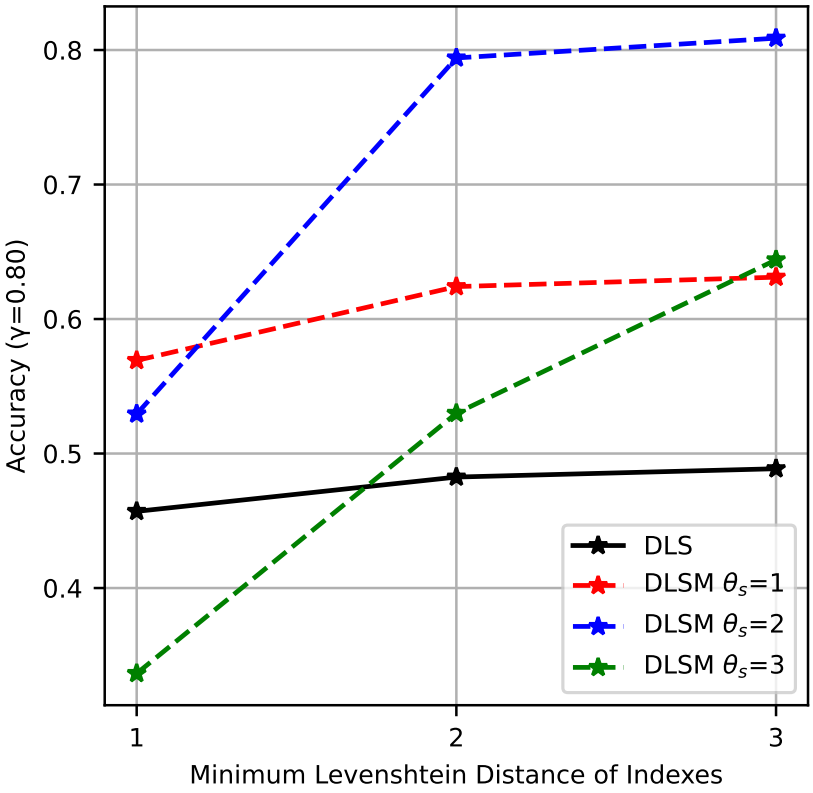


Figure 6.1: The accuracy of clustering methods with different index tables that have different minimum Levenstein distances when $\gamma = 0.8$.

Figure 6.1 presents the simulated accuracy results obtained using different index tables with varying minimum Levenshtein distances, with a γ value of 0.8. When the minimum Levenshtein distance of the index table is 1, DLSM improves the accuracy of DLS by 10% when using a search threshold θ_s of 1. However, the accuracy decreases with larger thresholds. On the other hand, when the minimum Levenshtein distance of the index table is increased to 2 or 3, DLSM achieves a substantial accuracy improvement of over 30% by setting the search threshold to 2. This observation validates that significant accuracy enhancements can be achieved by increasing the minimum Levenshtein distance of the index table when using the DLSM algorithm.

6.2 Threshold Selection for DLSM

Figure 6.1 also shows that different search thresholds θ_s yield significantly different accuracy results. As shown in Figure 6.2, the comparison of accuracy across different thresholds shows the optimal selections vary depending on the index tables with

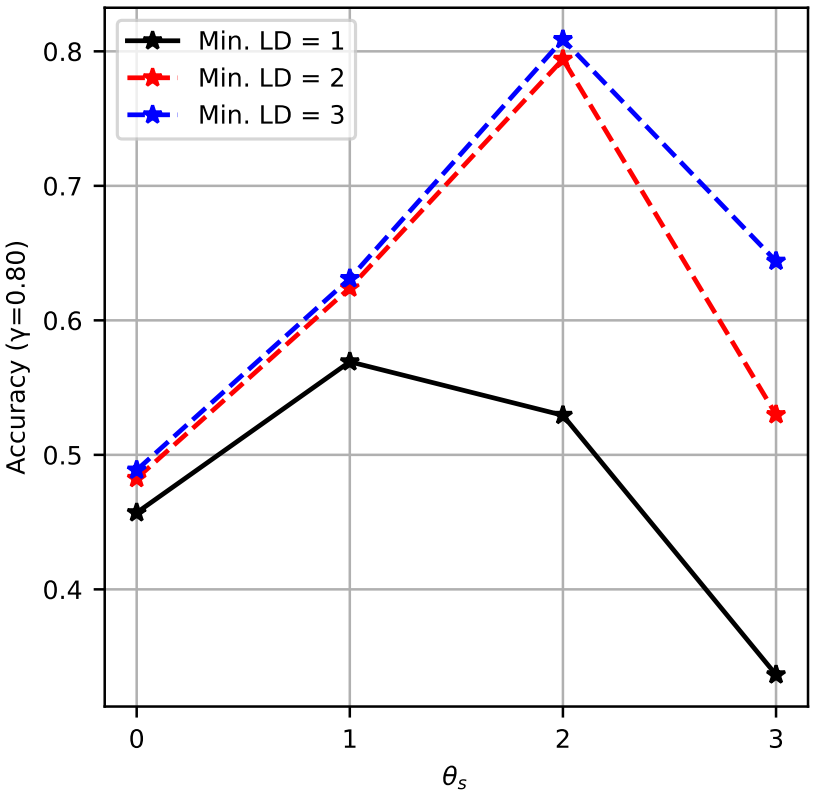


Figure 6.2: The accuracy of DLS ($\theta_s = 0$) and DLSM ($\theta_s > 0$) using different search thresholds.

different minimum Levenshtein distances. When considering an index table without a minimum Levenshtein distance requirement, the indexes only need to be different from each other, so the minimum Levenshtein distance is 1. In this case, optimal accuracy is obtained when θ_s is set to 1. For index tables with a minimum Levenshtein distance of 2 or 3, the best accuracy is achieved with θ_s set to 2. These results show that optimizing the search threshold in DLSM is crucial for achieving good clustering results. And setting θ_s to 1 is a reasonable choice when benchmarking DLSM with real-world datasets that do not have optimized indexes with a minimum Levenshtein distance in Chapter 5.

7
Conclusion

In this thesis, we began by exploring the DNA storage model and identified indexing technology as a potential area for clustering. Our methods improved a trie-based clustering method called Clover, an enhancement of traditional incremental clustering methods for DNA biological analysis. As Clover only considers substitutions, we proposed a depth-limited search method to further improve the capability to match between the input sequence and trie with the presence of IDS. The new DLS methods were validated to have higher stability and accuracy on real-world datasets compared to Clover algorithms. We also developed the DLSM method, which utilizes poucet search to merge clusters during trie updates. The DLSM method provides a second check for each read and is verified that it can cluster more sequences than DLS. Benchmarks showed that DLS takes the least time for clustering compared to three other state-of-the-art DNA clustering methods, and provides comparable clustering results with linear scalability to the number of reads, making it suitable for large-scale DNA storage datasets. DLSM, although took more time for clustering, can improve the accuracy of clustering to some extent. And by using an indexing scheme optimized by the minimum Levenshtein distance, DLSM has the potential to further improve clustering accuracy.

Memory usage is a concern in our proposed methods, which can be taken into account for further work. Although the trie structure has compressed the size of the representatives set, The memory usage was predicted to be tens of gigabytes when clustering millions of reads with DLSM. There are two potential ways to address this issue. One is to combine two tries of DLSM, so DLS and poucet search are performed on the same trie. But the problem with this scheme is the additional branches that DLSM adds to the trie would increase paths to poucet search and runtime. Another method is to compress trie structures, such as double-array trie [23], but the challenge is that it is slow to update the branch of trie as the trie nodes need to be sorted in consecutive memory. Another potential improvement for current methods is to optimize the average Levenshtein distance of the indexing area so that it can be better separated by the poucet search.

In summary, this thesis proposes an efficient improvement of incremental clustering for DNA storage, which makes us easier to cluster large-scale DNA storage reads in the future.

Bibliography

[1] Y. Dong, F. Sun, Z. Ping, Q. Ouyang, and L. Qian, "DNA storage: research landscape and future prospects," *National Science Review*, vol. 7, no. 6, pp. 1092–1107, Jan. 2020.

[2] C. Wang, G. Ma, D. Wei, *et al.*, "Mainstream encoding–decoding methods of dna data storage," *CCF Transactions on High Performance Computing*, vol. 4, no. 1, pp. 23–33, Mar. 2022.

[3] W. Li and A. Godzik, "Cd-hit: A fast program for clustering and comparing large sets of protein or nucleotide sequences," *Bioinformatics*, vol. 22, no. 13, pp. 1658–1659, May 2006.

[4] R. C. Edgar, "Search and clustering orders of magnitude faster than blast," *Bioinformatics*, vol. 26, no. 19, pp. 2460–2461, Aug. 2010.

[5] T. Rognes, T. Flouri, B. Nichols, C. Quince, and F. Mahé, "Vsearch: A versatile open source tool for metagenomics," *PeerJ*, vol. 4, e2584, Oct. 2016.

[6] B. T. James, B. B. Luczak, and H. Z. Girgis, "Meshclust: An intelligent tool for clustering dna sequences," *Nucleic acids research*, vol. 46, no. 14, e83, May 2018.

[7] Y. Erlich and D. Zielinski, "Dna fountain enables a robust and efficient storage architecture," *Science*, vol. 355, no. 6328, pp. 950–954, Mar. 2017.

[8] L. Song, F. Geng, Z.-Y. Gong, *et al.*, "Robust data storage in dna by de bruijn graph-based de novo strand assembly," *Nature Communications*, vol. 13, no. 1, p. 5361, Sep. 2022.

[9] L. Organick, S. D. Ang, Y.-J. Chen, *et al.*, "Random access in large-scale dna data storage," *Nature biotechnology*, vol. 36, no. 3, pp. 242–248, Mar. 2018.

[10] R. N. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. J. Stark, "Robust chemical preservation of digital information on dna in silica with error-correcting codes," *Angewandte Chemie International Edition*, vol. 54, no. 8, pp. 2552–2555, Feb. 2015.

[11] N. Goldman, P. Bertone, S. Chen, *et al.*, "Towards practical, high-capacity, low-maintenance information storage in synthesized dna," *Nature*, vol. 494, no. 7435, pp. 77–80, Feb. 2013.

[12] R. Heckel, G. Mikutis, and R. N. Grass, "A characterization of the dna data storage channel," *Scientific reports*, vol. 9, no. 1, pp. 1–12, Jul. 2019.

[13] M. Davey and D. Mackay, "Reliable communication over channels with insertions, deletions, and substitutions," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 687–698, Feb. 2001.

[14] J. H. Do, D. Choi, *et al.*, "Clustering approaches to identifying gene expression patterns from dna microarray data," *Molecules and cells*, vol. 25, no. 2, p. 279, Mar. 2008.

[15] G. Qu, Z. Yan, and H. Wu, "Clover: Tree structure-based efficient dna clustering for dna-based data storage," *Briefings in Bioinformatics*, vol. 23, no. 5, Aug. 2022.

[16] D. Gusfield, *Core String Edits, Alignments, and Dynamic Programming*. Cambridge University Press, 1997, pp. 215–253.

[17] E. Zorita, P. Cuscó, and G. J. Filion, "Starcode: Sequence clustering based on all-pairs search," *Bioinformatics*, vol. 31, no. 12, pp. 1913–1919, Jun. 2015.

[18] C. Rashtchian, K. Makarychev, M. Racz, *et al.*, "Clustering billions of reads for dna data storage," in *Advances in Neural Information Processing Systems 30*, Dec. 2017, pp. 3362–3373.

[19] R. Ward and T. Molteno, "Table of linear feedback shift registers," *Datasheet, Department of Physics, University of Otago*, Oct. 2007.

[20] J. Zhang, K. Kobert, T. Flouri, and A. Stamatakis, "Pear: A fast and accurate illumina paired-end read merger," *Bioinformatics*, vol. 30, no. 5, pp. 614–620, Oct. 2013.

[21] B. Langmead, C. Wilks, V. Antonescu, and R. Charles, "Scaling read aligners to hundreds of threads on general-purpose processors," *Bioinformatics*, vol. 35, no. 3, pp. 421–432, Jul. 2018.

[22] M. Steinegger and J. Söding, "Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets," *Nature biotechnology*, vol. 35, no. 11, pp. 1026–1028, Nov. 2017.

[23] J.-I. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software: Practice and Experience*, vol. 22, no. 9, pp. 695–721, Sep. 1992.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY