



OPEN

Reconstruction algorithms for DNA-storage systems

Omer Sabary[✉], Alexander Yucovich, Guy Shapira & Eitan Yaakobi

Motivated by DNA storage systems, this work presents the *DNA reconstruction problem*, in which a length- n string, is passing through the *DNA-storage channel*, which introduces deletion, insertion and substitution errors. This channel generates multiple noisy copies of the transmitted string which are called *traces*. A *DNA reconstruction algorithm* is a mapping which receives t traces as an input and produces an estimation of the original string. The goal in the DNA reconstruction problem is to minimize the edit distance between the original string and the algorithm's estimation. In this work, we present several new algorithms for this problem. Our algorithms look globally on the entire sequence of the traces and use dynamic programming algorithms, which are used for the *shortest common supersequence* and the *longest common subsequence* problems, in order to decode the original string. Our algorithms do not require any limitations on the input and the number of traces, and more than that, they perform well even for error probabilities as high as 0.27. The algorithms have been tested on simulated data, on data from previous DNA storage experiments, and on a new synthesized dataset, and are shown to outperform previous algorithms in reconstruction accuracy.

Over the past decade, significant advancements have been made in both DNA synthesis and sequencing technologies^{1–7}. These advancements have also led to the emergence of DNA-based data storage technology. A DNA storage system consists of three main components. The first is the *DNA synthesis* which generates the *oligonucleotides*, also called *strands*, that encode the data. Due to the need for high throughput production with acceptable error rates, the length of these strands is typically limited to roughly 250 nucleotides⁸. The second part is a storage container with compartments which stores the DNA strands, however without order. Finally, *DNA sequencing* is performed to read back a representation of the strands, which are called *reads* or *traces*.

Current synthesis technologies are not able to generate a single copy for each DNA strand, but only multiple copies where the number of copies is in the order of thousands to millions. Moreover, sequencing of DNA strands is usually preceded by PCR amplification which replicates the strands⁹. Hence, every strand has multiple copies and several of them are read during sequencing.

The process of encoding the user's binary data into DNA strands, as well as the reverse process of decoding, are external to the storage system. Both processes need to be designed to ensure the recoverability of the user's binary data, even in the presence of errors. The decoding step consists of three main stages known as 1. *clustering*, 2. *reconstruction*, and finally 3. *error correction*. The clustering step is the process of partitioning the noisy sequencing reads into *clusters* based on their origin, i.e., each cluster contains noisy copies of the same original designed strand. Following the clustering step, the objective is to reconstruct each individual strand using all available noisy copies. This stage is the primary focus of study in this paper. Finally, any errors that remain after the reconstruction step, including mis-clustering errors, lost strands, and other error mechanisms, should be addressed using an error-correcting code.

During the second stage of the decoding, a reconstruction algorithm is performed on each cluster to retrieve the original strand from the noisy copies in the cluster. The availability of multiple copies for each strand enables error correction during this process. In fact, this setup falls within the broader framework of the *string reconstruction problem*, which involves recovering a string based on multiple noisy copies of it. Examples of these problems include the *sequence reconstruction problem*, initially investigated by Levenshtein^{10,11}, as well as the *trace reconstruction problem*^{12–16}. Generally, these models assume that information is transmitted through multiple channels, and the decoder, equipped with all channel estimations, exploits the inherent redundancy to correct errors.

Generally speaking, the main problem studied under the paradigm of the sequence reconstruction and trace reconstruction problems is to find the minimum number of channels that guarantee successful decoding either in the worst case or with high probability. However, in DNA-based storage systems we do not necessarily have control on the number of strands in each cluster. Hence, the goal of this work is to propose efficient algorithms

The Henry and Marilyn Taub Faculty of Computer Science, Technion, 3200003 Haifa, Israel. ✉email: Omersabary@cs.technion.ac.il

for the reconstruction problem as it is reflected in DNA-based storage systems where the cluster size is a given parameter. Then, the goal is to output a strand that is close to the original one so that the number of errors the error-correcting code should correct will be minimized. We present algorithms that work with a flexible number of copies and various deletion, insertion and substitution probabilities.

In our model we assume that the clustering step has been done successfully. This could be achieved by the use of indices in the strands and other advanced coding techniques; for more details see¹⁷ and references therein. Thus, the input to the algorithms is a cluster of noisy read strands, and the goal is to efficiently output the original strand or a close estimation to it with high probability. We also apply our algorithms on simulated data, on data from previously published DNA-storage experiments^{18–20}, and on our new data. We compare the accuracy and the performance of our algorithms with state of the art algorithms known from the literature.

DNA Storage and DNA Errors

In the last decade, several DNA storage experiments were conducted^{20–23}. The processes of synthesizing, storing, sequencing, and handling DNA strands are susceptible to errors. Each step within these processes has the potential to introduce a notable number of errors independently. Furthermore, the DNA storage channel possesses distinct characteristics that set it apart from other storage media, such as tapes, hard disk drives, and flash memories. We will provide an overview of some of these dissimilarities and discuss the unique error behavior observed in DNA.

1. Deletion, insertion, and substitution errors can be introduced in both the read and synthesized strands during the processes of synthesis and sequencing.
2. Existing synthesis methods are incapable of producing a single copy for each strand. Instead, they generate thousands to millions of noisy copies, each with its own unique error distribution. Additionally, certain strands may have a substantially larger number of copies, while others may not have any copies at all.
3. The use of DNA for storage or other applications typically involves PCR amplification of the strands in the DNA pool⁹. The PCR process sometimes prefer some of the strands over others. Thus, it is possible that this process can affect the distribution of the number of copies of individual strands and their error profiles^{24,25}.
4. Longer DNA strands can be sequenced using designated sequencing technologies, e.g. PacBio and Oxford Nanopore^{20,23,26,27}. However, the error rates of these technologies can be higher, with deletions and substitutions as the most dominant errors^{28,29}.
5. Furthermore, there are emerging synthesis technologies, e.g. photolithographic light-directed synthesis^{28,30}, in which the reported error rates are up to 25%-30%.

A detailed characterization of the errors in the DNA-storage channel and more statistics about previous DNA-storage experiments can be found in^{9,29}.

This work

In this work, we introduce several novel reconstruction algorithms specifically tailored for DNA storage systems. Our primary goal is to address the reconstruction problem as it is reflected in DNA storage systems, and thus our algorithms focus on maximizing the similarity between the algorithms' outputs and the original strands. Notably, our algorithms differ from previously published reconstruction algorithms in several key aspects. First, we do not impose any assumptions on the input. This means that the input can be arbitrary and does not necessarily adhere to an error-correcting code. Second, our algorithms are not restricted to specific cluster sizes, nor do they require dependencies between error probabilities or assume zero errors at specific strand locations. Third, we have the capability to control the complexity of our algorithms, allowing us to limit their runtime when applied to real data derived from prior DNA storage experiments. Finally, given that clusters in DNA storage systems can exhibit variations in size and error distributions, our algorithms are designed to minimize the edit distance between our output and the original strand, while considering that these errors can be corrected using an error-correcting code.

Preliminaries and problem definition

We denote by $\Sigma_q = \{0, \dots, q-1\}$ the alphabet of size q and $\Sigma_q^* \triangleq \bigcup_{\ell=0}^{\infty} \Sigma_q^\ell$. The length of $\mathbf{x} \in \Sigma^n$ is denoted by $|\mathbf{x}| = n$. The *Levenshtein distance* between two strings $\mathbf{x}, \mathbf{y} \in \Sigma_q^*$, denoted by $d_L(\mathbf{x}, \mathbf{y})$, is the minimum number of insertions and deletions required to transform \mathbf{x} into \mathbf{y} . The *edit distance* between two strings $\mathbf{x}, \mathbf{y} \in \Sigma_q^*$, denoted by $d_e(\mathbf{x}, \mathbf{y})$, is the minimum number of insertions, deletions and substitutions required to transform \mathbf{x} into \mathbf{y} , and $d_H(\mathbf{x}, \mathbf{y})$ denotes the *Hamming distance* between \mathbf{x} and \mathbf{y} , when $|\mathbf{x}| = |\mathbf{y}|$. For a positive integer n , the set $\{1, \dots, n\}$ is denoted by $[n]$.

The *trace reconstruction problem* was first proposed in¹² and was later studied in several theoretical works; see e.g.^{14–16,31}. Under this framework, a length- n string \mathbf{x} , yields a collection of noisy copies, also called *traces*, $\mathbf{y}_1, \dots, \mathbf{y}_t$, where each \mathbf{y}_i is independently obtained from \mathbf{x} by passing through a *deletion channel*, under which each symbol is independently deleted with some fixed probability p_d . Suppose the input string \mathbf{x} is arbitrary. In the trace reconstruction problem, the main goal is to determine the required minimum number of i.i.d traces in order to reconstruct \mathbf{x} with high probability. This problem has two variants. In the “worst case”, the success probability refers to all possible strings, and in the “average case” (“random case”) the success probability is guaranteed for a random (uniformly chosen) input string \mathbf{x} .

The trace reconstruction problem can be extended to the model where each trace is a result of \mathbf{x} passing through a *insertion-deletion-substitution channel*. Here, in addition to deletions, each symbol can be switched with

some substitution probability p_s , and for each j , with probability p_i , a symbol is inserted before the j th symbol of \mathbf{x} . Under this setup, the goal is again to find the minimum number of channels which guarantee successful reconstruction of \mathbf{x} with high probability. For this purpose, it should be noted that there are many interpretations for the insertion-deletion-substitution, most of them differ on the event when more than one error occurred on the same index. Our interpretation of this channel is described in “Results” section.

Motivated by the storage channel of DNA and in particular the fact that different clusters can be of different sizes, this work is focused on another variation of the trace reconstruction problem, which is referred by the *DNA reconstruction problem*. The setup is similar to the trace reconstruction problem. A length- n string \mathbf{x} is transmitted t times over the *deletion-insertion-substitution channel* and generates t traces $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_t$. A *DNA reconstruction algorithm* is a mapping $R : (\Sigma_q^*)^t \rightarrow \Sigma_q^*$ which receives the t traces $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_t$ as an input and produces $\hat{\mathbf{x}}$, an estimation of \mathbf{x} . The goal in the DNA reconstruction problem is to minimize $d_e(\mathbf{x}, \hat{\mathbf{x}})$, i.e., the edit distance between the original string and the algorithm’s estimation. When the channel of the problem is the *deletion channel*, the problem is referred by the *deletion DNA reconstruction problem* and the goal is to minimize the Levenshtein distance $d_L(\mathbf{x}, \hat{\mathbf{x}})$. While the main figure of merit in these two problems is the edit/Levenshtein distance, we will also be concerned with the complexity, that is, the running time of the proposed algorithms. Due to lack of space, this paper presents only the main results for the DNA reconstruction problem. Our algorithm and results of the deletion DNA reconstruction problem can be found in the supplementary material.

Related work

This section reviews the related works on the different reconstruction problems. In particular we list the reconstruction algorithms that have been used in previous DNA storage experiments. A summary of some of the main theoretical results on the trace reconstruction problem can be found in the supplementary material.

Batu et al.¹² studied the trace reconstruction problem as an abstraction and a simplification of the multiple sequence alignment problem in bioinformatics. Here the goal is to reconstruct the DNA of a common ancestor of several organisms using the genetic sequences of those organisms. They focused on the deletion case of this problem and suggested a majority-based algorithm to reconstruct the sequence, which they referred by the *bitwise majority alignment (BMA) algorithm*. They aligned all traces by considering the majority vote per symbol from all traces, while maintaining pointers for each of the traces. If a certain symbol from one (or more) of the traces does not agree with the majority symbol, its pointer is not incremented and it is considered as a deletion. They showed and proved that even though this technique works locally for each symbol, its success probability is relatively high when the deletion probability is small enough.

Viswanathan and Swaminathan presented in³² a BMA-based algorithm for the trace reconstruction problem under the deletion-insertion-substitution channel. Their algorithm extends the BMA algorithm so it can support also insertions and substitutions. It works iteratively on “segments” from the traces, where a segment consists of consecutive bits and its size is a fixed fraction of the trace that is given as a parameter to the algorithm.

Gopalan et al.³³ used the approach of the BMA algorithm from¹² and extended to support DNA storage systems. In their approach, they considered deletion errors as was done in¹², and also considered insertion and substitution errors. Their method works as follows. They considered a majority vote per symbol as was done in¹², but with the following improvements. For any trace that its current symbol did not match the majority symbol, they used a “lookahead window” to look on the next 2 (or more) symbols. Then, they compared the next symbols to the majority symbols and classified it as an error accordingly. Organick et al. conducted a large scale DNA storage experiments in²⁰ where they successfully reconstructed their sequences using the reconstruction algorithm of Gopalan et al.³³

For the case of sequencing via nanopore technology, Duda et al.³⁴ studied the trace reconstruction problem, while considering insertions, deletions, and substitutions. They focused on dividing the sequence into homopolymers (consecutive replicas of the same symbol), and proved that the number of copies required for accurately reconstructing a long strand is logarithmic with the strand’s length. Yazdi et al. used in²³ a similar but different approach in their DNA storage experiment. They first aligned all the strands in the cluster using the multiple sequence alignment algorithm MUSCLE^{35,36}. Then, they divided each strand into homopolymers and performed majority vote to determine the length of each homopolymer separately. Their strands were designed to be balanced in their GC-content, which means that 50% of the symbols in each strands were G or C. Hence, they could perform additional majority iterations on the homopolymers’ lengths until the majority sequence was balanced in its GC-content. All of these properties guaranteed successful reconstruction of the strands and therefore they did not need to use any error-correcting code in their experiment²³. Another related characterization of the DNA storage channel was discussed in³⁷. In their work, the error model is different compared to our work, since it also includes rearrangement errors. In rearrangement errors several parts of two or more strands are stuck to each other to inform one read. The suggested reconstruction method of their work was based on de Bruijn graphs.

For the case where the error probabilities are priorly known, Bar-Lev et al.³⁸ and Srinivasavaradhan et al.³⁹ presented two different methods to solve the reconstruction problem. Srinivasavaradhan et al. based their method on trellises that model the probability to obtain the traces from any possible sequence, and then returning the sequence with the highest probability. Bar-Lev et al. presented a DNN based algorithm that is trained with labeled simulated data and then uses transformer to estimate the original sequence from its noisy copies.

Methods

This section studies the DNA reconstruction problem. Assume that a cluster consists of t traces, $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_t$, where all of them are noisy copies of a synthesized strand. This model assumes that every trace is a sequence that is independently received by the transmission of a length- n sequence \mathbf{x} (the synthesized strand) through a

deletion-insertion-substitution channel with some fixed probability p_d for deletion, p_i for insertion, and p_s for substitution. Our goal is to propose an efficient algorithm which returns \hat{x} , an estimation of the transmitted sequence x , with the intention of minimizing the edit distance between x and \hat{x} . In our simulations, we consider several values of t and a wide range of error probabilities. Additionally, we tested our suggested algorithm on new biological data sets, as well as on data from previous DNA storage experiments^{18–20,28,39}; see more details in “Results” section. Due to space limitation, some of our algorithms, as well as more simulations and examples can be found in the supplementary material.

Definitions and notations

While all previous works of reconstructing algorithms used variations of the majority algorithm on localized areas of the traces, we take a different global approach to tackle this problem. Namely, the algorithms presented in this work are heavily based on the *maximum likelihood decoder* for multiple deletion channels as studied recently in^{40,41} as well as the concepts of the *shortest common supersequence* and the *longest common subsequence*. Hence, we first briefly review the main ideas of these concepts, while the reader is referred to⁴⁰ for a more comprehensive study of this summary.

For a sequence $x \in \Sigma_q^*$ and a set of indices $I \subseteq [|x|]$, the sequence x_I is the *projection* of x on the indices of I which is the subsequence of x received by the symbols at the entries of I . A sequence $x \in \Sigma^*$ is called a *supersequence* of $y \in \Sigma^*$, if y can be obtained by deleting symbols from x , that is, there exists a set of indices $I \subseteq [|x|]$ such that $y = x_I$. In this case, it is also said that y is a *subsequence* of x . Furthermore, x is called a *common supersequence* (subsequence) of some sequences y_1, \dots, y_t if x is a supersequence (subsequence) of each one of these t sequences. The *length of the shortest common supersequence* (SCS) of y_1, \dots, y_t is denoted by $SCS(y_1, \dots, y_t)$. The set of all shortest common supersequences of $y_1, \dots, y_t \in \Sigma^*$ is denoted by $SCS(y_1, \dots, y_t)$. Similarly, the *length of the longest common subsequence* (LCS) of y_1, \dots, y_t , is denoted by $LCS(y_1, \dots, y_t)$, and the set of all longest subsequences of y_1, \dots, y_t is denoted by $LCSS(y_1, \dots, y_t)$.

Before we present the algorithm, we list here several more notations and definitions. An *error vector* of y and x , denoted by $EV(y, x)$, is a vector of minimum number of edit operations to transform y to x . Each entry in $EV(y, x)$ consists of the index in y , the original symbol in this index, the edit operation and in case the operation is an insertion, substitution the entry also includes the inserted, substituted symbol, respectively. Note that for two sequences y and x , there could be more than one sequence of edit operations to transform y to x . The edit distance between a pair of sequences is computed using a dynamic programming table and the error vector is computed by backtracking on this table. Hence, $EV(y, x)$ is not unique and can be defined uniquely by giving priorities to the different operation in case of ambiguity. That is, if there is an entry in the vector $EV(y, x)$ (from the last entry to the first), where more than one edit operation can be selected, then, the operation is selected according to these given priorities. The error vector $EV(y, x)$ also maps each symbol in y to a symbol in x (and vice versa). We denote this mapping as $V_{EV}(y, x) : \{1, 2, \dots, |y|\} \rightarrow \{1, 2, \dots, |x|\} \cup \{?\}$, where $V_{EV}(y, x)(i) = j$ if and only if the i th symbol in y appears as the j th symbol in x , with respect to the error vector $EV(y, x)$. Note that in the case where the i th symbol in y was classified as a deleted symbol in $EV(y, x)$, $V_{EV}(y, x)(i) = ?$. Furthermore, since inserted symbols appear only in x and not in y , these symbols are not represented in the mapping $V_{EV}(y, x)$, and are only referred as insertions in the error vector $EV(y, x)$. The mapping $V_{EV}(y, x)$ can also be represented as a vector of size $|y|$, where the i th entry in this vector is $V_{EV}(y, x)(i)$. The *reversed cluster* of a cluster C , denoted by C^R , consists of all the traces in C where each one of them is reversed. The reverse trace of $y_i \in C$ is denoted by $y_i^R \in C$. The symbols of each trace $y_i^R \in C^R$ are arranged in reverse order compared to how they appear in their original form $y_i \in C$. For example, for the cluster $C = \{y_1 = ACGTC, y_2 = CCGTA\}$, the reversed cluster is $C^R = \{y_1^R = CTGCA, y_2^R = ATGCC\}$.

The iterative reconstruction algorithm

We are now ready to discuss the main algorithm of this work, referred to as the *iterative reconstruction algorithm* or in short *ITR algorithm*. The ITR algorithm receives the design length n and a cluster of t noisy traces, all generated from the same design strand x . The algorithm consists of several different procedures that are used to create *candidates* from the traces. Candidates are revised version of traces from the cluster, or some other estimations that are created by the algorithm and its procedures. The revisions applied on a trace are intended to decrease the edit distance between the resulting candidate and the designed sequence x . The candidates that are being created by the algorithm's procedures are then assessed by the algorithm, and a score is given to each of them based on their number of occurrences and their average edit distance to the other traces in the cluster. Finally, the candidate with the highest score is returned by the algorithm (while ties are broken arbitrarily).

The different procedures that are used in the algorithm are given below in a high level description. For a detailed description of each of the procedures, as well as the ITR algorithm, the reader is referred to the supplementary material of this paper.

1. The Error-Vector Majority Algorithm (EV Algorithm).

Brief description. The EV algorithm selects a random trace from the cluster y_k and uses dynamic programming in a pairwise manner to compute the required vector of edit operations (insertion, deletion, or substitution) that should be taken in order to transform y_k into any of the other traces in the cluster. Then, the algorithm calculates the most common operation on each of the symbols, and if it finds out that they include insertions or substitutions, it performs these operations on y_k . Deletions are ignored by the EV algorithm, since in most of the cases they are corrected more accurately by the PP algorithm, which is described next. The revised version of y_k is considered as a candidate.

Intuition and motivation. This algorithm extends the majority vote approach that was used in the BMA algorithm¹² and in the BMA lookahead³³. By using dynamic programming, the algorithm takes into account that deletion, insertion, and substitution errors occur. Thus, it is possible to detect specific symbols that were inserted into y_k and do not appear in most of the other traces, or symbols that were substituted with others. Therefore, these two types of errors are corrected by the EV algorithm. The correction of deleted symbols is done by the PP algorithm which is described next.

Computational Complexity. The algorithm iterates on every pairing of y_k with another trace y_i , for $i \neq k$, where the cluster size is t , thus it works on $(t - 1)$ pairs. For each such pairs of traces, the algorithm uses dynamic programming to calculate the edit distance, which works in quadratic complexity with the design length n . Thus, the complexity of this algorithm is $\mathcal{O}(tn^2)$.

2. **The Pattern Path Algorithm (PP Algorithm).**

Brief description. The PP algorithm explores *patterns* in the cluster, which are subsequences that appear around the same location in two or more traces. To do so, the algorithm starts with a trace from the cluster y_k , and uses dynamic programming in order to find the longest common subsequence (LCS) of y_k and any of the other traces in the cluster. These LCSs are then used to create a directed acyclic graph that describes the different patterns, and their location together with their occurrences in the computed LCSs. The longest path in this graph induces a sequence that is then serves as a candidate.

Intuition and motivation. This algorithm exploits the LCS of y_k and any of the other traces in the cluster to detect the most common patterns in the cluster. The graph is used to represent the different patterns, their location and the number of times they appeared in each of the observed LCSs. The weight of the edges in the graph are defined by the number of times each pattern was seen in a specific location in the LCSs. Therefore, the longest path in the graph, is served as the algorithm selected candidate. The patterns that are computed by the PP algorithm also consider deletions. The pattern of a deleted symbol includes not just the deleted symbol itself but also the two symbols preceding it in the traces and the two symbols immediately following it. Moreover, since the patterns in the graph are weighted by their frequency of occurrence, we believe that the PP algorithm is able to correct deletion errors more precisely. Thus, the PP algorithm is used by the VR algorithm and the HR algorithm (described below) to correct deletion errors.

Computational Complexity. This algorithm calculates the LCS of y_k and each of the other traces in the cluster. The cluster size is t , and thus the algorithm calculates the LCS for $t - 1$ pairs. The calculation of the LCS is done with dynamic programming which takes $\mathcal{O}(n^2)$ complexity. Then, for every such a pair, the algorithm calculates the patterns that were observed in the LCS and this process is done in linear time with the length. Lastly, the algorithm creates the directed acyclic graph, in which it has at most nt vertices. Hence, the complexity of finding its longest path is linear with respect to the number of vertices and number of edges, which is at most nt^2 . To conclude, the overall complexity of this algorithm is $\mathcal{O}(n^2t + nt^2)$.

3. **The Vertical Reconstruction Algorithm (VR Algorithm).**

Brief description. The VR algorithm is invoked on each of the traces in the cluster. In each of these iterations, the VR algorithm starts with the k th trace in the cluster, y_k , and performs the following algorithms on it. First, the algorithm performs the EV algorithm to correct substitution and insertion errors in y_k and the PP Algorithm to correct deletions in y_k . The correction of these errors is done by editing y_k based on the algorithms' outputs. The revised version of y_k is then inserted into the cluster. The algorithm is invoked again on the updated cluster, with the new inserted revised version of y_k .

Intuition and motivation. This algorithm is invoked twice in two iterations on each trace in the cluster y_k (for $1 \leq k \leq t$). In the first iteration, it is performed on the original cluster, and in the second time it is invoked on the updated cluster that include the traces of the cluster with one additional trace, which is the algorithm estimation from the first iteration. This algorithm employs the PP and EV algorithms on y_k to edit it with the goal of correcting any of its errors. Then, the revised version of y_k is inserted into the cluster, and the algorithm is invoked one additional time on the updated cluster.

Computational Complexity. For any trace $y_k \in C$, the VR algorithm works in two iterations, in which the EV algorithm, the PP algorithm are performed two, four times, respectively. Thus, its complexity for a single trace is given by $6(n^2t + nt^2) = \mathcal{O}(n^2t + nt^2)$, and its total complexity is $\mathcal{O}(n^2t^2)$.

4. **The Horizontal Reconstruction (HR Algorithm).**

Brief description. Similarly to the VR algorithm, the HR algorithm performs the EV algorithm to correct substitution and insertion errors in y_k and the PP Algorithm to correct deletions in y_k , for any $y_k \in C$. However, this algorithm creates the set of candidate differently. In the HR algorithm, the cluster is updated with the addition of the revised versions of y_k after any iteration of the EV/PP algorithm (where in the VR algorithm, the cluster is updated only after performing both of them). That is, any iteration of each of these algorithms is followed by considering a new candidate, which is the revised version of y_k and adding it into the cluster. Similarly to the previous algorithm, this algorithm is invoked two times, when on the second time is performed on the updated cluster.

Intuition and motivation. This algorithm works similarly to the HR algorithm, however, it adds a few more candidates to the set of candidates. Some of the added candidates may be added more than once. Thus, the algorithm helps assessing the number of time they occur, which is crucial to the selection of the algorithm's output.

Computational Complexity. Similarly to the VR algorithm, the HR algorithms also works in two iterations of the same form, for any $y_k \in C$. Hence, its complexity is $\mathcal{O}(n^2t^2)$.

The four algorithms described above are invoked by the ITR algorithm on both the original cluster C and the reversed cluster C^R . Considering also the reversed cluster C^R can help detect and eliminate the synchronization

effect that is created when deletion errors shorten the traces. Thus, the ITR algorithm creates a set of candidates. As mentioned in the beginning of this section, the candidates are evaluated and scored by the ITR algorithm based on their length, number of occurrences and their distance with the original traces in the cluster. Finally, the ITR algorithm returns as an output the candidates with the highest score. This calculation also works in quadratic complexity, and thus the overall complexity of the algorithm is $\mathcal{O}(n^2 t^2)$. The ITR algorithm is applied twice on each cluster. This is because, for the majority of clusters, the set of candidates converges after the second iteration, and further iterations, such as the third, do not lead to improved results.

The pattern path algorithm (PP Algorithm)

In this section we present the Pattern Path algorithm, or in short the PP Algorithm. The algorithm, described also in Algorithm 1, is the main procedure of the iterative algorithm (ITR Algorithm) that corrects edit errors. Denote by w an arbitrary LCS sequence of x and y of length ℓ . The sequence w is a subsequence of x , and hence, all of its ℓ symbols appear in some indices of x , and assume these indices are given by $i_1^x < i_2^x < \dots < i_\ell^x$. It should be noted that a subsequence can have more than one set of such indices, while the number of such sets is defined as the embedding number^{42,43}. In our algorithm, we chose one of these sets arbitrarily. Furthermore, given a set of such indices $i_1^x < i_2^x < \dots < i_\ell^x$, we define the *embedding sequence* of w in x , denoted by $u_{x,w}$, as a sequence of length $|x|$ where for $1 \leq j \leq \ell$, $u_{x,w}(i_j^x)$ equals to $x(i_j^x)$ and otherwise it equals to $?$.

The *gap* of x , y and their length- ℓ LCS sequence w in index $1 \leq j \leq |x|$ with respect to $u_{x,w}$ and $u_{y,w}$, denoted by $\text{gap}_{u_{x,w}}^{u_{y,w}}(j)$, is defined as follows. In case the j th or the $(j - 1)$ th symbol in $u_{x,w}$ equals $?$, $\text{gap}_{u_{x,w}}^{u_{y,w}}(j)$ is defined as an empty sequence. Otherwise, the symbol $u_{x,w}(j)$ also appears in w . Denote by j' , the index of the symbol $u_{x,w}(j)$ in w . Recall that the sequence w is an LCS of x and y , and $u_{y,w}$ is the embedding sequence of w in y . Given $u_{y,w}$, we can define the sequence of indices $i_1^y < i_2^y < \dots < i_\ell^y$, such that $w(j') = y(i_{j'}^y)$ for $1 \leq j' \leq \ell$. Given such a sequence of indices, $\text{gap}_{u_{x,w}}^{u_{y,w}}(j)$ is defined as the sequence $y_{[i_{j'-1}^y+1:i_{j'}^y-1]}$, which is the sequence between the appearances of the j' th and the $(j' - 1)$ th symbols of w in y . Note that since $i_{j'}^y$ can be equal to $i_{j'-1}^y + 1$, $\text{gap}_{u_{x,w}}^{u_{y,w}}(j)$ can be an empty sequence. Roughly speaking, the $\text{gap}_{u_{x,w}}^{u_{y,w}}(j)$ holds every symbol that appears in y between the $(j' - 1)$ th and j' th symbols of the LCS w , based on the embedding sequence $u_{w,y}$.

The *pattern* of x and y with respect to the LCS sequence w , its embedding sequences $u_{x,w}$ and $u_{y,w}$, an index $1 \leq i \leq |x|$ and a length $m \geq 2$, denoted by $\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m)$, is defined as:

$$\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m) \triangleq (u_{x,w}(i - 1), \text{gap}_{u_{x,w}}^{u_{y,w}}(i), u_{x,w}(i), \dots, \text{gap}_{u_{x,w}}^{u_{y,w}}(i + m - 1), u_{x,w}(i + m - 1)),$$

where for $i < 1$ and $i > |x|$, the symbol $u_{x,w}(i)$ is defined as the null character and $\text{gap}_{u_{x,w}}^{u_{y,w}}(i)$ is defined as an empty sequence. The parameter m defines the length of the pattern, that is the number of embedding sequences and gaps that comprises the patterns. In our implementation of the algorithm, the length of the patterns is defined as $m = 2$.

We also define the prefix and suffix of a pattern $\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m)$ to be:

$$\text{Prefix}(\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m)) \triangleq (u_{x,w}(i - 1), \text{gap}_{u_{x,w}}^{u_{y,w}}(i), u_{x,w}(i), \dots, u_{x,w}(i + m - 2)),$$

$$\text{Suffix}(\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m)) \triangleq (u_{x,w}(i), \text{gap}_{u_{x,w}}^{u_{y,w}}(i + 1), \dots, u_{x,w}(i + m - 1)).$$

Finally, we define

$$P(x, y, w, u_{x,w}, u_{y,w}, m) \triangleq \{\text{Ptn}(x, y, w, u_{x,w}, u_{y,w}, i, m) : 1 \leq i \leq |x|\}.$$

The Pattern Path Algorithm receives a cluster C of t traces and one of the traces in the cluster y_k . First, the algorithm initializes $L[y_k]$, which is a set of $|y_k|$ empty lists. For $1 \leq i \leq |y_k|$, the i th list of $L[y_k]$ is denoted by $L[y_k]_i$. The algorithm pairs y_k with each of the other traces in C . For each pair of traces, y_k and y_h , the algorithm computes an arbitrary LCS sequence w , and an arbitrary embedding sequence $u_{y_k,w}$. Then it uses w and $u_{y_k,w}$ to compute $P(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, m)$. For $1 \leq i \leq |y_k|$, the algorithm saves $\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m)$ in $L[y_k]_i$. Then, the algorithm builds the *pattern graph* $G_{\text{pat}}(y_k) = (V(y_k), E(y_k))$, which is a directed acyclic graph, and is defined as follows.

- $V(y_k) = \{((\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m), i) : 1 \leq h \leq t, h \neq k, 1 \leq i \leq |y_k|) \cup \{S, U\}$.
The vertices are pairs of patterns and their index. Note that the same pattern can appear in several vertices with different indices i . The value $|V|$ equals to the number of distinct pattern-index pairs.
- $E(y_k) = \{e = (v, u) : v = (\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m), i), u = (\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i+1, m), i+1), \text{Suffix}(\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m)) = \text{Prefix}(\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i+1, m))\}$.
- The weights of the edges are defined by $w : E \rightarrow N$ as follows:
For $e = (v, u)$, where $u = (\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m), i)$, it holds that
 $w(e) = |\{\text{Ptn} \in L[y_k]_i : \text{Ptn} = \text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m)\}|$,
which is the number of appearances of $\text{Ptn}(y_k, y_h, w, u_{y_k,w}, u_{y_h,w}, i, m)$ in $L[y_k]_i$.
- The vertex S , which does not correspond to any pattern, is connected to all vertices of the first index. The weight of these edges is the number of appearances of the incoming vertex pattern.
- The vertex U has incoming edges from all vertices of the last index and the weight of each edge is zero.

Finally, the Pattern Path Algorithm identifies a longest path from S to U in the graph. This path induces a sequence, denoted by $\hat{\mathbf{y}}_k$, which is formed by concatenating the patterns of \mathbf{y}_k (including their gaps if such exist), that appears in the vertices of the longest path in the pattern graph. It's important to note that $\hat{\mathbf{y}}_k$ represents a modified version of \mathbf{y}_k , as it incorporates the patterns present in \mathbf{y}_k . The algorithm returns $\hat{\mathbf{y}}_k$, which is an updated rendition of \mathbf{y}_k , while also incorporating any gaps inherited from the vertices along the longest path. To illustrate the Pattern Path Algorithm's workflow, we provide an example in the following section.

Input:

- Cluster \mathbf{C} of t noisy traces: $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_t$.
- Design length = n
- $\mathbf{y}_k \in \mathbf{C}$ - a trace from cluster \mathbf{C} .

Output:

- $\hat{\mathbf{y}}_k$ - a revised version of \mathbf{y}_k . The sequence $\hat{\mathbf{y}}_k$ consists of \mathbf{y}_k 's original symbols and also includes some additional symbols, which are estimations of the symbols deleted from \mathbf{y}_k .
1. $L[\mathbf{y}_k] = \{L[\mathbf{y}_k]_1, \dots, L[\mathbf{y}_k]_{|\mathbf{y}_k|}\}$, a list of $|\mathbf{y}_k|$ empty lists, where each represents the list of patterns before the symbol i in \mathbf{y}_k , where the last list represents symbols before the end of the sequence.
 2. /*In this stage we pair \mathbf{y}_k with all the copies from the cluster, create list $L[\mathbf{y}_k]$ of $|\mathbf{y}_k|$ lists of patterns of symbol i and their frequencies*/
 - for $\mathbf{y}_h \in \mathbf{C}$ do
 - (a) Compute \mathbf{w} an LCS sequence of $\mathbf{y}_k, \mathbf{y}_h$.
 - (b) Compute $\mathbf{u}_{\mathbf{y}_k, \mathbf{w}}$ an embedding sequence for \mathbf{y}_k and \mathbf{w} .
 - (c) Compute $P(\mathbf{y}_k, \mathbf{y}_h, \mathbf{w}, \mathbf{u}_{\mathbf{y}_k, \mathbf{w}}, m = 3)$.
 - (d) For each $1 \leq i \leq \mathbf{y}_k$ add to $L[\mathbf{y}_k]_i$ the pattern $Ptn(\mathbf{y}_k, \mathbf{y}_h, \mathbf{w}, \mathbf{u}_{\mathbf{y}_k, \mathbf{w}}, i, 3)$.
 - end for
 3. Build $G_{pat} = (V, E)$ - the pattern graph.
 4. Find the longest path from the source vertex S in G_{pat} .
 5. Let $\hat{\mathbf{y}}_k$ bet the sequence that inherited from the patterns of the vertices of the longest path. That is, $\hat{\mathbf{y}}_k$ is the sequence that is created by concatenating the patterns of the longest path, one after the other, based on their positions in the sequence. It should be noted, that in the patterns deletion and insertion errors are marked by X and lower case symbols, respectively. Thus, the sequence $\hat{\mathbf{y}}_k$ can be used to correct these errors.
 6. Return $\hat{\mathbf{y}}_k$.
-

Algorithm 1. The Pattern Path Algorithm (PP Algorithm)

Example 1 Example of the PP Algorithm

We present here a short example of the Pattern Path Algorithm and its related definitions.

Step 1 - Input of the algorithm. The original strand in this example is \mathbf{x} which is given below. The cluster of its $t = 5$ traces is $\mathbf{C} = \{\mathbf{y}_1, \dots, \mathbf{y}_5\}$. The original design length of \mathbf{x} is $n = 10$. The traces are noisy copies of \mathbf{x} and include deletions, insertions, and substitutions. In this example Algorithm 1 receives the cluster \mathbf{C} and the trace $\mathbf{y}_k = \mathbf{y}_1$ as its input.

- $\mathbf{x} = \text{GTAGTGCCTG}$.
- $\mathbf{y}_1 = \text{GTAGGTGCCG}$.
- $\mathbf{y}_2 = \text{GTAGTCCTG}$.
- $\mathbf{y}_3 = \text{GTAGTGCCTG}$.
- $\mathbf{y}_4 = \text{GTAGCGCCAG}$.
- $\mathbf{y}_5 = \text{GCATGCTCTG}$.

Step 2 - Computation of the LCSs and the patterns in cluster. After receiving the input the PP algorithm continues with the next step of computing the LCSs and the patterns of the cluster. Figure 1 presents the process of computing the patterns of $(\mathbf{y}_1, \mathbf{y}_2), (\mathbf{y}_1, \mathbf{y}_3), (\mathbf{y}_1, \mathbf{y}_4), (\mathbf{y}_1, \mathbf{y}_5)$. For each pair, \mathbf{y}_1 and \mathbf{y}_i , Fig. 1 depicts \mathbf{w}_i , which is an LCS of the sequences \mathbf{y}_1 and \mathbf{y}_i . Then, the figure presents $\mathbf{u}_{\mathbf{y}_1, \mathbf{w}_i}$ and $\mathbf{u}_{\mathbf{y}_i, \mathbf{w}_i}$, which are the embedding sequences that the Pattern Path Algorithm uses in order to compute the patterns. Lastly, the list of patterns of

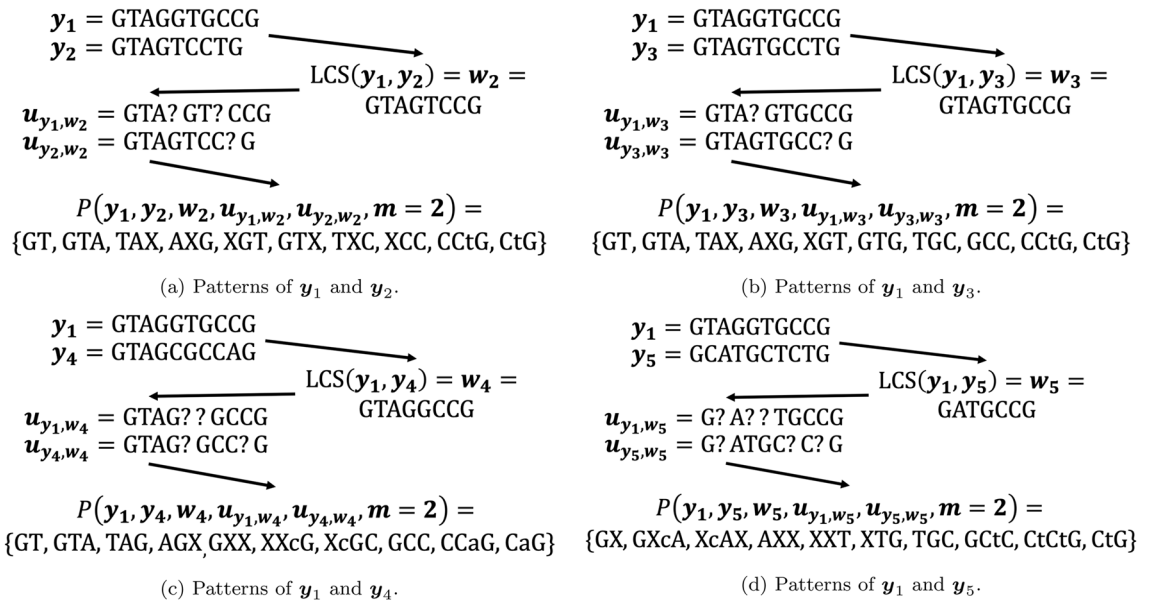


Figure 1. Algorithm 1 Example - Patterns of y_1 .

each pair is depicted in an increasing order of their indices. Note that lowercase symbols, present gaps and X presents the symbol ?.

Step 3 - Evaluating the patterns and their accuracy. The following list summarizes the patterns and their frequencies. Each list includes patterns from specific index. The numbers on the right side of each pattern in a list represents the pattern's frequency.

- $L[y_1]_1 = \{GT : 3, GX : 1\}$.
- $L[y_1]_2 = \{GTA : 3, GXcA : 1\}$.
- $L[y_1]_3 = \{TAX : 2, TAG : 1, XcAX : 1\}$.
- $L[y_1]_4 = \{AXG : 2, AGX : 1, AXX : 1\}$.
- $L[y_1]_5 = \{XGT : 2, GXX : 1, XXT : 1\}$.
- $L[y_1]_6 = \{GTG : 1, GTX : 1, XXcG : 1, XTG : 1\}$.
- $L[y_1]_7 = \{TGC : 2, TXC : 1, XcGC : 1\}$.
- $L[y_1]_8 = \{GCC : 2, XCC : 1, GCtG : 1\}$.
- $L[y_1]_9 = \{CCtG : 2, CCaG : 1, CtCtG : 1\}$.
- $L[y_1]_{10} = \{CtG : 3, CaG : 1\}$.

Step 4 - Creating the pattern path graph. Next, based on the list of patterns above, the pattern path is created. it can be shown that every pattern is a vertex in the graph, and that is represented by its sequence and its position in the sequence ($1 \leq i \leq 10$). Furthermore, the weights of the edges in the graph represents the frequencies of the patterns. As can be seen in Fig. 2, the created graph is a directed acyclic graph.

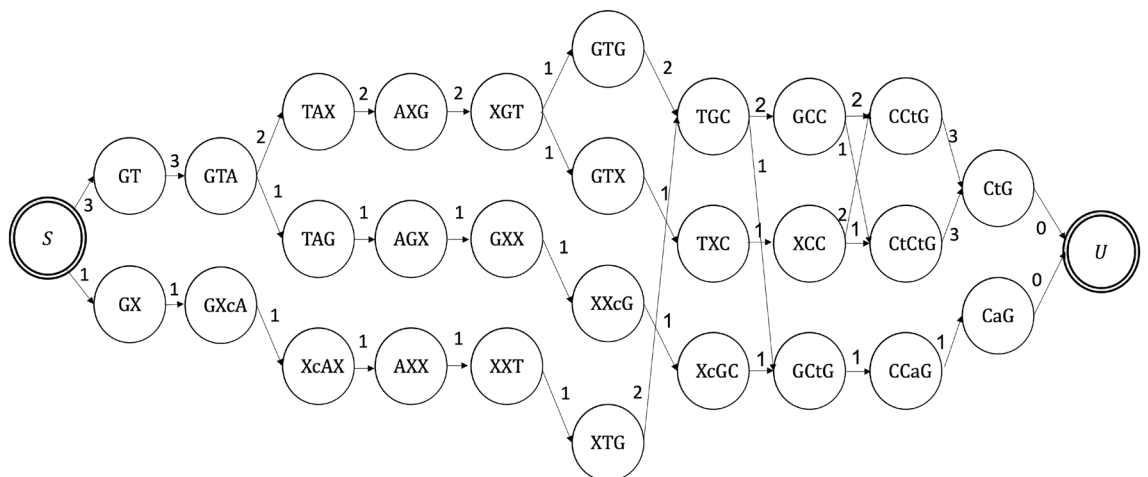


Figure 2. The pattern-path graph.

Step 5 - Output.

It is not hard to observe that the longest path in the pattern path graph of this example is:

$$S \rightarrow GT \rightarrow GTA \rightarrow TAX \rightarrow AXG \rightarrow XGT \rightarrow GTG \rightarrow TGC \rightarrow GCC \rightarrow CcTG \rightarrow CtG \rightarrow U,$$

and the algorithm output will be $\hat{y}_1 = GTAGTGCCTG = x$.

The divider BMA algorithm

In this section we shortly describe the *Divider BMA algorithm*, our own variation to the BMA-Lookahead algorithm presented in³³, that can improve results on large clusters. This algorithm is a stand-alone algorithm, designed for cases in which the clusters are relatively large and have lower error rates with shorter run time on such clusters (compared to the ITR algorithm and the PP algorithm). The Divider BMA algorithm receives a cluster and the design length n . The Divider BMA algorithm divides the traces of the cluster into three sub-clusters by their lengths, traces of length n , traces of length smaller than n , and traces of length larger than n . It performs a majority vote on the traces of length n . Then, similarly to the technique presented in the BMA algorithm¹² and in³³, the Divider BMA algorithm performs a majority vote on the sub-cluster of traces of length smaller than n , while detecting and correcting deletion errors. Lastly, the Divider BMA algorithm uses the same technique on the traces of length larger than n to detect and correct insertion errors. The time complexity of this algorithm is linear.

Results

In this section we present an evaluation of the accuracy of our algorithms on our new data set and on data from previous DNA storage experiments^{18–20,28,39}. The presented algorithms include the ITR algorithm, which uses the Pattern Path Algorithm (Algorithm 1) along with other algorithms that are described in the supplementary material. We also implemented the linear-time algorithm from³³, in the implementation we used several different value of the window size parameter of the algorithm, denoted by w . That is, our results include the algorithm from³³ with $2 \leq w \leq 4$. Additionally, we also implemented the algorithm from³², with parameters of $\ell = 5$, $\delta = (1 + p_s)/2$, $r = 2$ and $\gamma = 3/4$, while for the data of the DNA storage experiments the substitution probability p_s was taken from²⁹. For further information about the parameters we used in the algorithms from^{32,33}, the reader is referred to their original papers. Finally, we also included an additional algorithm which is our variation of the BMA algorithm^{12,33} to support also insertion and substitution errors, which is referred by the Divider BMA Algorithm.

In our new dataset, we designed 1000 random strands, each of length 128 symbols, and we matched each one of the strands with an index of length 12. The synthesis of this dataset was done by Twist Bioscience. The synthesized strands were amplified using 8 PCR cycles with the enzyme NebNext Q5 high-fidelity DNA polymerase. The sequencing was done using Oxford MinION, with the LSK109 ligation kit. The clustering of the data was done with the SOLQC tool²⁹ by binning the reads by their indices. The error characterization was also done using the SOLQC tool²⁹. The characterization process was done by calculating the edit distance of the traces and their original design sequence (which is given to SOLQC as input for error characterization). For further details about the clustering process and the error characterization process please see²⁹. The inspected error rate of the data is 3.5%. Additionally, we also evaluated our algorithm on a new data set that is based on the design from³⁹. We re-ordered the same design sequences as in³⁹ from Twist Bioscience. Then we amplified the synthesized DNA molecules with 8 PCR cycles using the Q5 enzyme and performed the sequencing using Oxford MinION, with the LSK110 ligation kit. According to our analysis the error rate in this data set is 3.41%.

Figure 3 presents the results of the tested algorithms on data from previous DNA storage experiments^{18–20,28,39} as well as on our new designed dataset. The clustering of these data sets was made by the SOLQC tool²⁹. We performed each of the tested algorithms on the data and evaluated the edit error rates. Note that in order to reduce the runtime of the ITR algorithm we filtered clusters of size $t > 25$ to have only the first 25 traces. The ITR algorithm presented the lowest edit error rates in almost all of the tested data sets.

We also evaluated the accuracy of our algorithms by simulations. First, we present our interpretation of the insertion-deletion-substitution channel, in which the sequence is transmitted symbol-by-symbol. First, before transmitting the symbol, it checks for an insertion error before the transmitted symbol. The channel flips a coin, and with probability p_i , an insertion error occurs before the transmitted symbol. If an insertion error occurs, the inserted symbol is chosen uniformly. Then, the channel checks for a deletion error, and again flips a coin, and with probability p_d the transmitted symbol is deleted. Lastly, the channel checks for a substitution error. The channel flips a coin, and with probability p_s the transmitted symbol is substituted to another symbol. The substituted symbol is chosen uniformly. In case that both deletion and substitution errors occurs in the same symbol, we refer to it as a substitution.

We simulated 100,000 clusters of size $t = 10, 20$, the sequences length was $n = 100$, and the alphabet size was $q = 4$. The deletion, insertion, and substitution probabilities were all identical, and ranged between 0.01 and 0.1. It means that the actual error probability of each symbol was $1 - (1 - p_i)(1 - p_s)(1 - p_d)$ and ranged between 0.029701 and 0.271. We reconstructed the original sequences of the clusters using the ITR Algorithm and the algorithms from³³ and from³². For each algorithm we evaluated its edit error rate and the success rate which is the fraction of clusters that were reconstructed successfully with no error. The edit error rate of the ITR Algorithm was the lowest among the tested algorithms, while the algorithm from³² presented the highest edit error rates. Moreover, it can be seen that the ITR Algorithm presented significantly low edit error rates for higher values of error probabilities. Such high probabilities can reflect the error probabilities of light-directed array-based synthesis as reported in²⁸. The results on the simulated data for $t = 20$ are depicted in Fig. 4.

Additionally, we performed another simulation to study the accuracy of our algorithms on the expected near-future DNA synthesis technologies. To do so, we simulated 10, 000 clusters of sequences of length $n = 300$. For

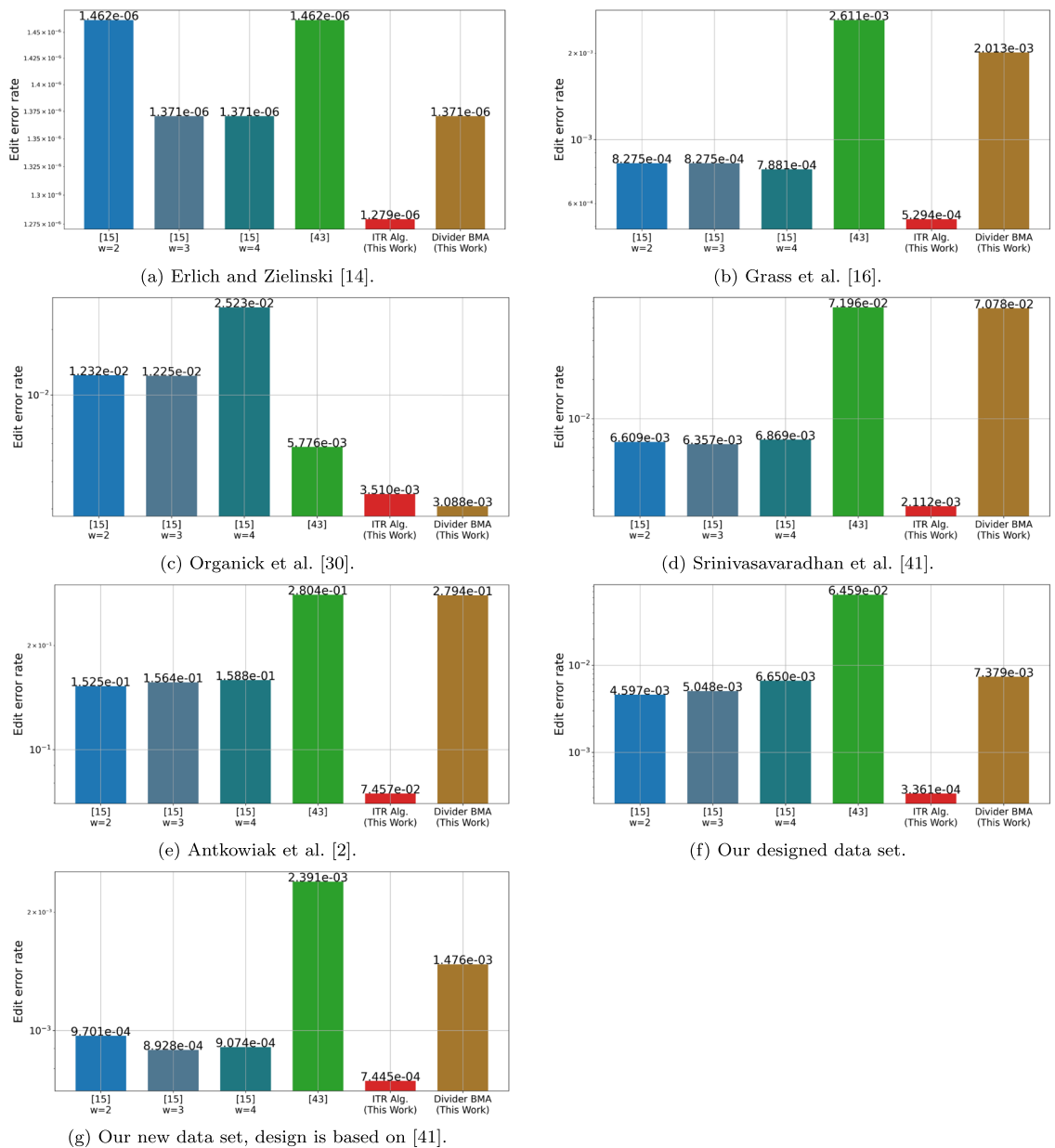


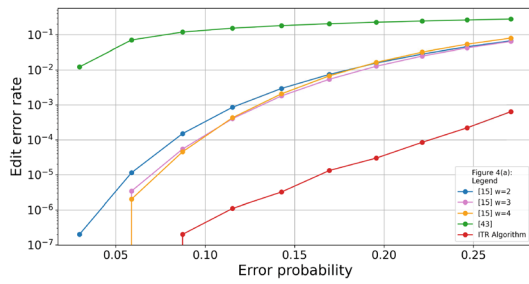
Figure 3. Edit error rate by the reconstruction algorithm, the results are presented on our new designed data sets and on data from DNA storage experiments^{18–20,28,39}.

each cluster, we simulated $t = 20$ noisy traces as described in the previous paragraph. Finally, we evaluate the accuracy of our algorithm on these data sets. In all cases the ITR algorithm shows significantly lower error rates compare to any of the other tested algorithms. The results are summarized in Fig. 6.

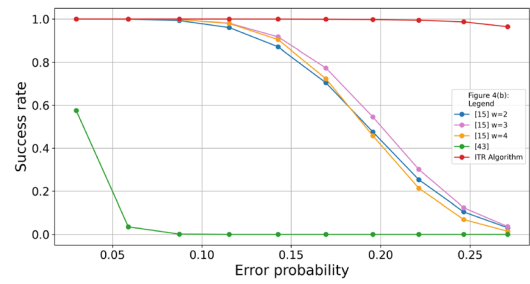
Lastly, we compared our algorithms with the Trellis BMA algorithm that was published in³⁹. This algorithm was designed to support data sets with higher error rate and hence we evaluated it on the data sets that presented relatively higher rates. These data sets include our new designed data sets, and the data set from^{28,39}. Similarly to what the authors suggested in³⁹, to improve the accuracy and the running time of the algorithm and to ensure fair comparison of this algorithm, we evaluated it on clusters of size 10 or less. Thus, we randomly sampled 10 traces from each of the clusters in these data sets and then invoked the algorithms on them. For clusters of size 10 or less we simply ran the algorithms on the entire cluster. The error rates of the algorithms are described in Fig. 5, where it can be seen that the ITR algorithm shows less error rates compared to the Trellis BMA algorithm³⁹ and the BMA Lookahead algorithm³³.

Performance evaluation

We evaluate the performance of the different algorithms discussed in this paper. The performance evaluation was performed on our server with Intel(R) Xeon(R) CPU E5-2630 v3 2.40 GHz. To test their performances we implemented our algorithms as well as the previously published algorithm from³³ and from³⁹, which presented

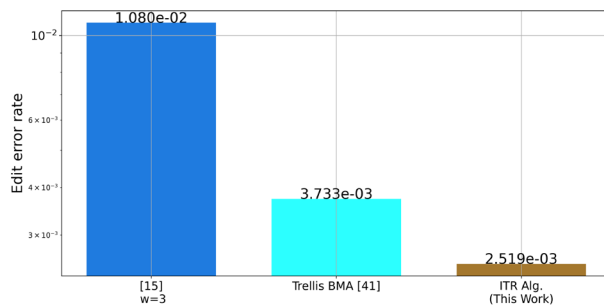


(a) Edit error rate by the error probability. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the edit error rate, which is the normalized edit distance between the estimated and the original sequences.

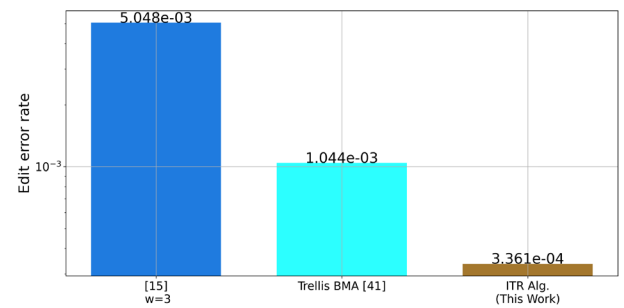


(b) Success rate by the error probability. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the success rate, which is the fraction of clusters that were fully reconstructed.

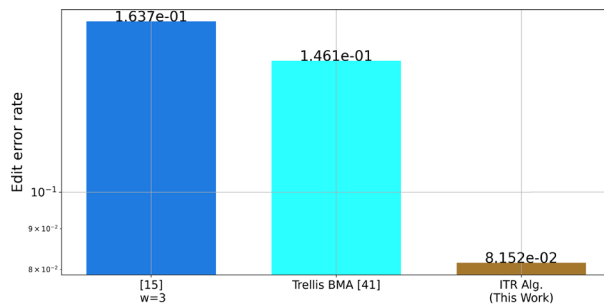
Figure 4. Edit error rate and success rate by the error probabilities for cluster size of $t = 20$. The length of the original sequence was $n = 100$ and the error probabilities ranges between 0.029701 and 0.271.



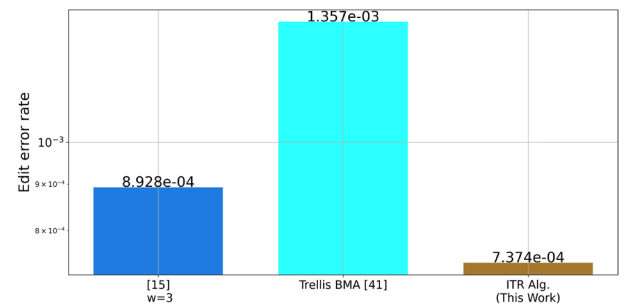
(a) Srinivasavaradhan et al. [41].



(b) Our designed data set.



(c) Antkowiak et al [2].

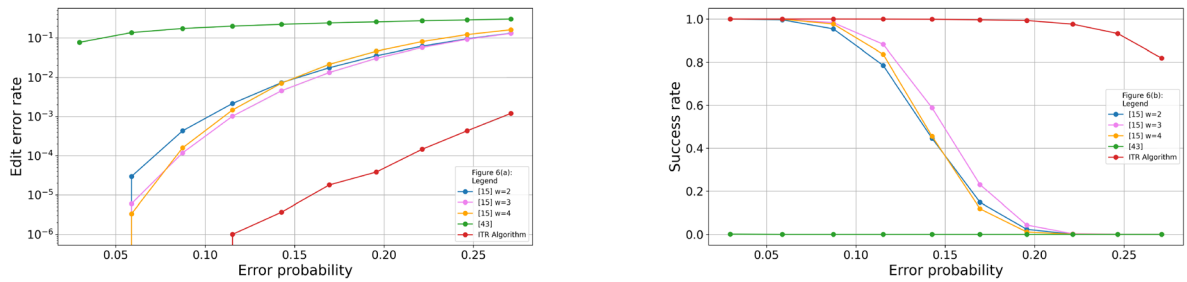


(d) Our data set, design is based on [41].

Figure 5. Edit error rate by the reconstruction algorithm on small clusters, the results are presented on our new designed data set and on data from DNA storage experiments^{28,39}.

the second-lowest error rates in our results from “Results” section. In order to present reliable performance evaluation, the clusters in our experiments were reconstructed in serial order. However, it is important to note, that for practical uses, additional performance improvements can be made by performing the algorithms on the different clusters in parallel and shortening the running time.

- Experiment A evaluates the running time of the algorithms on simulated data of 450 clusters of sequence length of $n = 110$, $p_d = p_s = p_i = 0.05$, so the total error rate was 0.142625. The cluster sizes were distributed normally between $t = 1$ and $t = 30$. The algorithm from³³ (with parameter $w = 3$) reconstructed the full data set in 0.6 second with total error rate of 0.02111, and success rate of 0.531. The ITR algorithm reconstructed the 450 clusters in 683 seconds and presented significantly less errors with total edit error rate of 0.000566, and success rate of 0.982262.
- Experiment B studies the running time of the reconstruction of the data set from³⁹. This data set includes 9984 clusters. Algorithm achieved a success rate of 0.875714, error rate of 0.00211 with total run time of 11, 486 seconds, and reported error rate of roughly 7%. The algorithm from³³ (with parameter $w = 3$) was able to reconstruct this data set in 12 seconds, while achieving three times larger error rate of 0.006 and success rate



(a) Edit error rate by the error probability. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the edit error rate, which is the normalized edit distance between the estimated and the original sequences.

(b) Success rate by the error probability. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the success rate, which is the fraction of clusters that were fully reconstructed.

Figure 6. Edit error rate and success rate by the error probabilities for cluster size of $t = 20$. The length of the original sequence was $n = 300$ and the error probabilities ranges between 0.029701 and 0.271.

- of 0.779948. It should be further noted that the runtime of the Trellis BMA algorithm, that was presented in³⁹ is significantly larger when running serially, and takes more than 10 days.
- Experiment C consists of the first 100,000 clusters from the data set of²⁰, and measures the running time of their reconstruction. The ITR algorithm was able to achieve a success rate of 0.99788 and an error rate of 0.0000282 in 93, 927 seconds, while the algorithm from³³ (with parameter $w = 3$) reconstructed the clusters in 79 seconds, while presenting success rate of 0.9979 and total edit error rate of 0.000047, which is approximately 1.6 times more errors compared to the ITR algorithm. Since for this data set, the divider BMA presented lower error rate compare to³³, we decided to evaluate its performance on this data set. The divider BMA algorithm reconstructed the data set in 64 seconds and error rate of 0.000044.

It can be seen that, the run times of the ITR algorithm is remarkably longer. However, they can significantly improve both the success rate and the error rate of the algorithm from³³. Their advantage is even larger for small clusters as in experiment A, and when the error rates are relatively high as in experiments A and in³⁹. Hence, it is also possible to consider hybrid algorithm that invokes the ITR algorithm or the the algorithm³³ based on the given cluster sizes. Since in data from previous DNA storage experiments the variance in the cluster size and in the error rates can be really high²⁹, we added an additional condition to the hybrid algorithm, so it invokes the algorithm from³³ if the cluster is of size 20 or larger, or if the absolute distance of the difference between the average length of the traces in the cluster and the design length is larger than 10% of the design length. The hybrid algorithm reconstructed the first 100,000 clusters of²⁰ (experiment C) in 58 seconds and presented error rate of 0.000333 are the largest success rate among all tested algorithm of 0.998. The results of the performance experiments are also depicted in Table 1.

We also study the running time of our algorithms on simulated data designed to resemble the near-future synthesis technologies in Experiment D. In this test, we assess 10,000 clusters, generated from random DNA sequences of length $n = 300$. In this scenario, the simulated error probabilities are set to be $p_d = p_s = p_i = 0.05$. Since we also wanted to compare our results on these clusters with the results of the Trellis BMA algorithm³⁹, we had to restrict the cluster size to be $t = 10$ (to ensure acceptable running time), meaning each cluster in this simulation includes 10 traces. Given this cluster size, the hybrid algorithm cannot significantly enhance the

	ITR algorithm	Gopalan et al. ³³	Hybrid algorithm
Experiment A - time (sec.)	683	0.6	155
Experiment A - error rate	0.000566	0.02111	0.0121414
Experiment A - success rate	0.982	0.531	0.695
Experiment B ³⁹ - time (sec.)	11,486	12	401
Experiment B ³⁹ - error rate	0.00211	0.006	0.005
Experiment B ³⁹ - success rate	0.875714	0.779948	0.8002
Experiment C ²⁰ - time (sec.)	93,927	78	58
Experiment C ²⁰ - error rate	0.0000282	0.000047	0.000033
Experiment C ²⁰ - success rate	0.9979	0.9979	0.998

Table 1. Performance evaluation of the ITR Algorithm, Gopalan et al. algorithm³³, and the Hybrid Algorithm. The number presented the running time in seconds and the error rates of each of the algorithms for each of the experiments.

	ITR Algorithm	Gopalan et al. ³³	Trellis BMA ³⁹
Experiment D - time (sec.)	80,301	59.7	864,000
Experiment D - error rate	0.00057	0.0324	0.0044
Experiment D - success rate	0.8955	0.037	0.5496

Table 2. Performance evaluation of the ITR Algorithm, Gopalan et al. algorithm³³, and the Trellis BMA Algorithm³⁹. The number presented the running time in seconds and the error rates of each of the algorithms for each of the experiments.

running time of the ITR algorithm. Therefore, in this evaluation, we compare the ITR algorithm, the Trellis BMA algorithm³⁹, and the BMA Lookahead algorithm³³. We observe that the ITR shows the higher success rate and the lower error rate compared to the tested algorithms. Additionally, the second best algorithm in terms of accuracy is the Trellis BMA³⁹ that shows running time which is more than ten times higher compared to the ITR algorithm. It should be noted that during the listed running time of 864,000 seconds, the Trellis BMA³⁹ was not able to reconstruct the entire set of 10,000 clusters, but only a fraction of the first 1,000 clusters. Thus, the actual running time of this algorithm is much higher than listed in the table. The success rate of the BMA Lookahead algorithm³³ is the lowest among the tested algorithm, and thus we believe it might not be applicable to the near-future technologies. The results of this experiment are summarized in Table 2.

Conclusions and discussion

We presented in this paper several new algorithms for the DNA reconstruction problem. While most of the previously published algorithms either use a symbol-wise majority approaches or have a prior-knowledge of the error rates, our algorithms look globally on the entire sequence of the traces, and use the LCS or SCS of a given set of traces. Our algorithms are designed to specifically support DNA storage systems and to reduce the edit error rate of the reconstructed sequences. According to our tests on simulated data and on data from DNA storage experiments, we found out that our algorithms significantly improved the accuracy compared to the previously published algorithms. While the ITR algorithm's running time is longer than that of previously published algorithms^{12,33}, it still stands as a significantly more efficient choice than the Trellis BMA algorithm³⁹. Furthermore, the ITR algorithm demonstrates substantial accuracy improvements, especially in datasets featuring high error rates, smaller clusters, and longer sequences. Nevertheless, for the broader application of DNA storage systems, future research should focus on reducing latency in synthesis, sequencing, clustering, and reconstruction processes. Additionally, this future research should also consider optimizing the ITR algorithm's running time through parallelization of its steps.

Data availability

The datasets generated during the current study are available in the google drive repository, as in the following https://drive.google.com/drive/folders/1c3kopMcUsW_tYnjfgjPLuMaLv-cDBV8O?usp=sharing.

Code availability

Implementation of the algorithms and instructions on how to use them can be found in the GitHub repository in the following <https://github.com/omersabary/Reconstruction>. For commercial use please contact the authors.

Received: 24 June 2023; Accepted: 9 January 2024

Published online: 23 January 2024

References

- Barrett, M. T. *et al.* Comparative genomic hybridization using oligonucleotide microarrays and total genomic DNA. *Proc. Natl. Acad. Sci.* **101**(51), 17765–17770 (2004).
- Chen, Z. *et al.* Highly accurate fluorogenic DNA sequencing with information theory-based error correction. *Nat. Biotechnol.* **35**(12), 1170 (2017).
- Kosuri, S. & Church, G. M. Large-scale de novo DNA synthesis: Technologies and applications. *Nat. Methods* **11**(5), 499 (2014).
- Lee, H. H., Kalthor, R., Goela, N., Bolot, J. & Church, G. M. Terminator-free template-independent enzymatic DNA synthesis for digital information storage. *Nat. Commun.* **10**(1), 1–12 (2019).
- LeProust, E. M. *et al.* Synthesis of high-quality libraries of long (150mer) oligonucleotides by a novel depurination controlled process. *Nucleic Acids Res.* **38**(8), 2522–2540 (2010).
- Palluk, S. *et al.* De novo DNA synthesis using polymerase-nucleotide conjugates. *Nat. Biotechnol.* **36**(7), 645 (2018).
- Snir, S., Yeager-Lotem, E., Chor, B. & Yakhini, Z. Using restriction enzymes to improve sequencing by hybridization. Technical report, Computer Science Department, Technion (2002).
- Beaucage, S. L. & Iyer, R. P. Advances in the synthesis of oligonucleotides by the phosphoramidite approach. *Tetrahedron* **48**(12), 2223–2311 (1992).
- Heckel, R., Mikutis, G. & Grass, R. N. A characterization of the DNA data storage channel. *Sci. Rep.* **9**, 9663 (2019).
- Levenshtein, V. I. Efficient reconstruction of sequences. *IEEE Trans. Inf. Theory* **47**(1), 2–22 (2001).
- Levenshtein, V. I. Efficient reconstruction of sequences from their subsequences or supersequences. *J. Comb. Theory Ser. A* **93**(2), 310–332 (2001).
- Batu, T., Kannan, S., Khanna, S. & McGregor, A. Reconstructing strings from random traces. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 910–918 (Society for Industrial and Applied Mathematics, 2004).
- De, A., O'Donnell, R., & Servedio, R. A. Optimal mean-based algorithms for trace reconstruction. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 1047–1056 (ACM, 2017).

14. Holden, N., Pemantle, R., Peres, Y & Zhai A. Subpolynomial trace reconstruction for random strings and arbitrary deletion probability. *Mathemat. Statist. Learn.* **2**(3), 275–309 (2020).
15. Holenstein, T., Mitzenmacher, M., Panigrahy, R., & Wieder, U. Trace reconstruction with constant deletion probability and related results. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 389–398 (Society for Industrial and Applied Mathematics, 2008).
16. Peres, Y. & Zhai, A. Average-case reconstruction for the deletion channel: Subpolynomially many traces suffice. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, 228–239 (2017).
17. Shinkar, T., Yaakobi, E., Lenz, A. & Wachter-Zeh, A. Clustering-correcting codes. In *IEEE International Symposium on Information Theory (ISIT)*, 81–85 (2019).
18. Erlich, Y. & Zielinski, D. DNA fountain enables a robust and efficient storage architecture. *Science* **355**(6328), 950–954 (2017).
19. Grass, R. N., Heckel, R., Puddu, M., Paunescu, D. & Stark, W. J. Robust chemical preservation of digital information on DNA in silica with error-correcting codes. *Angew. Chem. Int. Ed.* **54**(8), 2552–2555 (2015).
20. Organick, L. *et al.* Random access in large-scale DNA data storage. *Nat. Biotechnol.* **36**, 242 EP- (2018).
21. Anavy, L., Vaknin, I., Atar, O., Amit, R. & Yakhini, Z. Data storage in DNA with fewer synthesis cycles using composite. *DNA Lett. Nat. Biotechnol.* **37**(10), 1229–1236 (2019).
22. Takahashi, C. N., Nguyen, B. H., Strauss, K. & Ceze, L. Demonstration of end-to-end automation of DNA data storage. *Sci. Rep.* **9**(1), 1–5 (2019).
23. Yazdi, S. H. T., Gabrys, R. & Milenkovic, O. Portable and error-free DNA-based data storage. *Sci. Rep.* **7**(1), 5011 (2017).
24. Pan, W. *et al.* DNA polymerase preference determines PCR priming efficiency. *BMC Biotechnol.* **14**(1), 10 (2014).
25. Ruijter, J. *et al.* Amplification efficiency: Linking baseline and bias in the analysis of quantitative PCR data. *Nucleic Acids Res.* **37**(6), 45 (2009).
26. Chandak, S., *et al.* Improved read/write cost tradeoff in DNA-based data storage using LDPC codes. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 147–156 (2019).
27. Lopez, R. *et al.* DNA assembly for nanopore data storage readout. *Nat. Commun.* **10**(1), 1–9 (2019).
28. Antkowiak, P. L. *et al.* Low cost DNA data storage using photolithographic synthesis and advanced information reconstruction and error correction. *Nat. Commun.* **11**, 5345 (2020).
29. Sabary, O., *et al.* SOLQC: Synthetic oligo library quality control tool. *Bioinformatics* **37**(5), 720–722 (2019).
30. Lietard, J. *et al.* Chemical and photochemical error rates in light-directed synthesis of complex DNA libraries. *Nucleic Acids Res.* **49**(12), 6687–6701 (2021).
31. Nazarov, F. & Peres, Y. Trace reconstruction with exp (o (n 1/3)) samples. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 1042–1046 (2017).
32. Viswanathan, K., & Swaminathan, R. Improved string reconstruction over insertion-deletion channels. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 399–408 (2008).
33. Gopalan, P. S., *et al.* Trace reconstruction from noisy polynucleotide sequencer reads, (July 26 2018). US Patent App. 15/536,115.
34. Duda, J., Szpankowski, W. & Grama, A. Fundamental bounds and approaches to sequence reconstruction from nanopore sequencers. *arXiv preprint arXiv:1601.02420* (2016).
35. Edgar, R. C. Muscle: Multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res.* **32**(5), 1792–1797 (2004).
36. MATLAB. Multialign function. <https://www.mathworks.com/help/bioinfo/ref/multialign.html> (2016).
37. Song, L. *et al.* Robust data storage in DNA by de Bruijn graph-based de novo strand assembly. *Nat. Commun.* **13**, 5361 (2022).
38. Bar-Lev, D., Orr, I., Sabary, O., Etzion, T. & Yaakobi, E. Deep DNA storage: scalable and robust DNA storage via coding theory and deep learning. *arXiv preprint arXiv:2109.00031* (2021).
39. Srinivasavaradhan, S. R., Gopi, S., Pfister, H. D. & Yekhanin, S. Trellis BMA: Coded trace reconstruction on IDS channels for DNA storage. In *IEEE International Symposium on Information Theory (ISIT)*, 2453–2458 (2021).
40. Sabary, O., Yaakobi, E. & Yucovich, A. The error probability of maximum-likelihood decoding over two deletion channels. In *IEEE International Symposium on Information Theory (ISIT)*, 763–768 (2020).
41. Srinivasavaradhan, S. R., Du, M., Diggavi, S. & Fragouli, C. On maximum likelihood reconstruction over multiple deletion channels. In *IEEE International Symposium on Information Theory (ISIT)*, 436–440 (2018).
42. Atashpendar, A., Beunardeau, M., Connolly, A., Gérard, R., Mestel, D., Roscoe, A. W. & Ryan, P. Y. A. From clustering supersequences to entropy minimizing subsequences for single and double deletions. *arXiv preprint arXiv:1802.00703* (2019).
43. Elzinga, C., Rahmann, S. & Wang, H. Algorithms for subsequence combinatorics. *Theoret. Comput. Sci.* **409**(3), 394–404 (2008).

Acknowledgements

The research was funded in part by the European Union (ERC, DNASStorage, 865630). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was also partially supported by the Israel Innovation Authority grant 75855. The authors of this paper thank Prof. Roei Amit, Dr. Sarah Goldberg and Dr. Nanami Kikuchi for their help with the PCR process of the published libraries, and for the fruitful discussion and lab equipment. The authors also wish to thank the Technion Genome Center, specifically Dr. Liat Linde and Dr. Nitsan Fourier, for their help with the sequencing process of the libraries. The authors thank Matika Lidgi for her help with the divider BMA algorithm and Rotem Samuel for his help with the implementation and simulations of the algorithms in the paper. They also thank Dr. Cyrus Rashtchian for helpful discussions and Prof. Gala Yadgar for her invaluable help with the simulation infrastructure and they also thank Daniella Bar-Lev for helpful and inspiring discussions along the way. Finally, the authors thank the reviewers and the editor of Scientific Reports for their valuable comments and suggestions that improved this paper significantly.

Author contributions

O.S., A.Y., and E.Y. initiated the study and designed the algorithmic approach. O.S. and A.Y. developed the software. O.S. performed an analysis of data from previous experiments. G.S. developed algorithms for the deletion channel with large clusters. O.S. performed the sequencing and data analysis of the data. O.S. and E.Y. wrote the manuscript. E.Y. supervised the study.

Additional information

Supplementary Information The online version contains supplementary material available at <https://doi.org/10.1038/s41598-024-51730-3>.

Correspondence and requests for materials should be addressed to O.S.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

© The Author(s) 2024