Adaptive Learning of Compressible Strings

Gabriele Fici[a], Nicola Prezza[b], Rossano Venturini[c]

[a]*Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Palermo, Italy*
gabriele.fici@unipa.it
[b]*Dipartimento di Scienze Ambientali, Informatica e Statistica, Università Ca' Foscari, Venezia, Italy*
nicola.prezza@unive.it
[c]*Dipartimento di Informatica, Università di Pisa, Pisa, Italy*
rossano.venturini@unipi.it

Abstract

Suppose an oracle knows a string S that is unknown to us and that we want to determine. The oracle can answer queries of the form "Is s a substring of S?". In 1995, Skiena and Sundaram showed that, in the worst case, any algorithm needs to ask the oracle $\sigma n/4 - O(n)$ queries in order to be able to reconstruct the hidden string, where σ is the size of the alphabet of S and n its length, and gave an algorithm that spends $(\sigma-1)n + O(\sigma\sqrt{n})$ queries to reconstruct S.

The main contribution of our paper is to improve the above upper-bound in the context where the string is compressible. We first present a universal algorithm that, given a (computable) compressor that compresses the string to τ bits, performs $q = O(\tau)$ substring queries; this algorithm, however, runs in exponential time. For this reason, the second part of the paper focuses on more time-efficient algorithms whose number of queries is bounded by specific compressibility measures. We first show that any string of length n over an integer alphabet of size σ with rle runs can be reconstructed with $q = O(\mathsf{rle}(\sigma + \log \frac{n}{\mathsf{rle}}))$ substring queries in linear time and space. We then present an algorithm that spends $q \in O(\sigma g \log n)$ substring queries and runs in $O(n(\log n + \log \sigma) + q)$ time using linear space, where g is the size of a smallest straight-line program generating the string.

Keywords: String reconstruction, String learning, Adaptive learning, Kolmogorov complexity, String Compression, Lempel-Ziv, Centroid decomposition, Suffix tree.

1. Introduction

String reconstruction (or learning) from substrings queries is a well-established problem that has natural applications in many areas, including bioinformatics, data compression, security, etc. (see, for example, [1, 2, 3, 4]).

In a more general setting, one is interested in understanding whether and how it is possible to reconstruct an unknown target string S from some piece of information about S. This information can be, for example, a collection of substrings (e.g., the classical NP-hard Shortest

*Supported by MIUR PRIN 2017 Project 2017K7XPAN and Università di Pisa Project PRA_2020-2021_26.

Superstring Problem), or substring compositions ([5]), or subwords ([6]), of S. Furthermore, the problem can be viewed from different angles, e.g., combinatorial, computational, algorithmic, information theoretical.

In this paper, we deal with the problem of reconstructing a string from information about its substrings. Apart from the classical static model for the reconstruction (exact or with uncertainty), many different models have been introduced in the literature for the string reconstruction problem, including the one we consider in this paper, and which has been presented in 1995 by Skiena and Sundaram [3]. In this model, one can ask an oracle, which knows the target string S, queries of the form "Is s a substring of S?" and is interested in designing an *adaptive algorithm* minimizing the number of such queries. In this setting, *adaptive* means that the algorithm may reuse the information resulting from previous queries in order to decide which queries to ask next. It is worth mentioning that, with the same model, other query complexities have been investigated very recently by Amir et al. [7].

A trivial information-theoretic argument implies a worst-case lower bound of $n \log \sigma$ queries, where σ is the size of the alphabet of S. Skiena and Sundaram [3] improved this bound and showed that $\sigma n/4 - O(n)$ queries are necessary to reconstruct S in the worst case. This remains true even if the oracle returns for each substring query the number of its occurrences in S. In the same paper, they gave an algorithm for the reconstruction which spends at most $(\sigma - 1)n + O(\sigma\sqrt{n})$ queries, thus asymptotically matching the lower bound. They also gave an algorithm that spends at most $(\sigma-1)n+2\log n+O(\sigma)$ queries if the length n of S is known. Iwama et al. gave an algorithm for binary strings that spends $n + O(1)$ queries *on average* [8]. Amir et al. [7] recently proved that if the string has period $p > 0$, then it can be reconstructed using $O(\sigma p + \lg n)$ substring queries, even if both n and p are unknown.

We stress out that these bounds hold in the *adaptive* case: answers to previous queries can be used to decide the next query. As shown by Skiena et al. [2], the non-adaptive model is much harder: if the algorithm has to reconstruct all substrings (of the unknown string) of length $k \leq n$ after a pre-determined batch of queries, then $\sigma^{k/2}/k$ queries are needed to solve the problem. Tsur [4] explored this model more in detail, providing bounds as a function of the fraction $(1 - \epsilon)$ (with $0 \leq \epsilon \leq 1$) of substrings of length k that have to be reconstructed: in this case, $\Omega(\epsilon^{-1/2}k^2)$ non-adaptive queries are sufficient and necessary.

1.1. A novel approach to the problem

While the aforementioned papers tackled the problem in the worst-case, the minimum number of queries needed to reconstruct a string may be significantly smaller than the worst-case on particular instances. For example, consider a string of the form a^n, where a is a single letter. An algorithm could first try to find out if the string is of this form by issuing $O(\log n) + 2\sigma$ queries and, only if the check fails, proceed with Skiena and Sundaram's algorithm [3]. Observe that the resulting algorithm optimizes for a particular class of *highly-compressible* strings. In fact, in this paper we show that this reasoning continues to hold for *any compressor*. Our first result is a universal algorithm that, given as input a computable compressor $\mathcal{C}$, performs the reconstruction asking a number of queries that is proportional to the bit-size $|\mathcal{C}(S)|$ of the string compressed by $\mathcal{C}$. We complement this result by showing

that any deterministic adaptive algorithm for reconstructing a string yields a string compressor. Together, these results imply the equivalence between the string reconstruction and compression problems.

Motivated by the fact that our universal algorithm performs an exponential number of calls to $\mathcal{C}$, we then focus on optimizing the running time and the space usage for commonly used compressors, including run-length encoding, Lempel-Ziv factorization and context-free grammars.

In measuring the efficiency of an algorithm, we assume that any query can be submitted to the oracle in constant time and space regardless of the length of the queried substring. The reason for this assumption is that the implementation of the oracle strongly depends on the application. For example, if the application admits a collaborative oracle, there are several possible approaches to achieve constant query time, e.g., using hashing. Moreover, one could also assume that the oracle knows the reconstruction strategy and therefore it could run the reconstruction algorithm itself, that is, we do not even need to transmit the next substring query because the oracle already knows the next query it has to answer.

1.2. Preliminary definitions

Let S be a binary string. A *compressor* is an injective computable function $\mathcal{C} : \{0,1\}^+ \to \{0,1\}^+$ that converts any $S \in \{0,1\}^+$ into a reversible representation $\mathcal{C}(S)$ of size $|\mathcal{C}(S)|$ bits. We require also the inverse function $\mathcal{C}^{-1}$ (i.e. the function such that $\mathcal{C}^{-1}(\mathcal{C}(S)) = S$) to be computable; this function is the *decompressor* associated with $\mathcal{C}$. Informally speaking, a function $\mathcal{C}$ qualifies as a good compressor if $|\mathcal{C}(S)| \ll |S|$ on particular string families (for example, repetitive strings), and $|\mathcal{C}(S)| \in O(|S|)$ for all other strings outside this family.

A popular compressor is the LZ77 factorization of S. The Lempel-Ziv 1977 (LZ77) algorithm [9] parses a string S into a sequence of z phrases, where each new phrase is either a fresh character or the longest string that also occurs starting from a position strictly smaller than the phrase start position. The bit-size of the LZ77 factorization of S is $\Theta(z \log n)$. For example, the LZ77 factorization of the string *abbabba* is $a|b|b|abba$. In this example, the string is factored into $z = 4$ phrases. A more restricted version of LZ77 does not allow overlaps between a phrase and its source. We denote this version as *LZ77 without overlaps* and with z_{no} the number of generated phrases. Between those two measures, it holds $z_{no} \in O(z \log n)$ [10]. Clearly, also $z \leq z_{no}$ always holds. This version factorizes the above string as $a|b|b|abb|a$, with $z_{no} = 5$.

Another common measure of compressibility is the number rle of equal-letter runs in S, that is, the number of maximal unary substrings of S. This measure is not as strong as z; in fact, it is easy to see that $z_{no} \leq \mathsf{rle}$.

In this work we also consider the size (number of nonterminals) g of the smallest straight-line program (SLP) producing (only) S. SLPs are particular cases of acyclic context-free grammars composed of rules of the kind $A \to BC$, where A is a nonterminal and B, C are either nonterminals or terminals.

The known relations between g and z_{no} are $g \in O(z_{no} \log(n/z_{no}))$ and $z_{no} \leq g$. See Navarro's recent survey [10] for more details on these and several other relations between string complexity measures.

2. Universal string reconstruction

In this section we present an algorithm that, given a compressor C, reconstructs any string S with $O(|C(S)|)$ queries to the oracle. We furthermore prove a dual result: any reconstruction algorithm performing $\chi(S)$ queries yields a compression algorithm (with associated decompressor) that compresses the string to $\chi(S)$ bits. These findings show that the string reconstruction problem essentially coincides with the string compression problem. For simplicity, in this section we restrict our attention to binary alphabets only.

We start with a lemma of Skiena and Sundaram [3] stating that any set M of binary strings admits a string that is a substring of a constant fraction of the members of M. Letting M be a set of strings, we let $M(S)$ denote the subset of M whose elements contain S as a substring.

Lemma 1. *([3, Lem. 12]) Let $M \subseteq \{0,1\}^n$ be a set of binary strings, each of length n. Then, there exists a string S such that $\frac{1}{5}|M| \leq |M(S)| \leq \frac{4}{5}|M|$.*

Lemma 1 can be turned into a universal algorithm for determining the substring queries to be asked to the oracle as a function of any given compressor.

Lemma 2. *Let $\mathcal{C}$ be a compressor, and let $S \in \{0,1\}^n$ be an unknown binary string of known length n. Then, there is an algorithm that reconstructs S using $O(|\mathcal{C}(S)|)$ substring queries.*

Proof. Let $M_k = \{S \in \{0,1\}^n \ : \ |\mathcal{C}(S)| \leq k\}$ be the set of strings of length n compressed to at most k bits by $\mathcal{C}$. Note that $|M_k| \leq 2^{k+1} - 2$, since $\mathcal{C}$ is injective and there are no more than $2^{k+1} - 2$ binary strings (compressed representations) of length at most k. Assuming we know the value of $\tau = |\mathcal{C}(S)|$ (later we remove this assumption), it is easy to design an (exponential-time) algorithm that builds M_τ: simply apply $\mathcal{C}$ to all strings of length n, keeping only those such that $|\mathcal{C}(S)| \leq \tau$. By definition of τ, note that $S \in M_\tau$. Then, by applying recursively Lemma 1 starting from the set M_τ, we end up selecting S from this set. Each recursive iteration yields a string that we use to perform a substring query on S, thereby reducing the number of candidates by a factor 4/5 in the worst case. After $O(\log |M_\tau|) = O(\tau)$ iterations (i.e., substring queries on S), we discover which element of M_τ corresponds to S. To conclude, we can remove the assumption that we know τ. To achieve this goal it is sufficient to run an exponential search on the above strategy, i.e., run it on $M_{\tau'}$ for $\tau' = 1, 2, 4, \ldots, 2^{\lceil \log_2 \tau \rceil}$. The last iteration will reveal S, after a total of $O(\tau)$ substring queries on S. □

We finally prove the following lemma.

Lemma 3. *Let A be a deterministic adaptive algorithm that reconstructs any string S by asking $\chi(S)$ queries to the oracle, for some (computable) function $\chi(S)$. Then, there exists a compressor $\mathcal{C}$ (and an associated decompressor $\mathcal{C}^{-1}$) such that $|\mathcal{C}(S)| = \chi(S)$.*

Proof. It is straightforward to turn A into a compressor $\mathcal{C}$: the compressed representation $\mathcal{C}(S)$ of S is the binary string of length $\chi(S)$ formed by the $\chi(S)$ answers received by the oracle while reconstructing S. The $\chi(S)$ answers can be computed by any pattern matching

algorithm testing membership of the substrings queried by A in the substring closure of S. Similarly, A itself can be turned into a decompressor: by definition of A, the $\chi(S)$ answers of the oracle (i.e. the compressed file representation) are sufficient to reconstruct A. □

While the above results establish an asymptotic equivalence between the string reconstruction and compression problems, they do not yield time-efficient algorithms for reconstructing a string in time proportional to its compressed size. In the next section we tackle this problem by focusing on particular string compressors.

3. Feasible algorithms for the reconstruction

Let S be a string of length n over an integer alphabet $\Sigma = [1, \ldots, \sigma]$. A trivial algorithm for reconstructing S with $\sigma(n+1)$ substring queries is the following [3]: We make queries of single character substrings, so that after at most σ queries a new character of S is determined. Let s be a known substring of S. In general, we can increase the length of this known substring by one character by querying on the strings $s\sigma_i$, for every character σ_i. At least one of these queries must be a substring of S, unless s is a suffix of S that has no other occurrences in S. When s can no longer be extended to the right, i.e., s is a suffix of S not appearing elsewhere in S, we can continue the process by prepending characters to the known substring s, until it can no longer be extended to the left, and the string S is then reconstructed.

This algorithm is optimal up to constant factors due to the following lower bound [3].

Theorem 4. *([3, Thm. 8]) In the worst case, $\frac{\sigma n}{4} - O(n)$ substring queries are necessary to reconstruct a string of length n.*

In the rest of this section, we will provide algorithms for reconstructing the string S whose efficiency is measured towards commonly used measures of compression for strings.

Let us first show an easy result for the size rle of the run-length encoding of S, i.e., rle is the number of runs (maximal repetitions of the same character) in S. We show that S can be reconstructed with $O(\mathsf{rle}(\sigma + \log \frac{n}{\mathsf{rle}}))$ queries. The reconstruction is done in rle steps. Let $\hat{S}_{i-1}$ be the substring reconstructed so far. In the ith step, we first identify the character c that follows $\hat{S}_{i-1}$ in S. This is done by querying $\hat{S}_{i-1} \cdot c$ for any $c \in \Sigma$. Once we know c, we need to identify the length of the run of c, i.e., the maximal value r_i such that $\hat{S}_{i-1} \cdot c^{r_i}$ is a substring of S. This can be done with an exponential search on r_i, which takes $\Theta(\log r_i)$ queries. When at some step j, $\hat{S}_j$ cannot be extended further, we continue the process by prepending (runs of) characters to the known substring.

The overall number of queries is $q = O(\sum_{i=1}^{\mathsf{rle}}(\sigma + \log r_i))$. This is in $O(\mathsf{rle}(\sigma + \log \frac{n}{\mathsf{rle}}))$ queries because the sum of the terms $\log r_i$ is maximized when every r_i is in $\Theta(\frac{n}{\mathsf{rle}})$.

Theorem 5. *Any string of length n with rle runs can be reconstructed with $q = O(\mathsf{rle}(\sigma + \log \frac{n}{\mathsf{rle}}))$ substring queries in $O(q)$ time and $O(\mathsf{rle})$ space.*

Note that, accordingly to Theorem 4, this result is optimal up to a constant factor for sufficiently large σ.

Our next aim is to give algorithms whose complexity grows as a function of the size of the LZ77 parsing of S. We let z_{no} denote the number of phrases of the LZ77 parsing when the parse does not allow overlapping phrases (both settings are commonly considered in the literature).

We can use Theorem 4 to prove a lower bound on $\chi(S)$ in terms of z_{no}.

Theorem 6. *In the worst case, $\Omega(\sigma z_{no} \log_\sigma n)$ substring queries are necessary to reconstruct a string of length n.*

Proof. It is well known that for any string $z_{no} = O(\frac{n}{\log_\sigma n})$. The theorem follows by combining this fact with Theorem 4. □

We are not required to know the length n of S, but we assume to know its alphabet $\Sigma = [1, \ldots, \sigma]$. If instead also the alphabet is unknown, we need $O(\log \sigma)$ queries to identify the largest character in S. This is done by performing an exponential search to identify the largest character occurring in S. Notice that this is correct only if all the characters in Σ occur in S (in particular, $\sigma \leq n$), which we assume as hypothesis.

Our goal is to prove the following theorem.

Theorem 7. *Let S be a string of length n over the alphabet $\Sigma = [1, \ldots, \sigma]$. There exists an algorithm that reconstructs S with $q = O(\sigma z_{no} \log(n/z_{no}) \log n)$ substring queries to the oracle. The algorithm runs in $O(n(\log n + \log \sigma) + q)$ time using linear space.*

Note that this result is optimal up to a factor $O(\log \frac{n}{z_{no}} \log \sigma)$ by Theorem 6.

In the next subsection we review a technique to solve pattern matching queries on a text which exploits the centroid decomposition of the suffix tree of a string. This will allow us to give an efficient algorithm for reconstructing the suffix tree of S, from which S is therefore determined.

Pattern matching with the centroid decomposition. This technique has been introduced by Naor [11] and has found applications, for example, in designing cache-oblivious string B-trees [12, 13] or randomized pattern matching [14] on a dictionary of strings.

The *centroid decomposition* of a tree $\mathcal{T}$ (also known as *separator decomposition*) is a popular and powerful technique to obtain a tree $\mathcal{T}_\mathcal{C}$ of logarithmic height. The decomposition is based on a theorem proved by Jordan in 1869 [15].

Lemma 8. *Any tree $\mathcal{T}$ of n nodes has at least a node, called centroid, whose removal leaves connected components of size at most $n/2$.*

The centroid decomposition is defined recursively. Given $\mathcal{T}$, we identify a centroid node u, which is chosen to be the root of the new rooted tree $\mathcal{T}_\mathcal{C}$. Then, we remove u from $\mathcal{T}$ and recurse on each connected component to get u's subtrees in $\mathcal{T}_\mathcal{C}$. The children of u in $\mathcal{T}_\mathcal{C}$ are the roots of the centroid decompositions of these components. Let us use $\mathsf{children}_{\mathcal{T}_\mathcal{C}}(\mathsf{u})$ to denote the set of children of u in $\mathcal{T}_\mathcal{C}$. As we have a (possibly empty) component for u's parent and children in $\mathcal{T}$, the outdegree of u in $\mathcal{T}_\mathcal{C}$, i.e., $|\mathsf{children}_{\mathcal{T}_\mathcal{C}}(\mathsf{u})|$, is at most the outdegree of

u in $\mathcal{T}$ plus one. The resulting decomposition is a new tree $\mathcal{T}_\mathcal{C}$ on the same nodes, whose height is $\Theta(\log n)$.

A folklore algorithm computes the centroid decomposition in $\Theta(n \log n)$ time as follows. We first observe that a centroid node of $\mathcal{T}$ can be easily identified in linear time. Indeed, we can arbitrary choose a root in $\mathcal{T}$ and visit the tree to compute the size of each subtree. Then, we start from the root and move to the largest subtree until we reach a node whose subtrees have size at most $n/2$. This node is a centroid of the tree. The centroid decomposition is computed by repeating the above algorithm recursively in each component. It easily follows that the decomposition of the tree can be computed in $\Theta(n \log n)$ time. However, there exist construction algorithms to compute the decomposition in linear time [16, 17].

In the following, we will use the centroid decomposition $\mathcal{ST}_\mathcal{C}$ of the suffix tree $\mathcal{ST}$ of a string S. Given a node u in $\mathcal{ST}$, we use $\mathsf{locus(u)}$ to denote its locus, i.e., the string obtained by concatenating the sequence of labels encountered along the path from the root to u.

Assume we are given a pattern $P[1, p]$ and our goal is to find the suffix of S that shares the longest common prefix with P. This problem can be easily solved with the suffix tree $\mathcal{ST}$ of S with the following two-phase strategy: We first identify the highest node u^* in $\mathcal{ST}$ such that $\mathsf{locus(u^*)}$ shares the longest common prefix with P. Then, we try to extend the match by comparing the remaining characters of P with the characters on the edge between u^* and one of its children, i.e., the child where the label starts with the character $P[|\mathsf{locus(u^*)}| + 1]$.

We can perform the same search for u^* on the centroid decomposition $\mathcal{ST}_\mathcal{C}$ of the suffix tree of S. The search is done by traversing a root-to-node path of $O(\log n)$ nodes. We start from the root of $\mathcal{ST}_\mathcal{C}$ and we move down to the leaves. For every node u we visit, we compare $\mathsf{locus(u)}$ with P and decide in which of its children we have to continue the search. As the target node u^* is guaranteed to be visited, we simply take track of the visited node sharing the longest common prefix with P. Based on the result of comparing $\mathsf{locus(u)}$ and P, there are the following cases:

- If $\mathsf{locus(u)}$ equals P, then u is our target node u^* and we conclude.
- If $\mathsf{locus(u)}$ is not a prefix of P, we continue to search on the child of u which corresponds to the connected component containing the parent of u in the suffix tree. If such a node does not exist, we conclude.
- If $\mathsf{locus(u)}$ is a prefix of P, we continue the search on the connected component containing one of the children of u in the suffix tree. The child is the node v such that the first character of the edge between u and v equals the character $P[|\mathsf{locus(u)}| + 1]$. This is exactly the node v that a normal search on the suffix tree would visit next, once the search reaches u. Notice that in general v is not a child of u in the centroid decomposition. If such a node does not exist, we finish the visit.

Let us now show how to use the above algorithm to reconstruct an unknown string S.

Solution with prefix queries to the oracle. We first describe our algorithm for querying the oracle in an easier setting. Instead of answering substring queries, the oracle answers *prefix*

queries: given a string P, the oracle tells us whether P is a prefix of the unknown string S. This model is stronger because it allows us to remain anchored to the beginning of S while reconstructing it.[1] A direct consequence is that the algorithm is easier and faster.

We now describe how to reconstruct a string S with $\Theta(\sigma z_{no} \log n)$ prefix queries to the oracle.

Our algorithm works in steps. In the i-th step it reconstructs the i-th LZ77 phrase Z_i. Once Z_i is reconstructed, the algorithm knows the string $\hat{S}_i$, which is the concatenation of all the phrases reconstructed so far. Observe that Z_i is the longest substring of $\hat{S}_{i-1}$ such that the prefix query $Q_i = \hat{S}_{i-1} \cdot \mathsf{Z}_i$ is answered affirmatively.

The phrase Z_i is identified with $O(\sigma(\log |\hat{S}_{i-1}| + 1))$ prefix queries as follows. Assume we have the suffix tree $\mathcal{ST}$ of $\hat{S}_{i-1}$ and its centroid decomposition $\mathcal{ST}_{\mathcal{C}}$. Our first goal is to identify the lowest node u^* in $\mathcal{ST}$ such that $\hat{S}_{i-1} \cdot \mathsf{locus}(\mathsf{u}^*)$ is a prefix of S. This can be done by performing a search for the unknown pattern $P = \mathsf{locus}(\mathsf{u}^*)$ on $\mathcal{ST}_{\mathcal{C}}$. Even if u^* is unknown, the search can be performed correctly. Indeed, observe that u^* and all its ancestors in $\mathcal{ST}$ are the only nodes u such that $\hat{S}_{i-1} \cdot \mathsf{locus}(\mathsf{u})$ is a prefix of S. Thus, we perform the search on the centroid tree by binary searching for u^* on root-to-u^* path. The cost of the search is $O(\sigma(\log |\hat{S}_{i-1}| + 1))$ prefix queries. Indeed, we need to visit $O(\log |\hat{S}_{i-1}| + 1)$ nodes of $\mathcal{ST}_{\mathcal{C}}$ to identify u^*. For each visited node u, we need a query to check if $\hat{S}_{i-1} \cdot \mathsf{locus}(\mathsf{u})$ is a prefix of S. If this is the case, at most σ queries of the form $\hat{S}_{i-1} \cdot \mathsf{locus}(\mathsf{u}) \cdot \mathsf{c}$, with $c \in \Sigma$, are needed to know in which child of u we have to continue our search. Otherwise, we move to the component containing the parent of u, if any.

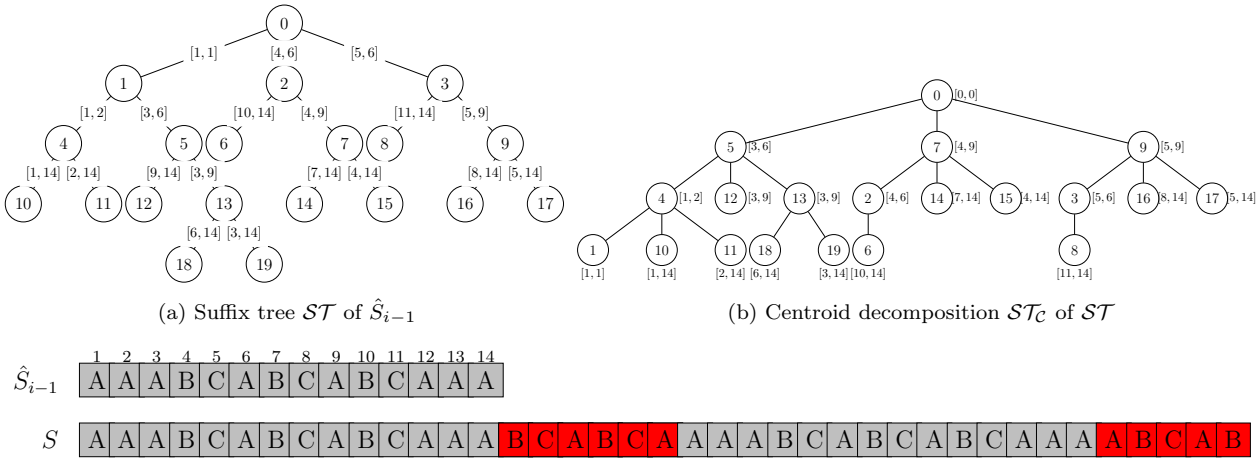
Once we know u^*, we have to extend $\mathsf{locus}(\mathsf{u}^*)$ to match Z_i. Indeed, Z_i may end up in the middle of the edge from u^* to one of its children, say v, in $\mathcal{ST}$. This step can be easily done with $O(\sigma + \log |\mathsf{Z}_i|)$ queries. First, we use $O(\sigma)$ queries to identify the child v of u, then we perform an exponential search on the length of the edge label.

We conclude by proving that the reconstruction of S takes $O(n \log n + n \log \sigma)$ time. A trivial implementation of our algorithm consists in rebuilding at each step i the suffix tree of string $\hat{S}_i$ and its centroid decomposition from scratch. This takes quadratic time.

A faster algorithm is the following: First, we observe that, as the string is reconstructed from left to right, we can use the Ukkonen's construction of the suffix tree [18]. This construction builds the suffix tree in $O(n \log \sigma)$ time and linear space. It is an online algorithm that processes the string from left to right, hence it allows us to build the suffix tree of the prefix of the string that we have already reconstructed.

The centroid decomposition of the suffix tree is instead kept updated dinamically. Brodal et al. [16] showed how to keep an approximation of the centroid decomposition of a tree subject to insertions of new nodes in $O(\log n)$ amortized time per insertion. The decomposition is approximated in the sense that each selected centroid node splits the tree in connected components having a fraction $\frac{1}{2} + \epsilon$ of the overall tree dimension, for any $0 < \epsilon < \frac{1}{4}$. The height of the $\mathcal{T}_{\mathcal{C}}$ is still $O(\log n)$, thus this approximated decomposition suffices for our purposes.

[1] An oracle for prefix queries can be obtained from an oracle for substring queries if we assume that S begins with a special character \$ not belonging to Σ.



(a) Suffix tree $\mathcal{ST}$ of $\hat{S}_{i-1}$

(b) Centroid decomposition $\mathcal{ST}_{\mathcal{C}}$ of $\mathcal{ST}$

(c) The reconstructed string $\hat{S}_{i-1}$ and the unknown string S

Figure 1: A running example

Solution with substring queries to the oracle. The string S can be reconstructed with substring queries using an easy variant of the above algorithm. The algorithm reconstructs (a portion of) the string exactly as described above, until the string reconstructed so far, say $\hat{S}_i$, cannot be further extended to the right, hence we know that $\hat{S}_i$ is a suffix of S. Then, we start extending $\hat{S}_i$ from its beginning, proceeding backwards (that is, prepending characters to $\hat{S}_i$). More formally, our strategy first queries forward strings and builds the suffix tree and the LZ77 factorization of some suffix $S[i, n]$ of the string. Then, we build the suffix tree of $\overleftarrow{S[i, n]}$ and proceed backwards, building the LZ77 factorization of the remaining portion $\overleftarrow{S[1, i-1]}$. Since the size g of the smallest grammar is invariant under reversals and upper-bounds the number of Lempel-Ziv phrases, in both phases we generate at most $O(g)$ phrases. The following theorem is therefore immediate.

Theorem 9. *Let S be a string of length n over the alphabet $\Sigma = [1, \ldots, \sigma]$. There exists an algorithm that reconstructs S with $q = O(\sigma g \log n)$ substring queries to the oracle. The algorithm runs in $O(n(\log n + \log \sigma) + q)$ time using linear space.*

Finally, Theorem 7 follows from the well-known bound $g \in O(z_{no} \log(n/z_{no}))$ (see also Navarro [10]).

Running example. Suppose we have already reconstructed the string $\hat{S}_{i-1} =$ `AAABCABCABCAAA`. This is a substring of the unknown string S shown in Figure 1c. There are two occurrences of $\hat{S}_{i-1}$ in S and the red cells highlight the characters that we still need to learn. The suffix tree $\mathcal{ST}$ of $\hat{S}_{i-1}$ (see Figure 1a) has been built online with Ukkonen's algorithm and it will be updated as soon as we learn more characters. For this reason we do not append any special character at the end of $\hat{S}_{i-1}$. Thus, there may exist suffixes of $\hat{S}_{i-1}$ which do not have their leaves in $\mathcal{ST}$ because they are proper prefixes of some another suffix. In our example this happens to the last three suffixes `A`, `AA` and `AAA` which are proper prefixes of the whole string

$\hat{S}_{i-1}$. Nodes of $\mathcal{ST}$ are numbered (in our example levelwise just for convenience) to map the corresponding node in the centroid decomposition $\mathcal{ST}_{\mathcal{C}}$(see Figure 1b).

The label on the edge from node u to its child v reports the interval $[i, j]$ of positions on string $\hat{S}_{i-1}$ representing the locus of node v. For example, the label on the edge from node 5 to node 13 is $[3, 9]$ and, thus, $\mathsf{locus(13)} = \mathtt{ABCABCA}$. In the centroid decomposition $\mathcal{ST}_{\mathcal{C}}$, each node u is labeled with the interval of positions of $\mathsf{locus(u)}$. The label of node 13 is $[3, 9]$ because $\mathsf{locus(13)} = \mathtt{ABCABCA}$. In the centroid tree, the leftmost child of any node u is the centroid decomposition of the connected component containing the parent of u (if any) while the other children are the centroid decompositions of the subtrees rooted at the children of u in $\mathcal{ST}$(if any). Note that for any child v of node u but, possibly, the leftmost one, we have that $\mathsf{locus(u)}$ is a prefix of $\mathsf{locus(v)}$.

We start from the root of $\mathcal{ST}_{\mathcal{C}}$ (node 0) which in our example, by coincidence, corresponds to the root of $\mathcal{ST}$. We query the oracle for the substring $\hat{S}_{i-1} \cdot \mathsf{locus(0)}$. As $\mathsf{locus(0)}$ is the empty string, the oracle's answer will be positive. The algorithm continues on a child of node 0.

For any child v of u, let be c_v the character such that $\mathsf{locus(u)} \cdot \mathsf{c_v}$ is a prefix of $\mathsf{locus(v)}$, i.e., $c_v = \mathsf{locus(v)}[|\mathsf{locus(u)}| + 1]$. We process the children of node 0 and continue on a node v such that the query for the substring $\hat{S}_{i-1} \cdot \mathsf{locus(0)} \cdot \mathsf{c_v}$ is successful. There may be several such nodes v. For example, we could continue on both nodes 5 and 7. This is because both substrings $\hat{S}_{i-1} \cdot \mathtt{A}$ and $\hat{S}_{i-1} \cdot \mathtt{B}$ occur in S. We can arbitrarily choose any of these nodes but, of course, the length of the substring we reconstruct may vary. Suppose we continue on node 5. Then, we ask for the substring $\hat{S}_{i-1} \cdot \mathsf{locus(5)}$. As the answer is positive, we continue with node 12 because $c_{12} = \mathtt{B}$ and $\hat{S}_{i-1} \cdot \mathsf{locus(5)} \cdot \mathsf{c_{12}}$ occurs in S. We query for $\hat{S}_{i-1} \cdot \mathsf{locus(12)}$. As the answer is negative, we binary search for the longest prefix P of $\mathsf{locus(12)}$ such that $\hat{S}_{i-1} \cdot P$ occurs in S. This way, we reconstruct the substring $X = \mathtt{ABCAB}$ and we learn $\hat{S}_i = \hat{S}_{i-1} \cdot X$.

4. Conclusions and future work

We investigated the connection between the string reconstruction and compression problems, establishing that they essentially coincide: the number of substring queries that need to be asked to an oracle in order to reconstruct a string S is proportional to the complexity of S.

We also showed that it is possible to efficiently reconstruct a string of length n over an alphabet of size σ using $O(\sigma g \log n) \subseteq O(\sigma \cdot z_{no} \log(n/z_{no}) \log n)$ queries in $O(n(\log \sigma + \log n))$ time, where z_{no} is the number of phrases of the LZ77 factorization of S without overlaps and g is the size of the smallest grammar producing S. Immediate improvements over our work would be to replace z_{no} with the more powerful z (i.e., allowing overlaps), or to shave log factors from the complexities of our algorithms. In particular, we know that the number of queries cannot be improved by more than a factor $O(\log \frac{n}{z_{no}} \log \sigma)$ in general.

In our setting, we aim at reconstructing the whole unknown string S. One can also consider the problem of reconstructing the set of all substrings of S of a given length k, see for example [4]. Notice that knowing all the substrings of S of length $r(S) + 2$ allows one

to uniquely determine S, where $r(S)$ is the repetition index of S, that is, the length of the longest repeat of S [19, 20].

Another direction of investigation consists in introducing uncertainty into the model. For example, allowing the oracle to answer the queries with a certain probability of returning a wrong result — this could model strings with character ambiguities, e.g., DNA strings arising from a sequencing — or allowing the oracle to return positive answers to queries within a limited Hamming distance from substrings of the target string.

References

[1] T. Jiang, M. Li, DNA sequencing and string learning, Mathematical systems theory 29 (4) (1996) 387–405. doi:10.1007/BF01192694.

[2] D. Margaritis, S. Skiena, Reconstructing strings from substrings in rounds, in: 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995, IEEE Computer Society, 1995, pp. 613–620. doi:10.1109/SFCS.1995.492591.

[3] S. Skiena, G. Sundaram, Reconstructing strings from substrings, J. Comput. Biol. 2 (2) (1995) 333–353. doi:10.1089/cmb.1995.2.333.

[4] D. Tsur, Tight bounds for string reconstruction using substring queries, in: C. Chekuri, K. Jansen, J. D. P. Rolim, L. Trevisan (Eds.), Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th International Workshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings, Vol. 3624 of Lecture Notes in Computer Science, Springer, 2005, pp. 448–459. doi:10.1007/11538462_38.

[5] J. Acharya, H. Das, O. Milenkovic, A. Orlitsky, S. Pan, String reconstruction from substring compositions, SIAM J. Discret. Math. 29 (3) (2015) 1340–1371. doi:10.1137/140962486.

[6] A. W. M. Dress, P. L. Erdős, Reconstructing words from subwords in linear time, Annals of Combinatorics 8 (4) (2005) 457–462. doi:10.1007/s00026-004-0232-4.

[7] R. Afshar, A. Amir, M. T. Goodrich, P. Matias, Adaptive exact learning in a mixed-up world: Dealing with periodicity, errors and jumbled-index queries in string reconstruction, in: SPIRE 2020: Proceedings of the 27th International Symposium on String Processing and Information Retrieval, Vol. 12303 of Lecture Notes in Computer Science, Springer, 2020, pp. 155–174. doi:10.1007/978-3-030-59212-7_12.

[8] K. Iwama, J. Teruyama, S. Tsuyama, Reconstructing strings from substrings: Optimal randomized and average-case algorithms, CoRR abs/1808.00674 (2018).
URL `http://arxiv.org/abs/1808.00674`

[9] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343. doi:10.1109/TIT.1977.1055714.

[10] G. Navarro, Indexing highly repetitive string collections (2020). arXiv:2004.02781.

[11] M. Naor, String matching with preprocessing of text and pattern, in: ICALP 1991: Proceedings of the 18th International Colloquium on Automata, Languages, and Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 739–750.

[12] M. A. Bender, M. Farach-Colton, B. C. Kuszmaul, Cache-oblivious string b-trees, in: PODS 2006: Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2006, pp. 233–242. doi:10.1145/1142351.1142385.

[13] P. Ferragina, R. Venturini, Compressed cache-oblivious string B-tree, ACM Transactions on Algorithms (TALG) 12 (4) (2016) 52:1–52:17. doi:10.1145/2903141.

[14] A. Amir, M. Farach, Y. Matias, Efficient randomized dictionary matching algorithms, in: Combinatorial Pattern Matching, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 262–275.

[15] C. Jordan, Sur les assemblages de lignes, Journal für die reine und angewandte Mathematik 70 (1869) 185–190.

[16] G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, A. Östlin, The complexity of constructing evolutionary trees using experiments, in: ICALP 2001: Proceedings of the 28th International Colloquium on Automata, Languages and Programming, 2001, pp. 140–151. doi:10.1007/3-540-48224-5_12.

[17] D. Della Giustina, N. Prezza, R. Venturini, A new linear-time algorithm for centroid decomposition, in: SPIRE 2019: Proceedings of the 26th International Symposium on String Processing and Information Retrieval, Springer, 2019, pp. 274–282. doi:10.1007/978-3-030-32686-9_20.

[18] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260. doi:10.1007/BF01206331.

[19] A. Carpi, A. de Luca, Words and special factors, Theoret. Comput. Sci. 259 (1-2) (2001) 145–182. doi:10.1016/S0304-3975(99)00334-5.

[20] G. Fici, F. Mignosi, A. Restivo, M. Sciortino, Word assembly through minimal forbidden words, Theor. Comput. Sci. 359 (1-3) (2006) 214–230. doi:10.1016/j.tcs.2006.03.006.