Incongruity-Sensitive Access to Highly Compressed Strings

Ferdinando Cicalese
Department of Computer Science, University of Verona, Italy

Travis Gagie
Center for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Zsuzsanna Lipták
Department of Computer Science, University of Verona, Italy

Gonzalo Navarro
IMFD & Department of Computer Science, University of Chile, Santiago, Chile

Nicola Prezza
Department of Environmental Sciences, Informatics and Statistics,
Ca' Foscari University, Venice, Italy

Cristian Urbina
Institute of Informatics, University of Warsaw, Poland

Abstract

Random access to highly compressed strings – represented by straight-line programs or Lempel-Ziv parses, for example – is a well-studied topic. Random access to such strings in strongly sublogarithmic time is impossible in the worst case, but previous authors have shown how to support faster access to specific characters and their neighbourhoods. In this paper we explore whether, since better compression can impede access, we can support faster access to less compressible substrings of highly compressed strings. We first show how, given a run-length compressed straight-line program (RLSLP) of size g_{rl} or a block tree of size L, we can build an $O(g_{rl})$-space or an $O(L)$-space data structure, respectively, that supports access to any character in time logarithmic in the length of the longest repeated substring containing that character. That is, the more "incongruous" a character is with respect to the characters around, the faster we can support access to it. We then prove a similar but more powerful and sophisticated result for parsings in which phrases' sources do not overlap much larger phrases, with the query time depending also on the number of phrases we must copy from their sources to obtain the queried character.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Data compression, parsing, straight-line program, random access, grammar compression, run-length grammar, longest repeated substring, distance-sensitive predecessor data structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.125

Funding *Travis Gagie*: Funded by NSERC grant RGPIN-07185-2020.
Zsuzsanna Lipták: Funded by INdAM – GNCS Project CUP E53C25002010001.
Gonzalo Navarro: Funded by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1260080, ANID, Chile.
Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the authors 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.
Cristian Urbina: Funded by the Polish National Science Center, grant no. 2022/46/E/ST6/00463.

1 Introduction

There has been a lot of work on supporting fast worst-case random access to highly compressed strings [1, 3, 4, 6, 7, 9, 10, 16, 21, 23, 24, 26, 28, 32, 34, 36] (see [22] for a recent discussion). For the purposes of this paper, "highly compressed" means stored in space that depends polynomially on a strong measure of compressibility – the size of the smallest straight-line program, the normalized substring complexity, or the size of the smallest string attractor, for example – and only polylogarithmically on the length of the string. We now know, for example, that we cannot support strongly worst-case sublogarithmic-time access to highly compressed strings [36], but also how to store a block tree [3, 26] for a string $S[1\,..\,n]$ over an alphabet of size σ in optimal $O\left(\delta \log \frac{n \log \sigma}{\delta \log n}\right) \subseteq O(\delta \log n)$ space, where δ is the normalized substring complexity [26], and support access in worst-case $O(\log(n/\delta))$ time. Other researchers have shown how to bookmark [15, 12] a character or put a dynamic finger [5] on it, such that queries regarding neighbouring characters can be answered in time logarithmic in their distance from the bookmark or finger.

Worst-case bounds are only concerned with the characters hardest to reach, however, and bookmarks and fingers can be applied to arbitrary characters. It is natural to ask whether the standard representations of highly compressed strings – straight-line programs or LZ77 parses, for example – treat all characters more or less equally or automatically make some characters easier to reach. In fact, over twenty years ago Gasieniec et al. [19, 18] showed how we can easily access characters at the ends of non-terminals' expansions in a straight-line program, and Claude and Navarro [11] used this in their grammar-based indexes. More generally, though, if better compression tends to slow down random access, then perhaps it is easier to access characters in less compressible areas of a compressed string. Those areas are typically the most interesting ones when analyzing repetitive collections, for example they correspond to mutations or rare variants in genomes. In this paper we take a step toward demonstrating this by showing how some standard representations of highly compressible strings can be augmented – without asymptotically increasing their space usage – to support fast access to characters that are incongruous with respect to their neighbours, in the sense that the longest repeated substrings containing those characters are short.

In Section 2 we review notation and definitions we need for working with straight-line programs (SLPs), run-length SLPs (RLSLPs), and parses. In Section 3 we first show how, given an RLSLP of size g_{rl} for S that is balanced in a "local" sense, we can build an $O(g_{rl})$-space data structure that supports access to any character $S[q]$ in time $O(\log \ell_q)$, where ℓ_q is the length of the longest repeated substring containing $S[q]$; see Lemma 1. It follows that we can store S in $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space and support $O(\log \ell_q)$-time access (Corollary 2). We then extend our result to block trees (Corollary 3) and arbitrary RLSLPs (Corollary 5). We then prove a similar but more powerful and sophisticated result for parsings (Theorem 9): Given a parsing with t phrases in which any phrase's source overlaps only phrases at most $w^{O(1)}$ times larger, where w is the machine word size, we can store S in $O(t)$ space and access $S[q]$ in $O(h_q + \log_w \ell_q)$ time, where the height h_q of $S[q]$ is the number of phrases we must copy from their sources to obtain $S[q]$. We conclude by showing in Lemma 14 that we can convert any bidirectional parse with b phrases into one with this constraint on source-phrase overlaps, at the cost of an $O(\log_w(n/b))$-factor increase in size but with no character's height increasing. It follows that, given a bidirectional parsing with b phrases, we can build an $O(b \log_w(n/b))$-space data structure supporting $O(h_q + \log_w \ell_q)$-time access; see Corollary 15. Along the way, we obtain various results of independent interest, for example that RLSLPs can be made to have logarithmic height without asymptotically increasing their size and

retaining local balance (Theorem 4) and that linear-space data structures exist to compute predecessors in distance-sensitive time $O(\log\log_w \Delta)$, where Δ is the distance from the query to the answer (Theorem 13).

2 Preliminaries

We denote by $[i,j]$ the integer interval $\{i, i+1, \ldots, j\}$, and $[i,j) = [i,j-1]$. Let $\Sigma = \{a_1, \ldots, a_\sigma\}$ be a finite set of characters called the *alphabet*. A *string* is a sequence $S[1\,..\,n] = S[1]\cdots S[n]$ of length $|S| = n$, where $S[i] \in \Sigma$ for all $i \in [1,n]$. The unique string of length 0, called the *empty string*, is denoted ε. A *substring* of $S[1\,..\,n]$ is any subsequence $S[i\,..\,j] = S[i]\cdots S[j]$, where $i, j \in [1,n]$, and we let $S[i\,..\,j] = \varepsilon$ when $i > j$. If $i = 1$ then $S[i\,..\,j]$ is called a *prefix*, and analogously, if $j = n$ then $S[i\,..\,j]$ is called a *suffix*. If $S[1\,..\,n]$ and $T[1\,..\,m]$ are strings, their concatenation is the string $(S\cdot T)[1\,..\,n+m] = S[1]\cdots S[n]T[1]\cdots T[m]$. We denote by S^k the concatenation of k copies of the string S. The *normalized substring complexity* of S, usually written δ, is the maximum over k of $1/k$ times the number of distinct k-tuples in S, that is, $\delta(S) = \max_{k\in[1,n]} |\{S[i\,..\,i+k-1] \,|\, i \in [1, n-k+1]\}|/k$. To simplify notation, by log we will mean $\log_2$, and when writing $\log_d x$ we will actually mean $\max(1, \log_d x)$ except where otherwise noted. We use the transdichotomous word RAM model of computation with words of size $w \geq \log n$, where n is the length of the uncompressed string.

2.1 Grammars

A *straight-line program (SLP)* is a context-free grammar (CFG) $G = (V, \Sigma, R, I)$, where V is the set of *nonterminals*, Σ is an alphabet disjoint from V, $R \subseteq V \times (\Sigma \cup V)^*$ is the set of *rules*, and $I \in V$ is the *initial nonterminal*. An SLP must have exactly one rule per nonterminal, be in Chomsky normal form (i.e., rules are of the form $A \to BC$ or $A \to a$, for nonterminals A, B, C and terminal a), and have no cycles (i.e., for any nonterminal A, we can not reach nonterminal A again by following the rules). In such a case, G generates exactly one string. We call $\exp(A)$ the string nonterminal A expands to, that is, $\exp(A) = \exp(B)\cdot\exp(C)$ if $A \to BC$, and $\exp(A) = a$ if $A \to a$. Further, we extend the function to $\exp(a) = a$ for terminals a. The string generated by G is then $S[1\,..\,n] = \exp(I)$.

A generalization of SLPs called *run-length SLP (RLSLP)* allows rules of the form $A \to B^k$ for integer $k > 2$, meaning $\exp(A) = \exp(B)^k$. The size of an SLP or RLSLP is defined as its number of rules and is usually called g or g_{rl}, respectively.

The *parse tree* of an SLP G is a binary tree whose root is labeled with the start symbol of G, the leaves with terminal symbols (i.e., the symbols of S if read left to right), and every internal node labeled with a nonterminal A, whose rule is $A \to BC$, has two children, the left one labeled B and the right one labeled C. In case of RLSLPs, rules $A \to B^k$ are represented as a node labeled A with k children labeled B. The parse tree always has exactly n leaves.

Any symbol $S[q]$ from an SLP or RLSLP can be extracted in time proportional to the height of its parse tree. We start aiming to extract $\exp(I)[q]$. In general, to extract $\exp(A)[q]$, where $A \to BC$, we recursively extract $\exp(B)[q]$ if $|\exp(B)| \geq q$, and $\exp(C)[q - |\exp(B)|]$ otherwise. For RLSLP rules $A \to B^k$, we reduce $\exp(A)[q]$ to $\exp(B)[1 + ((q-1) \bmod |\exp(B)|)]$. The recursion ends with $\exp(A)[1] = a$ when $A \to a$.

We will also use the so-called *grammar tree*. This is obtained by pruning from the parse tree, for each nonterminal label A, all but one occurrence of nodes with label A. Pruning means converting those other internal nodes to leaves. For RLSLP rules $A \to B^k$, the internal grammar tree node labeled A has a left child labeled B (which can be pruned or not) and a

special right child, labeled B^{k-1}, which is a leaf. It is easy to see that the grammar tree of an SLP or RLSLP has exactly $2g+1$ or $2g_{rl}+1$ nodes, respectively [30]. The grammar tree enables algorithms that run on the parse tree to use space $O(g)$ or $O(g_{rl})$ instead of $O(n)$.

An SLP or RLSLP is *balanced* if it has height $O(\log n)$ and *locally balanced* [8, Def 4.2] if there is a constant c such that every nonterminal A has height at most $c \cdot \log|\exp(A)|$ in the parse tree. A balanced SLP or RLSLP therefore supports worst-case logarithmic-time access. With a locally balanced SLP or RSLP, moreover, when given any nonterminal A and a position q we can access $\exp(A)[q]$ in $O(\log|\exp(A)|)$ time by recursively descending from A to a leaf in the parse tree.

2.2 Parses

In general, a *parsing* cuts $S[1..n]$ into substrings called *phrases*, $S = B_1 \cdots B_t$, each B_i being either explicit (and of length at most $O(\log_\sigma n)$ and fitting into $O(1)$ space) or a copy of another substring $S[p_i..p_i+|B_i|-1]$, starting at some position $p_i \neq \sum_{j=1}^{i-1}|B_j|+1$, called B_i's *source*. It follows that the phrase containing $S[q]$, if not explicit, has length at most ℓ_q, where ℓ_q is the length of the longest repeated substring containing $S[q]$. The number of phrases is called the parsing's *size*. If a phrase's sources are always to their left or always to their right then the parsing is called *unidirectional*, otherwise it is called *bidirectional*.

Let $x_1, \ldots, x_t$ be the starting positions of the phrases $B_1, \ldots, B_t$, and $x_{t+1} = n+1$. To access S from the parsing, we define $f(q) = q$ if the phrase B_i that $S[q]$ belongs to is explicit, and $f(q) = p_i + q - x_i$ otherwise; note that $S[q] = S[f(q)]$. To obtain $S[q]$, we apply f iteratively until we reach a character in an explicit phrase. If this never happens then S cannot be decoded from the parse.

Recall that, given a set of integers $x_1 \leq x_2 \leq \ldots \leq x_k$ and an integer q, a *predecessor query* $pred(q)$ returns the largest x_i such that $x_i \leq q$. The *height* h_q of $S[q]$ is the number of times we must apply f to q before reaching a character in an explicit block, and $S[q]$'s *referencing chain* is $q, f(q), \ldots, f^{h_q}(q)$. We can thus extract $S[q]$ in h_q+1 steps. If we use a predecessor data structure to find the phrase containing $f^k(q)$ for $0 \leq k \leq h_q$ – that is, each B_i such that x_i is the maximum value with $x_i \leq f^k(q)$ – then we can access $S[q]$ in time $O((h_q+1)\log\log n)$. We write simply h to denote the maximum height of any character in S.

3 Incongruity-sensitive access with RLSLPs and block trees

Suppose we are given a locally balanced SLP or RLSLP. The g' leaves of its grammar tree naturally partition S into g' substrings: let $A_1, \ldots, A_{g'}$ be the leaves' labels, where A_i can be a terminal or a nonterminal (or, with RLSLPs, an iteration B^{k-1}). The ith substring of the partition is then $\exp(A_i)$, so $S = \exp(A_1) \cdots \exp(A_{g'})$. Let $x_1, \ldots, x_{g'}$ be the starting positions of those substrings in S, and $x_{g'+1} = n+1$, that is, $x_1 = 1$ and $x_{i+1} = x_i + |\exp(A_i)|$.

We build a predecessor data structure on the values x_i for $1 \leq i \leq g'$, storing with each x_i the label A_i. This yields the following algorithm to extract $S[q]$:

1. use a predecessor query to find the largest $x_i \leq q$;
2. extract the $(q - x_i + 1)$th symbol of $\exp(A_i)$.

With a standard predecessor data structure the time for the predecessor query could dominate the $O(\log|\exp(A_i)|)$ extraction time. We can avoid this, however, by using a distance-sensitive predecessor data structure [2, 13]. From now on we focus on RLSLPs, which are more general. Recall that ℓ_q denotes the length of the longest repeated substring containing $S[q]$.

▶ **Lemma 1.** *Let a locally balanced RLSLP of size g_{rl} generate $S[1\,..\,n]$. Then there exists a data structure of size $O(g_{rl})$ that can access any $S[q]$ in time $O(\log \ell_q)$.*

Proof. We define $x_1, \ldots, x_{g'}$ as above and implement the 2-point algorithm above. For the first point, we use a linear-space distance-sensitive predecessor data structure [2, 13], which uses $O(g') = O(g_{rl})$ space and provides query time loglogarithmic on the distance between the query q and its answer x_i, that is, $O(\log\log(q - x_i + 1)) \subseteq O(\log\log(x_{i+1} - x_i))$; note $x_{i+1} - x_i = |\exp(A_i)|$.

We note that no leaf of the grammar tree labeled by a nonterminal can contain a unique substring, because by definition there must be another internal node with the same label, thus generating the same substring. This also holds for leaves labeled B^{k-1}, as there are at least two occurrences of $\exp(B)^{k-1}$ in $\exp(A) = \exp(B)^k$. It follows that $x_{i+1} - x_i \leq \ell_q$ and thus the predecessor data structure takes time $O(\log\log \ell_q)$.

Point 2 takes time $O(\log(x_{i+1} - x_i)) \subseteq O(\log \ell_q)$ because the RLSLP is locally balanced: each recursive step reduces the height of the current parse tree node by at least 1. ◀

This is similar in flavor to how we can shorten AVL grammars to have height $O(\log(n/g))$ rather than $O(\log n)$ [32], and support $O(\log(n/g))$-time access on them, but that technique does not yield incongruity-sensitive access bounds.

Charikar et al.'s α-balanced grammars [7] and Rytter's AVL grammars [32] are locally balanced, for example, so they can be used to support the faster access time. So are the RLSLPs of Christiansen et al. [8] and of Kociumaka et al. [25]. The $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ space upper bound of the second one [25] yields the following result.

▶ **Corollary 2.** *Let $S[1\,..\,n]$, over alphabet $[1\,..\,\sigma]$, have normalized substring complexity δ. Then there is a structure of size $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ that can access any $S[q]$ in time $O(\log \ell_q)$.*

The same idea can be applied to other compressed hierarchical representations not based on grammars. In particular, a *block tree* [3] on S is defined by hierarchically partitioning it into perfect halves until the leaves. Every node is called a block and represents a substring of S. We then prune every internal block of length m such that the substrings of length $2m$ starting with it and ending with it occur earlier in S (but not overlapping it; we say that a substring $S[i\,..\,j]$ overlaps a substring $S[i'\,..\,j']$ if $[i, j] \cap [i', j'] \neq \emptyset$). Those pruned nodes become then leaves in the block tree, and store a pointer to their source, which are the (one or two) blocks of the same length covering the leftmost occurrence of the block's content in S. A symbol $S[q]$ is accessed recursively, by going down left or right from the internal nodes, or shifting to the source of leaves. Since we always go down after having shifted to a source, and child block lengths are half the length of the block, access takes time $O(\log n)$.

The block tree satisfies almost the same conditions we have used to prove Lemma 1, if we replace grammar-tree leaves by block-tree leaves.

▶ **Corollary 3.** *Let $S[1\,..\,n]$ be represented by a block tree of size L. Then there exists a data structure of size $O(L)$ that can access any $S[q]$ in time $O(\log \ell_q)$.*

Proof. We define $x_1, \ldots, x_L$ as the starting position of the block tree leaves, and $x_{L+1} = n+1$. We build and use the same distance-sensitive predecessor data structure of the proof of Lemma 1, which uses $O(L)$ space and finds the leaf containing $S[q]$ in time $O(\log\log m)$, where $m = x_{i+1} - x_i$ is the leaf length.

The string of the leaf that contains $S[q]$ occurs earlier in S, since that leaf was created. It follows that $\ell_q \geq m$. The process of extracting $S[q]$ from the position $q - x_i + 1$ of the leaf takes time $O(\log m) = O(\log \ell_q)$. ◀

By proper parameterization, the size of block trees can be bounded by $O(\delta \log \frac{n \log \sigma}{\delta \log n})$ [26, Cor. VI.7], so we could prove Corollary 2 from Corollary 3. This could be useful because block trees can be smaller than RLSLPs on some texts.

We now aim at locally balancing a grammar in order to extend Lemma 1 to every RLSLP. There have been several useful results about balancing SLPs and RLSLPs recently, but none that exactly fit our goal. Ganardi et al. [17] showed how, given an SLP of size g, we can build a balanced SLP of size $O(g)$ of the same string. Navarro et al. [31] generalized this result to RLSLPs, but the resulting grammars might not be locally balanced: even if the parse tree has height $O(\log n)$, if the nonterminal A_i whose expansion we extract from in point (2) of our algorithm has height $\omega(\log |\exp(A_i)|)$, the extraction can take $\omega(\log \ell_q)$ time.

Ganardi [16] showed it is generally impossible, given an SLP of size g for S, to build an SLP of size $O(g)$ for S whose parse tree is weight-balanced (the sizes of any two siblings' subtrees are within constant factors of each other) or just height-balanced (the heights of any two siblings' subtrees differ by at most a constant). Fortunately, just as all weight-balanced trees are height-balanced but not vice versa, all SLPs with height-balanced parse trees are locally balanced but not all locally balanced SLPs have height-balanced parse trees. Ganardi therefore left open the possibility of there always being a locally balanced SLP of size $O(g)$ for S, and in fact he showed how we can build one.

We now generalize Ganardi's construction from SLPs to RLSLPs and prove the following theorem, which may be of independent interest, to extend Lemma 1 to arbitrary RLSLPs.

▶ **Theorem 4.** *Given an RLSLP for S with g_{rl} rules, in $O(g_{rl})$ time we can build a locally balanced RLSLP for S with $O(g_{rl})$ rules.*

▶ **Corollary 5.** *Let an RLSLP of size g_{rl} generate $S[1..n]$. Then there exists a data structure of size $O(g_{rl})$ that can access any $S[q]$ in time $O(\log \ell_q)$.*

To prove Theorem 4, we show how to transform an RLSLP into a CFG (with run-length rules, which we call a RLCFG) that generates the same string and is *contracting*, that is, if B is a child of A, then $|\exp(B)| \leq |\exp(A)|/2$. This property implies local balance. That contracting RLCFG is not yet an RLSLP because some right-hand sides have length more than two, but still bounded by a constant. Finally, we transform the contracting RLCFG again into an RLSLP by putting it into Chomsky normal form. This loses the contracting property but retains local balance (with an increased constant).

We borrow some results from Ganardi's work on constructing contracting CFGs [16], and show that (almost) the same ideas work to produce contracting RLCFGs.

Consider a (RL)CFG $G = (V, \Sigma, R, S)$. We call the *variables*, $V \cup \Sigma$, the nonterminals and terminals of G. A variable $B \in V$ is a *heavy child* of $A \in V$ with rule $A \to u$ if B appears in u and $|\exp(B)| > |\exp(A)|/2$. A rule $A \to u$ is *contracting* if u contains no heavy variables. Naturally, run-length variables and terminal symbols are considered contracting. If all rules in G are contracting, we call G contracting.

A *labeled tree* $T = (V, E, \eta)$ is a rooted tree where each edge $e \in E$ is labeled by a string $\eta(e)$. A prefix in T is the concatenation of the labels in any path starting from the root.

The *DAG* of $G = (V, \Sigma, R, S)$ is obtained from its grammar tree, by identifying every leaf labeled with a nonterminal A with the only internal node labeled A, and all the leaves labeled with a terminal symbol with a single node. The DAG contains exactly one node labeled with each variable; we will identify nodes and labels.

The *heavy forest* $H = (V, E_H)$ of G contains all edges (A, B) where B is a heavy child of A, and it is a subgraph of the DAG of G. Notice that the edges in H point towards the roots, that is, if $(A, B) \in E_H$ then A is a child of B in H. The ending points of those paths in E_H (i.e., their lowest nodes in the DAG) are called *roots*. Isolated nodes, in particular all terminal and run-length variables of G, are also roots.

Let us now assume G is an (RL)SLP. We define two labeling functions: The *left label* $\lambda(e)$ of an edge $e = (A, B) \in E_H$ is the left light child of A (if it exists, ε otherwise) and the *right label* $\rho(e)$ of e is the right light child of A (if it exists, ε otherwise). The connected components of (H, λ) and (H, ρ) are called the *left labeled and right labeled heavy trees*, which can be computed in linear time from G. If A_0 is a variable in a heavy path $A_0, e_1, A_1, \ldots, e_k, A_k$ with root A_k, where $k > 0$, we can factorize A_0 into the left labeling from A_0 to A_k (which can be obtained by reversing the left labeling from the root A_k to A_0), the root A_k, and the right labeling of the path from A_k to A_0. Indeed, $\exp(A_0) = \exp(\lambda(e_1))\exp(A_1)\exp(\rho(e_1))$, and in general, $\exp(A_i) = \exp(\lambda(e_{i+1} \ldots e_k))\exp(A_k)\exp(\rho(e_k \ldots e_{i+1}))$.

In that way one can redefine every variable using SLPs that define all prefixes (from the roots) in the left-labeled and the right-labeled heavy trees.

▶ **Theorem 6** ([16, Thm. 6]). *Given a labeled tree T with $|T|$ edges and labels of length $\leq t$, one can compute in linear time a contracting grammar with $O(|T|)$ variables and right-hand sides of length $O(t)$ defining all prefixes in T.*

As an example, let $S = abracad(abra)^7cabra$. Let A, B, C, D, R be the variables generating a, b, c, d, and r, respectively. Below, we exhibit an RLSLP that generates S.

Rule	$\exp(\cdot)$	$\lvert\exp(\cdot)\rvert$	Heavy child
$A_0 \to A_1A_4$	$abracad(abra)^7cabra$	40	A_1
$A_1 \to A_2A_3$	$abracad(abra)^7$	35	A_3
$A_2 \to A_5A_6$	$abracad$	7	A_5
$A_3 \to A_5^7$	$(abra)^7$	28	None
$A_4 \to CA_5$	$cabra$	5	A_5
$A_5 \to A_7A_8$	$abra$	4	None
$A_6 \to A_9D$	cad	3	A_9
$A_7 \to AB$	ab	2	None
$A_8 \to RA$	ra	2	None
$A_9 \to CA$	ca	2	None

The heavy forest of this RLSLP is shown in Figure 1. Note how the labels in the edges have $O(1)$ length, and the number of edges is no more than the number of variables of the RLSLP.

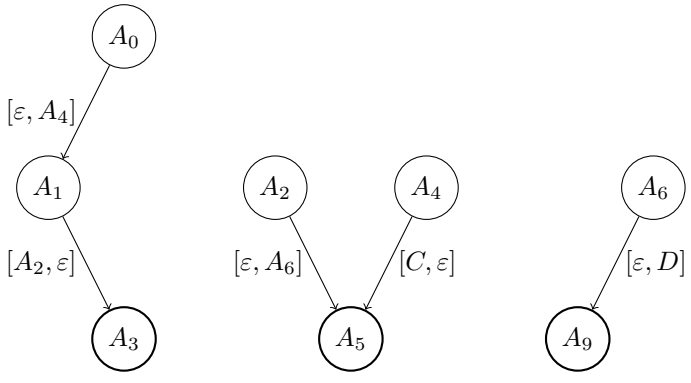

Figure 1 Heavy forest of an RLSLP generating the string $S = abracad(abra)^7cabra$ (we omit trees containing only one node). The edges are labeled with the left and right labels corresponding to each non-root variable. We use bold circles to highlight root nodes.

The grammars obtained using Ganardi's procedure are not necessarily in Chomsky normal form. Their procedure actually allows that rules in its input grammars are of different lengths. This allows accommodating our run-length rules. Run-length variables, if unfolded, would be roots in H under Ganardi's original definition of contracting variables. Hence, the labeled trees $H_L = (H, \lambda)$ and $H_R = (H, \rho)$ have $O(|G|)$ edges with labels of length at most 1. We obtain the following corollary.

▶ **Corollary 7.** *Let G be an RLSLP. One can build in $O(|G|)$ time a contracting RLCFG G_L of size $O(|G|)$ and right-hand sides of length $O(1)$ defining all prefixes in H_L.*

Proof. We apply Theorem 6 to all connected components of H_L. As the number of edges in H_L is no more than $|G|$, and the length of the labels is at most 1, the claim holds. ◀

An analogous result holds for H_R. Now we are ready to prove Theorem 4.

Proof of Theorem 4. Let G be an RLSLP generating a string S with size g_{rl}. We apply the construction of Corollary 7. For each non-root variable A of a connected component of H, let B be its root. Let $X_L \in G_L$ and $X_R \in G_R$ be variables generating the reversed left-labeled prefix and right-labeled prefix of the path from the root B to A. We observe that A derives the same string as $X_L \cdot B \cdot X_R$. To ensure A is locally contracting, we replace the right-hand side of the rule of A by the concatenation of the right-hand sides of the rules of X_L, B, and X_R. Because X_L, B, and X_R are contracting, the resulting rule for A is also contracting. Observe that the length of the new rule for A is $O(1)$. The root variables (which includes run-length variables of G) are not modified. The size of this equivalent RLCFG G' is $O(g_{rl})$, and all its rules are contracting, hence G' is contracting. Because rules are constant size, we can apply standard techniques to transform the non-run-length rules of G' into Chomsky normal form. This incurs only a constant increase in height, and so retains local balance. ◀

4 Incongruity-sensitive access with parses

We have actually already parsed S for Lemma 1, into $S = \exp(A_1) \cdots \exp(A_{g'})$, since each leaf in the grammar tree is either a terminal or a non-terminal that appears as an internal node elsewhere in the grammar tree. Indeed, we can view Lemma 1 as applying to a special kind of parse with two useful properties: first, each character $S[q]$ has height h_q at most logarithmic in the length of the phrase containing $S[q]$; second, sources' boundaries align nicely with phrases' boundaries in such a way that when extracting $S[q]$ we need perform only one predecessor query, to find the phrase containing $S[q]$. Needless to say, most parsings do not have these properties.

Typical parsings [27, 35] have good bounds on the number t of phrases, but do not provide upper bounds for the maximum height h other than t, and sometimes only the length n of the uncompressed string. For example, the original Lempel-Ziv parse [27] (whose parsing size is called z) and bidirectional macro schemes [35] (whose optimal parsing size is called b) can achieve less space than the smallest grammars and still recover S. It always holds $\delta \leq b \leq z \leq 2g^*_{rl} \leq 2g^*$, where g^* and g^*_{rl} are the sizes of the smallest SLP and RLSLP for S, respectively; the differences are asymptotically strict for some string families [29].

Some parsings increase t in order to obtain better upper bounds on h [23, 24, 28, 1, 34], but those with $t \in O(\gamma \log(n/\gamma))$ do not achieve $h = o(\log(n/t))$. Here γ, with $\delta \leq \gamma \leq b$, is the size of the smallest string attractor for S [23], so the minimum number of characters in S we need to select such that at least one selected character appears in some occurrence of any substring of S. For example, we can have $h = O(\log n)$ with $t = O(g^*_{rl})$ with a unidirectional

parsing [1], or $h = O(\log(n/\gamma))$ with $t = O(\gamma \log(n/\gamma))$ with a bidirectional parsing [23]. For some string families, these parsings can have t asymptotically smaller than the sizes of the smallest RLSLPs or block trees.

Our main result in this section is to improve the access time for $S[q]$ from our baseline of $O((h_q + 1) \log \log n)$ to $O(h_q + \log_w \ell_q)$ for a class of "well-behaved" parses.

▶ **Definition 8.** *A parse $S = B_1 \cdots B_t$ is* α-contracting *if no source $S[p_i \,..\, p_i + |B_i| - 1]$ overlaps a phrase B_j with $\alpha|B_i| < |B_j|$ (i.e., B_j is strictly more than α times longer than B_i).*

Notice that a c-contracting parsing is also c'-contracting for any $c' \geq c$. Crucially, if a parse is α-contracting with $\alpha \leq 1 - \varepsilon$ for some positive constant ε, then $h \in O(\log n)$ and, moreover, $h_q \in O(\log \ell_q)$ for all q.

While the parsings induced by RLSLPs and block trees are naturally 1-contracting and 1/2-contracting, respectively, general parses are not even $o(n)$-contracting. We will be able, however, to provide efficient access to other α-contracting parses with α polynomial in the word size $w \geq \log n$, as long as the accessed characters have small height. This will turn out to be significantly more challenging.

▶ **Theorem 9.** *Let $S[1 \,..\, n]$ have an α-contracting parse of t phrases for any $\alpha \leq w^{O(1)}$. Then there exists a data structure of size $O(t)$ that can access any $S[q]$ in time $O(h_q + \log_w \ell_q)$.*

This result encompasses those of Section 3, as those are all 1-contracting parsings with $h_q = O(\log \ell_q)$. It also applies to other α-contracting parsings not based on grammars. For example, Kempa and Prezza [23, Thm 3.12] describe a bidirectional parse of size $t = O(\gamma \log(n/\gamma))$ where the sources only overlap phrases no longer than half the phrase's size, implying $\alpha = 1/2$ and $h_q = O(\log \ell_q)$, so they obtain $O(\log \ell_q)$ access time.

We begin the proof of Theorem 9 with a modified construction (based on a classic result due to Gilbert and Moore [20]) of Bille et al.'s [6] interval-biased search trees:

▶ **Lemma 10.** *Suppose we are given a partition of the interval $[1, n)$ into s subintervals*

$$[x_1 = 1, x_2), [x_2, x_3), \ldots, [x_{s-1}, x_s), [x_s, x_{s+1} = n)\,.$$

For any integer $d \geq 2$, we can build a d-ary tree T storing keys $x_1, \ldots, x_s$ (regarded as sequences of d-ary digits) at its leaves, in which each x_i has depth

$$\left\lfloor \log_d \frac{n}{x_{i+1} - x_i} \right\rfloor + 2\,.$$

Proof. For $1 \leq i \leq s$, let β_i be the string consisting of the first

$$\left\lfloor \log_d \frac{n}{x_{i+1} - x_i} \right\rfloor + 2$$

digits to the right of the base-d point in the base-d representation of $\frac{x_i + x_{i+1}}{dn}$.

Assume $j > i$ (the other case is symmetric). Since $x_{j+1} > x_j \geq x_{i+1}$, we have that $x_j + x_{j+1} > 2x_j \geq 2x_{i+1} = x_i + x_{i+1} + (x_{i+1} - x_i)$. Thus, $(x_j + x_{j+1})$ differs from $(x_i + x_{i+1})$ by more than $(x_{i+1} - x_i)$ for all $j \neq i$. Hence, $\frac{x_i + x_{i+1}}{dn}$ differs from $\frac{x_j + x_{j+1}}{dn}$ by more than $\frac{x_{i+1} - x_i}{dn}$, so their base-d representations agree on fewer than $\log_d \frac{dn}{x_{i+1} - x_i} = \log_d \frac{n}{x_{i+1} - x_i} + 1$ digits to the right of their base-d points. It follows that β_i is not a prefix of β_j.

We build the d-ary trie for $\beta_1, \ldots, \beta_s$, ordering nodes' children by the natural order induced by digits $[0, d - 1]$, store $x_1, \ldots, x_s$ at the leaves, and discard the edge labels. ◀

Next, we prove a lemma about the trie T used in our proof of Lemma 10, similar to Gagie et al.'s [14, Sec. 2] observation about the structure of parse trees of locally balanced SLPs:

▶ **Lemma 11.** *Given l and r with $1 \leq l < r \leq s$, there exist $d' \leq d$ nodes $v_1, \ldots, v_{d'}$ at depths at least $\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$ in T whose subtrees together contain the lth through rth leaves. Furthermore, the lowest common ancestor u of the lth and rth leaves is such that (i) $lca(v_a, v_b) = u$ for all $1 \leq a < b \leq d'$, and (ii) the path from u to v_a (both excluded) contains only nodes with out-degree equal to one for all $1 < a < d'$.*

Proof. Recall that u is defined to be the lowest common ancestor of the lth and rth leaves. Let $u^l \neq u^r$ be the children of u being ancestors of the lth and rth leaves, respectively. The integer $d' \leq d$ will be the number of siblings (children of u) between u^l and u^r, both included. See Figure 2 for an illustration.

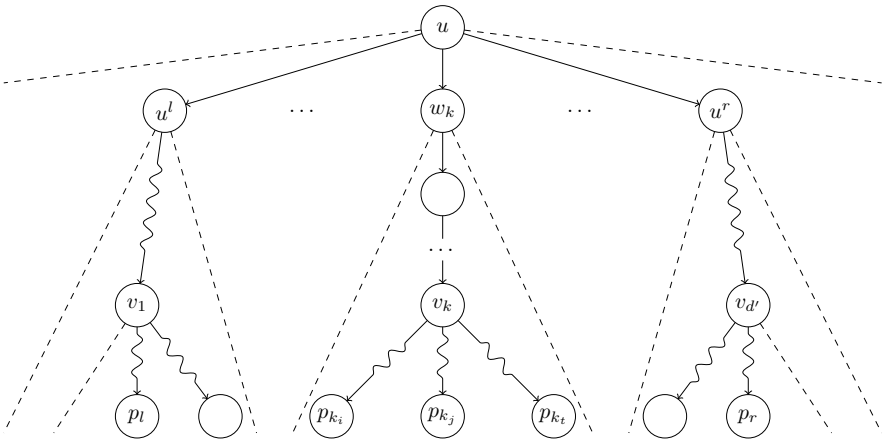

Figure 2 Subtrees of the trie T of Lemma 10 with roots $v_1, \ldots, v_{d'}$ at depth $\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$, containing the lth through rth leaves $p_l, \ldots, p_r$ of T (as explained in the proof of Lemma 11).

Finding v_1. We choose v_1 to be the lowest node on the path from u^l to the lth leaf such that v_1 is an ancestor of the rightmost leaf in the subtree rooted in u^l.

Letting v be a trie node and $c \in [0, d-1]$ be a base-d digit, let us denote with $v(c)$ the child of v by digit c. For the sake of a contradiction, assume v_1's depth is less than $\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$. Consider the path label β of v_1. Let y be the base-d digit corresponding to the child $v_1(y)$ of v_1 being an ancestor of the rightmost leaf in the subtree rooted in u^l. The numbers in $[0, 1)$ whose base-d representations start with $0.\beta y$ form a range R of size $|R| = d^{-(|\beta|+1)}$. Since above we assume $|\beta| < \log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$, the size of R satisfies $|R| = d^{-(|\beta|+1)} > \frac{x_r + x_{r+1} - x_l - x_{l+1}}{dn}$.

First, observe that R must end after $\frac{x_l + x_{l+1}}{dn}$: this follows from the definition of v_1, which implies that the path label of the lth leaf starts with $\beta y'$ for some $y' < y$. Additionally, since R cannot include $\frac{x_r + x_{r+1}}{dn}$ – otherwise v_1 would be an ancestor of the rth leaf, contrary to our choices of u and v_1 – it must start before

$$\frac{x_r + x_{r+1}}{dn} - \frac{x_r + x_{r+1} - x_l - x_{l+1}}{dn} = \frac{x_l + x_{l+1}}{dn}.$$

This means that R includes $\frac{x_l + x_{l+1}}{dn}$, which implies that the base-d representation of $\frac{x_l + x_{l+1}}{dn}$ starts with $0.\beta y$, so both the lth leaf and the rightmost leaf in the subtree rooted in u^l are in the subtree rooted in $v_1(y)$, contrary to our choice of v_1: a contradiction. Therefore, we conclude that v_1's depth is at least $\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$.

Finding $v_{d'}$. The choice of $v_{d'}$ is symmetric to that of v_1. Recall that u^r is the child of u being an ancestor of the rth leaf. We choose $v_{d'}$ to be the lowest node on the path from u^r to the rth leaf such that $v_{d'}$ is an ancestor of the leftmost leaf in the subtree rooted in u^r. A symmetric argument shows that $v_{d'}$'s depth is also at least $\log_d \frac{n}{x_r+x_{r+1}-x_l-x_{l+1}}$.

Finding $v_2, \ldots, v_{d'-1}$. If $d' > 2$, we are left to show how to find nodes $v_2, \ldots, v_{d'-1}$. Let w_k, for $k = 1, \ldots, d'$, denote the kth child of u starting from u^l included (that is, $u^l = w_1$, w_2 is the next sibling of w_1, and so on until $w_{d'} = u^r$). We define v_k to be the deepest descendant of w_k being an ancestor of all the leaves below w_k. In other words, v_k is the shallowest descendant of w_k with at least two children, or the unique leaf below w_k if no such descendant exists (note $v_k = w_k$ if and only if w_k has at least two children). By the definitions of u and $v_1, \ldots, v_{d'}$, it is clear that the subtrees below $v_1, \ldots, v_{d'}$ contain the lth through rth leaves. We are left to show that also nodes $v_2, \ldots, v_{d'-1}$ are at depths at least $\log_d \frac{n}{x_r+x_{r+1}-x_l-x_{l+1}}$. Consider any such node v_q, with $q \in [2, d-1]$. If v_q is a leaf, say, the tth leaf, then its depth is at least $\log_d \frac{n}{x_{t+1}-x_t}$, where $x_t \geq x_{l+1}$ and $x_{t+1} \leq x_r$. Then, the depth of v_q is at least $\log_d \frac{n}{x_{t+1}-x_t} \geq \log_d \frac{n}{x_r-x_{l+1}} \geq \log_d \frac{n}{x_r+x_{r+1}-x_l-x_{l+1}}$ (the latter inequality comes from $x_{r+1} - x_l > 0$).

If v_q is not a leaf, consider any two leaves below v_q – let us call them the l'th and r'th leaves (with $r' > l'$ without loss of generality). Since it must be $l < l' < r' < r$ and, hence, $x_l < x_{l+1} \leq x_{l'} < x_{l'+1} \leq x_{r'} < x_{r'+1} \leq x_r < x_{r+1}$, we have $\frac{x_{r'}+x_{r'+1}}{dn} - \frac{x_{l'}+x_{l'+1}}{dn} \leq \frac{x_r+x_{r+1}-x_l-x_{l+1}}{dn}$, meaning that the lowest common ancestor of the l'th and r'th leaves is at depth at least $\log_d \frac{dn}{x_r+x_{r+1}-x_l-x_{l+1}} > \log_d \frac{n}{x_r+x_{r+1}-x_l-x_{l+1}}$. Since this holds for all pairs of leaves below v_q and v_q is the deepest node being an ancestor of all those leaves, we obtain that v_q itself is at depth at least $\log_d \frac{n}{x_r+x_{r+1}-x_l-x_{l+1}}$. ◀

The next lemma follows from Lemmas 10 and 11. From now until we have finished proving Theorem 9, by $\log_d x$ we will really mean $\log_d x$ rather than $\max(1, \log_d x)$ (as in the rest of the paper).

▶ **Lemma 12.** *Suppose we are given the tree T of Lemma 10 for the partition*

$$[x_1 = 1, x_2), [x_2, x_3), \ldots, [x_{s-1}, x_s), [x_s, x_{s+1} = n),$$

and t arbitrary subintervals $[y_1, z_1) \ldots, [y_t, z_t)$ of $[1, n)$, with $n, x_1, \ldots, x_s, y_1, \ldots, y_t, z_1, \ldots, z_t$ all integers. Then, we can build an $O(t+s)$-space data structure on top of T with which, given i and an integer q in $[y_i, z_i)$, we can find j such that q is in $[x_j, x_{j+1})$ by traversing at most $\max\left(0, \log_d \frac{z_i-y_i}{x_{j+1}-x_j}\right) + O(1)$ edges of T and performing $O(\log_w d)$ additional constant-time operations, where w is the memory word size (in bits).

Proof. For each subinterval $[y_i, z_i)$, we store the index j' of the smallest $x_{j'} \geq y_i$ and the index j'' of the largest $x_{j''} < z_i$. This way, if $[x_j, x_{j+1})$ is not completely contained, or it is the only range contained, in $[y_i, z_i)$ then when given i and q we can find j with $O(1)$ integer comparisons and no accesses to T.

Otherwise, let $l < r$ be such that $[x_l, x_{l+1}), \ldots, [x_r, x_{r+1})$ are the ranges fully contained in $[y_i, z_i)$. We store, associated with every interval $[y_i, z_i)$, the three nodes $v_1, v_{d'}$, and u of T identified in Lemma 11 for the values l and r. We additionally store the two children v' and v'' of u that are ancestors of v_1 and $v_{d'}$, respectively (those were called u^l and u^r in the proof of Lemma 11). Further, for each node u of T with at least two children $u_1, \ldots, u_{d''}$ we build a fusion tree; for every such child u_p, the fusion tree stores the integer x_a such that the leftmost leaf below u_p is associated with interval $[x_a, x_{a+1})$. Because T has s leaves and the fusion trees are built on the nodes with at least two children, those fusion trees use $O(s)$ space in total.

Given an integer $q \in [y_i, z_i)$ and the node u associated with $[y_i, z_i)$, the fusion tree of u returns in $O(\log_w d)$ time the child u_p of u such that the jth leaf (i.e., the one such that $q \in [x_j, x_j+1)$) is below u_p. If u_p is equal to v' or v'', we replace it with v_1 or $v_{d'}$, respectively. Otherwise, we replace u_p with the deepest descendant of u_p being an ancestor of all the leaves below u_p; such substitution can be performed in constant time by explicitly associating such a node to each u_p in T (again, using $O(s)$ space overall). At this point, observe that u_p is precisely the node among $v_1, \ldots, v_{d'}$ identified by Lemma 11 such that (i) the jth leaf is below u_p and (ii) the depth of u_p in T is at least $\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}}$.

Since u_p has depth at least

$$\begin{aligned} &\log_d \frac{n}{x_r + x_{r+1} - x_l - x_{l+1}} \\ &\quad = \log_d \frac{n}{(x_r + x_{r+1})/2 - (x_l + x_{l+1})/2} - \log_d 2 \\ &\quad \geq \log_d \frac{n}{z_i - y_i} - 1 \end{aligned}$$

and x_j is at depth

$$\left\lfloor \log_d \frac{n}{x_{j+1} - x_j} \right\rfloor + 2,$$

we can find the jth leaf of T by starting from u_p, traversing at most

$$\log_d \frac{n}{x_{j+1} - x_j} - \log_d \frac{n}{z_i - y_i} + 3 \quad = \quad \log_d \frac{z_i - y_i}{x_{j+1} - x_j} + O(1)$$

edges of T. ◀

The last ingredient in our proof of Theorem 9 is the following result, proven in Appendix A, about linear-space distance-sensitive predecessor data structures. It improves the query time of Belazzougui et al.'s [2] and Ehrhardt's [13] from $O(\log\log\Delta)$ to $O(\log\log_w \Delta)$, where Δ is the the distance between the query and its predecessor.

▶ **Theorem 13.** *Let $X = \{x_1, \ldots, x_s\} \subseteq [1, n]$. We can build a data structure on X taking $O(s)$ words of space and answering predecessor queries on X in $O(\log\log_w \Delta)$ time, where $w \geq \log n$ is the machine word size in bits and Δ is the the distance between the query and its predecessor.*

Now we are finally in position to prove Theorem 9.

Proof of Theorem 9. Let $x_1, \ldots, x_t$ be the starting positions in S of the phrases, with $x_{t+1} = n + 1$. Further, let $[y_i, z_i)$ be the source of the phrase $[x_i, x_{i+1})$ (if the phrase is explicit, this interval is omitted). We build the tree T of Lemma 10 on the values x_i, and the additional data structure of Lemma 12 on top of T, with the intervals $[y_i, z_i)$. Further, we build a distance-sensitive predecessor data structure on the values $x_1, \ldots, x_t$ according to Theorem 13. The procedure to access $S[q]$ is then as follows:

1. Find the predecessor x_i of q with the data structure of Theorem 13.
2. If the phrase $[x_i, x_{i+1})$ is the explicit string B_i, return $B_i[q - x_i + 1]$.
3. Update $q \leftarrow q - x_i + y_i$, which belongs to $[y_i, z_i)$.
4. Given i and q, use Lemma 12 to find j such that q is in $[x_j, x_{j+1})$.
5. Update $i \leftarrow j$ and return to point 2.

The number of iterations of points 2–5 is at most h_q. By Lemma 12, the number of edges of T traversed in point 4 is at most $\max\left(0, \log_d \frac{z_i - y_i}{x_{j+1} - x_j}\right) + O(1) = \max\left(0, \log_d \frac{x_{i+1} - x_i}{x_{j+1} - x_j}\right) + O(1)$. Furthermore, Lemma 12 spends an additional time $O(\log_w d)$ on top of that. We now show that the sum of traversed edges telescopes along the iterations.

Let $x_{i_0} = x_i$ for x_i as obtained in point 1, and given $x_{i_k} = x_i$ in point 4, let $x_{i_{k+1}} = x_j$. Then the number of edges traversed in T along the h_q steps of the iteration is at most

$$O(h_q) + \sum_{k=0}^{h_q - 1} \max\left(0, \log_d \frac{x_{i_k+1} - x_{i_k}}{x_{i_{k+1}+1} - x_{i_{k+1}}}\right).$$

The sum would telescope, except that negative logarithms (i.e., when the source is longer than the phrase) introduce zeros in the sequence. Yet, since the parsing is α-contracting, it follows that $\log_d \frac{x_{i_k+1} - x_{i_k}}{x_{i_{k+1}+1} - x_{i_{k+1}}} \geq -\log_d \alpha$. We can then bound all terms $\max\left(0, \log_d \frac{x_{i_k+1} - x_{i_k}}{x_{i_{k+1}+1} - x_{i_{k+1}}}\right)$ $\leq \log_d \alpha + \log_d \frac{x_{i_k+1} - x_{i_k}}{x_{i_{k+1}+1} - x_{i_{k+1}}}$, and upper bound the summation by

$$O(h_q) + h_q \log_d \alpha + \sum_{k=0}^{h_q - 1} \log_d \frac{x_{i_k+1} - x_{i_k}}{x_{i_{k+1}+1} - x_{i_{k+1}}},$$

which now does telescope to $\log_d(x_{i_0+1} - x_{i_0}) + O(h_q(1 + \log_d \alpha))$.

Let $x_s = x_{i_0}$ and $l_q = x_{s+1} - x_s$ be the length of the block where q lies. We have just shown that the sum adds up to $\log_d(x_{s+1} - x_s) + O(h_q(1 + \log_d \alpha)) = O(h_q(1 + \log_d \alpha) + \log_d l_q)$. Each of those edge traversals can be done in constant time using perfect hashing to store the children of the nodes. The cost of point 1 is $O(\log\log_w(q - x_s)) \subseteq O(\log\log_w l_q)$. Taking into account the additional time $O(\log_w d)$ of each application of Lemma 12, the total cost becomes $O(h_q(1 + \log_d \alpha + \log_w d) + \log_d l_q + \log\log_w l_q)$. Choosing the arity of T as $d = w$ and remembering that we assume $\alpha \leq w^{O(1)}$, this cost becomes $O(h_q + \log_w l_q)$.

Because of the parsing invariants, a non-explicit phrase must occur elsewhere in S; therefore we can bound $l_q \leq \ell_q$. This proves the claimed time complexity.

Both the predecessor data structure and the extra structures of Lemma 12 use $O(t)$ space. The tree T, however, has t leaves but could have more internal nodes. To achieve the space bound of $O(t)$, we compact T to obtain T', replacing by a single edge every maximal path in T with nodes having exactly 1 child each. (Note T' is still a d-ary search tree storing on each of its edges the smallest integer x_i below it to guide search.) This process may eliminate the nodes v_1 or $v_{d'}$ of Lemma 11 but, if r' and r'' are their lowest ancestors in T with 2 or more children each, then we can still start from r' and r'' in T' and find the jth leaf of T' traversing the same amount of edges (or less). Note that, in T', we will not have the problem of handling unary paths we have discussed above. ◀

In future work we will explore how Theorem 9 can be applied. For example, we can prove the following lemma about converting bidirectional parses into somewhat larger contracting bidirectional parses.

▶ **Lemma 14.** *Let $S[1\,..\,n]$ have a bidirectional parse of size b, and denote with h_q the height of $S[q]$ in such a parse. For any integer $\alpha > 1$, we can build an α-contracting bidirectional parse of size $O(b \log_\alpha(n/b))$ for S such that the height h'_q of any character $S[q]$ in such a parse is no larger than that in the original parse: $h'_q \leq h_q$.*

Proof. First, we build the same string attractor Γ of [23] – of size $O(b)$ – on the input parse: for every phrase $S[i\,..\,j]$, we insert i and j in Γ. Then, we additionally insert in Γ integers 1, n, and $k \cdot \lceil n/b \rceil$ in Γ for every $k = 1, 2, \ldots b - 1$. As a result, Γ is a string attractor for S of

size $O(b)$ with the property that pairs of consecutive string attractor positions are within $O(n/b)$ positions from each other. Let $\Gamma = \{x_1 < \cdots < x_p\}$ be the string attractor obtained in this way.

We use a technique similar to the concentric exponential parsing [23, Thm 3.12] and build a bidirectional parse for S as follows. Repeat the following construction for every pair of consecutive positions $x_i < x_{i+1}$ in Γ. Let $m = (x_i + x_{i+1})/2$. We make $S[x_i]$, $S[x_{i+1}]$, and $S[m]$ explicit phrases. Then, we parse $S[x_i+1\,..\,m-1]$ left-to-right creating one (non-explicit) phrase of length α^1, followed by one (non-explicit) phrase of length α^2, and so on as long as the current phrase we are building is completely contained inside $S[x_i+1\,..\,m-1]$. We repeat a symmetric procedure parsing right-to-left the substring $S[m+1\,..\,x_{i+1}]$. These two procedures may leave a region spanning $S[m]$ not being covered by any phrase; we make that region a single phrase (observe that the length of this phrase is not necessarily a power of α). The above procedures create in total $O(b\log_\alpha(n/b))$ phrases.

Phrases' sources are chosen as follows. Observe that, by construction, each phrase in the parse we just built is strictly contained in a phrase of the original parse. Given a phrase $S[i\,..\,j]$ in the parse we just built, we follow its source $S[i'\,..\,j'] = S[i\,..\,j]$ in the original parse. If $S[i'\,..\,j']$ spans an element of Γ then we stop; otherwise, we recursively repeat following sources until this becomes true (by definition of bidirectional parse, this procedure cannot loop forever). Let $S[i''\,..\,j''] = S[i\,..\,j]$ be the substring (containing an element of Γ) found in this way. We assign $S[i''\,..\,j'']$ as source of $S[i\,..\,j]$. Observe that this source assignment is such that: (1) the height h'_q of any $S[q]$ in the new parse is no larger than that of the original parse: $h'_q \le h_q$, and (2) the resulting parse is indeed a valid bidirectional parse (i.e., no loops of referencing chains are introduced). Both properties follow from the fact that we possibly just "short-cut" referencing chains of the original parse, making them (possibly) shorter.

We now show that the source of every non-explicit phrase of length in $[\alpha^t, \alpha^{t+1})$ overlaps only phrases of length at most α^{t+1}, implying that the parse is α-contracting. To see this, let $S[i\,..\,i+d-1]$ be such a source ($d \in [\alpha^t, \alpha^{t+1})$), and let $j \in [i\,..\,i+d-1]\cap\Gamma$ be any attractor element crossing it. Consider the source's suffix $S[j\,..\,i+d-1]$ (a symmetric argument holds for the prefix $S[i\,..\,j-1]$), and let $\lambda = i+d-j \in [\alpha^t, \alpha^{t+1})$ be its length. By construction of our parse, the leftmost phrase of length at least α^{t+2} in $S[j\,..\,n]$ cannot start before $S[j+k]$, where $k = \sum_{z=0}^{t+1}\alpha^z > \alpha^{t+1}$. We conclude that $S[j\,..\,i+d-1]$ ends before $S[j+k]$, hence it can only overlap phrases of length at most α^{t+1}. Our claim follows. ◀

Setting $\alpha = w$ in Lemma 14, where w is the word size, and combining with Theorem 9, we obtain the following result.

▶ **Corollary 15.** *Let $S[1\,..\,n]$ have a bidirectional parse of size b. On a machine word of word size $w \ge \log n$ bits, we can store a data structure of $O(b\log_w(n/b))$ words supporting the extraction of any $S[q]$ in time $O(h_q + \log_w \ell_q)$, where ℓ_q is the length of the longest repeated substring of S containing position q and h_q the height of $S[q]$ in the parse.*

5 Conclusions

We have shown how access to highly-compressed strings $S[1\,..\,n]$ can be sped up at their less compressible regions, which are arguably the most interesting ones in applications (e.g., mutations or rare variants in genome collections, edits in versioned collections). Concretely, we chose ℓ_q, the length of the longest repeated substring containing $S[q]$, as a measure of how compressible the area of S surrounding position q is. We proved that arbitrary run-length

context-free grammars, as well as block trees, can be modified so as to access $S[q]$ in time $O(\log \ell_q)$ without asymptotically increasing their size. We extended this result to more general parses, proving that under some conditions on the dependency structure of the parse, one could access $S[q]$ in time $O(h_q + \log_w \ell_q)$, where h_q is the length of the referencing chain to obtain $S[q]$ through the parse and w is the size of the RAM machine word in bits. We showed how, for example, bidirectional macro schemes (the smallest known parses for highly compressible strings) of size b can be made to satisfy the needed condition by blowing up their size by a factor of $O(\log_w(n/b))$.

Several interesting questions remain open. A first one is whether we could balance grammars, while retaining their asymptotic size, so that the height of the parse tree of any nonterminal A is not just $O(\log|\exp(A)|)$ but $O(\log_w|\exp(A)|)$, where $\exp(A)$ is the string A expands to. This would enable accessing them in time $O(\log_w \ell_q)$ (which still does not break lower bounds [36]), closely matching our results on more general parses. Another challenge would be to make bidirectional macro schemes retain size $O(b)$, without any asymptotic blowup, and make them accessible in time $O(h_q + \log_w \ell_q)$. A very recent development [33] defines measure s, where $\gamma \leq s \leq b$ is the size of the smallest Substring Equation System (SES) whose unique solution is S. While SES are in principle harder to decompress than bidirectional macro schemes, could we access $S[q]$ in time $O(h_q \log_w \ell_q)$ within space $O(s \log_w(n/s))$, for example? Finally, are there other alternatives to measure ℓ_q that better describe the less compressible areas of S? Are there other features that make some areas of highly compressible strings S easier to access than others?

References

1 Hideo Bannai, Mitsuru Funakoshi, Diptarama Hendrian, Myuji Matsuda, and Simon J. Puglisi. Height-bounded Lempel-Ziv encodings. In *Proc. 32nd Annual European Symposium on Algorithms (ESA)*, pages 18:1–18:18, 2024. doi:10.4230/LIPIcs.ESA.2024.18.

2 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Predecessor search with distance-sensitive query time. *CoRR*, abs/1209.5441, 2012. doi:10.48550/arXiv.1209.5441.

3 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez, Simon J Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

4 Djamal Belazzougui, Patrick Hagge Cording, Simon J Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. European Symposium on Algorithms (ESA)*, pages 142–154, 2015. doi:10.1007/978-3-662-48350-3_13.

5 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *Theory of Computing Systems*, 62(8):1715–1735, 2018. doi:10.1007/s00224-017-9839-9.

6 Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015. doi:10.1137/130936889.

7 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

8 Anders R. Christiansen, Mikko B. Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):article 8, 2020.

9 Ferdinando Cicalese and Francesca Ugazio. On the complexity and approximability of bounded access Lempel Ziv coding. In *Proc. Conference on Developments in Language Theory (DLT)*, pages 114–130, 2024. doi:10.1007/978-3-031-66159-4_9.

10 Ferdinando Cicalese and Francesca Ugazio. Hardness and approximability of bounded access Lempel Ziv coding. *Information and Computation*, 307:article 105366, 2025.

11 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 180–192, 2012. doi:10.1007/978-3-642-34109-0_19.

12 Patrick Hagge Cording, Pawel Gawrychowski, and Oren Weimann. Bookmarks in grammar-compressed strings. In *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 153–159, 2016. doi:10.1007/978-3-319-46049-9_15.

13 Marcel Ehrhardt and Wolfgang Mulzer. Delta-fast tries: Local searches in bounded universes with linear space. In *Proc. Workshop on Algorithms and Data Structures (WADS)*, pages 361–372, 2017. doi:10.1007/978-3-319-62127-2_31.

14 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. A faster grammar-based self-index. In *Proc. Conference on Language and Automata Theory and Applications (LATA)*, pages 240–251, 2012.

15 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. Latin American Symposium on Theoretical Informatics (LATIN)*, pages 731–742, 2014.

16 Moses Ganardi. Compression by Contracting Straight-Line Programs. In *Proc. 29th Annual European Symposium on Algorithms (ESA)*, pages 45:1–45:16, 2021. doi:10.4230/LIPIcs.ESA.2021.45.

17 Moses Ganardi, Artur Jeż, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):1–40, 2021. doi:10.1145/3457389.

18 Leszek Gasieniec, Roman M Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Proc. Data Compression Conference (DCC)*, page 458, 2005. doi:10.1109/DCC.2005.78.

19 Leszek Gasieniec and Igor Potapov. Time/space efficient compressed pattern matching. *Fundamenta Informaticae*, 56(1-2):137–154, 2003. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-09.

20 Edgar N. Gilbert and Edward F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

21 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Communications of the ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.

22 Dominik Kempa and Tomasz Kociumaka. Tight lower bounds for central string queries in compressed space. In Kasper Green Larsen and Barna Saha, editors, *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1824–1840, 2026. doi:10.1137/1.9781611978971.65.

23 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. Symposium on Theory of Computing (FOCS)*, pages 827–840, 2018. doi:10.1145/3188745.3188814.

24 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-End parsing. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2847–2866, 2022. doi:10.1137/1.9781611977073.111.

25 Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86:1031–1056, 2024. doi:10.1007/s00453-023-01186-0.

26 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2022. doi:10.1109/TIT.2022.3224382.

27 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.

28 Zsuzsanna Lipták, Francesco Masillo, and Gonzalo Navarro. BAT-LZ out of hell. In *Proc. 35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 21:1–21:17, 2024. doi:10.4230/LIPIcs.CPM.2024.21.

29 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.

30 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

31 Gonzalo Navarro, Francisco Olivares, and Cristian Urbina. Balancing run-length straight-line programs. In *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 117–131, 2022. doi:10.1007/978-3-031-20643-6_9.

32 Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

33 Hiroki Shibata and Hideo Bannai. String representation in suffixient set size space. *CoRR*, 2604.04377, 2026. doi:10.48550/arXiv.2604.04377.

34 Hiroki Shibata, Yuto Nakashima, Yutaro Yamaguchi, and Shunsuke Inenaga. LZBE: an LZ-style compressor supporting $O(\log n)$-time random access. In *Proc. 37th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 34:1–34:18, 2026. doi:10.4230/LIPIcs.CPM.2026.34.

35 Jim A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

36 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013. doi:10.1007/978-3-642-38905-4_24.

A Proof of Theorem 13

▶ **Definition 16.** *Let $X = \{x_1, \ldots, x_s\} \subseteq [1, n]$ and let $x_0 = 0$. For any $q \in [1, n]$, the* predecessor *q^- of q in $X \cup \{0\}$ is defined as $q^- = \max\{x_i \ : \ i \in [0, s] \ \wedge \ x_i \leq q\}$. The* strict predecessor *of q in $X \cup \{0\}$ is $\max\{x_i \ : \ i \in [0, s] \ \wedge \ x_i < q\}$. We denote as $\Delta = q - q^-$ the distance to q from its predecessor. A predecessor query on X finds such $q^- \in X \cup \{0\}$ given q as input.*

Intuitively, to prove Theorem 13 we will build a w-ary z-fast trie (that is, a z-fast trie of arity w endowed with a fusion tree on each explicit node supporting constant-time predecessor queries on the outgoing edges of the node) on the base-w representations of the integers $x_1, \ldots, x_s$. Classic *fat binary search* on this trie yields predecessor queries in time $O(\log \log_w n)$. In order to speed this up to $O(\log \log_w \Delta)$, we generalize the exponential search of [2, 13] (designed on binary z-fast tries) to w-ary z-fast tries. The exponential search phase will precede classic fat binary search and will find, in $k \in O(\log \log_w \Delta)$ steps, a trie node at height (distance from the leaves) $2^k \in O(\log_w \Delta)$, hence fat binary search from that node will find the predecessor of q in $O(\log 2^k) \subseteq O(\log \log_w \Delta)$ time.

In the following, for convenience we switch to base w when dealing with integers (this will avoid an excessive use of ceiling operations and will simplify notation), where w is the machine word size in bits. More precisely, we work with integers from $[0, w^H)$ (H digits in base w), for the smallest integer H being a power of two and satisfying $H \geq \log_w 2^w$. Observe that any $q \in [0, w^H)$ still fits in $O(1)$ memory words, so this assumption is without loss of generality. We will moreover use string notation on the integers from $[0, w^H)$ (as well as standard integer notation). Given $q \in [0, w^H)$ and $i \in [1, H]$, $q[i]$ denotes the i-th most significant digit of q written in base w. In other words, q is treated as a string of length H over alphabet $[0, w)$.

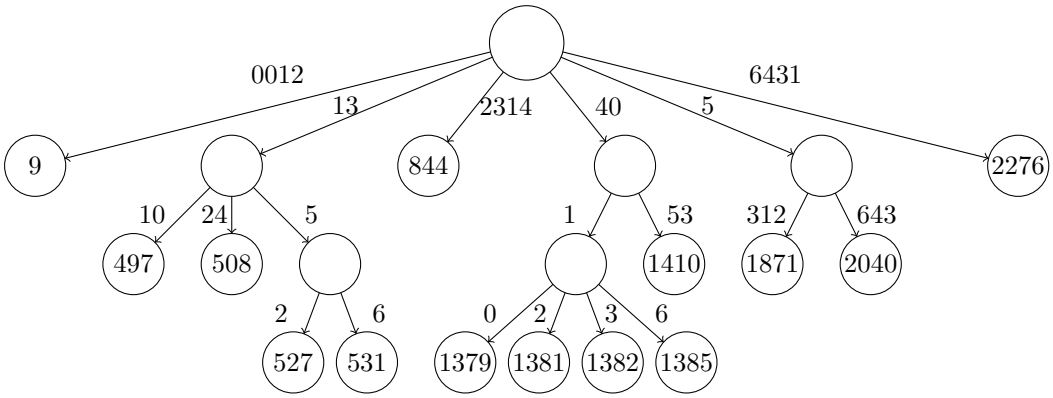

Figure 3 Example of the compacted trie Z with word size $w = 7$ and $H = 4 \geq \log_7 2^7 \approx 2.48$ for the set $X = \{9, 497, 508, 527, 531, 844, 1379, 1381, 1382, 1385, 1410, 1871, 2040, 2276\}$. In the leaf nodes there are the values of X. The concatenation of the labels from the root to a leaf node is the representation in base w of the value in that node.

Given $X = \{x_1, \ldots, x_s\} \subseteq [1, w^H)$, we denote by $pref(X)$ the set containing all prefixes of strings in X.

Consider the w-ary trie containing $x_1, \ldots, x_s$ (seen as strings of length H over alphabet $[0, w)$). Letting $\lambda(u) \in pref(X)$ denote the label from the root to trie node u, u is *explicit* if and only if either $|\lambda(u)| = H$ or if $\lambda(u) \cdot a$ and $\lambda(u) \cdot b$ belong to $pref(X)$ for two distinct $a, b \in [0, w)$. We path-compress this trie and denote by Z the resulting trie; nodes of Z correspond to the explicit nodes of the original trie and edges of Z are labeled with sequences of digits from $[0, w)$.

In the following, for brevity, when saying that edge e (of trie Z) is labeled $x_i[l, r]$, we also mean that the string read from the root of Z to the end of e is $x_i[1, r]$.

We identify one *canonical position* $f(l, r)$ inside each edge labeled $x_i[l, r]$ of the path-compressed trie Z, as follows:

▶ **Definition 17.** *The* 2-fattest number $f(l, r)$ *in a given interval* $[l, r]$ *(*$1 \leq l \leq r \leq w$*) is the number* $f(l, r) \in [l, r]$ *whose binary representation has the largest number of trailing zeros among all numbers in* $[l, r]$*.*

It is not hard to see that the 2-fattest number in any given interval is unique, because for any pair of integers $i < j$ whose binary representation has k trailing zeros, there always exists a third integer z with $i < z < j$ whose binary representation has $k + 1$ zeros. Intuitively (as we will see later), this particular choice of canonical position has the additional advantage of enabling a binary search strategy on the trie's height.

Based on the above definition, we define the *handle* of an edge as follows.

▶ **Definition 18** (handle)**.** *Let* e *be a trie edge labeled with* $x_i[l, r]$*. The* handle h_e *of* e *is the string* $h_e = x_i[1, f(l, r)]$*.*

The next ingredient towards defining our data structure is a map storing information about children of nodes in Z.

▶ **Definition 19.** *Let* e *be a trie edge labeled with* $x_i[l, r]$*. The (partial) function* $\mathcal{C}_e : [0, w) \to [0, w)^*$ *stores, for each* $c \in [0, w)$ *such that* $x_i[1, r] \cdot c \in pref(X)$*, the string* $\mathcal{C}_e(c) = x_{i'}[1, r']$ *prefixed by* $x_i[1, r] \cdot c$ *such that there exists a trie edge labeled* $x_{i'}[r + 1, r']$ *for some* i', r'*. We denote with* $\mathcal{C}_\varepsilon$ *the additional special case where* $x_i[1, r = 0] = \varepsilon$ *is the empty string.*

In other words, the domain of $\mathcal{C}_e$ corresponds to the characters that can extend edge e in the trie. For each such character c, the map $\mathcal{C}_e$ stores the label $\mathcal{C}_e(c)$ of the path starting in the trie's root and ending at the end of edge e' that is a child of edge e and whose label starts with c. The particular case $\mathcal{C}_\varepsilon$ corresponds to the trie's root: $\mathcal{C}_\varepsilon(c)$ is the label of the edge exiting the root and whose label starts with character c.

Finally, our structure will associate the following additional information with the trie's edges.

▶ **Definition 20.** *Let $x_i[l,r]$ be the label of an edge of Z. Then:*

- *$x^-_{i,r}$ is the strict predecessor in $X \cup \{0\}$ of the smallest $y \in X$ being prefixed by $x_i[1,r]$;*
- *$x^+_{i,r} : [0,w) \cup \{\varepsilon\} \to X$ is the (partial) function associating with every $c \in [0,w) \cup \{\varepsilon\}$ the largest number $x^+_{i,r}(c) \in X$ that is prefixed by $x_i[1,r] \cdot c$ (if no such number exists, then the map is not defined on c).*

The w-ary z-fast trie of $x_1, \ldots, x_s$ is a function $\mathcal{Z}$ defined as follows.

▶ **Definition 21** (w-ary z-fast trie)**.** *The w-ary z-fast trie of $x_1, \ldots, x_s$ is the (partial) function $\mathcal{Z}$ defined as follows. For every edge e of Z labeled with $x_i[l,r]$ (for some i,l,r), we define $\mathcal{Z}(h_e) = (r, \mathcal{C}_e, x^-_{i,r}, x^+_{i,r})$, where $h_e = x_i[1, f(l,r)]$ is the handle of e.*

Note that $x_i[1,r] = x^+_{i,r}(\varepsilon)[1,r]$, so below we take for granted that such a string can be retrieved in constant time from $\mathcal{Z}(h_e)$.

We implement $\mathcal{Z}$ using perfect hashing. Observe that membership in the domain of $\mathcal{Z}$ can be determined in $O(1)$ time since $\mathcal{Z}(h_e)$ can be used to obtain $x_i[1,r]$, which is an extension of h_e. Functions $\mathcal{C}_e$ and $x^+_{i,r}$ are returned in $\mathcal{Z}(h_e)$ as pointers (so retrieving $\mathcal{Z}(h_e)$ takes constant time), and are implemented using a fusion tree with constant-time predecessor queries over their domains.

Function $\mathcal{Z}$ is defined on $O(s)$ distinct arguments h_e (one for each trie's edge), hence integers r and $x^-_{i,r}$ returned in $\mathcal{Z}(h_e) = (r, \mathcal{C}_e, x^-_{i,r}, x^+_{i,r})$ use overall $O(s)$ words of space. Similarly, $\mathcal{C}_e$ is defined on n_e distinct arguments, where n_e is the number of one-character right-extensions of $x_i[1,r]$ ($x_i[l,r]$ being the label of edge e) yielding an element of $pref(X)$; for each such argument c, $\mathcal{C}_e(c) \in [0,w)^{\leq H}$ and therefore it can be packed in $O(1)$ machine words. Being also those right-extensions in bijection with the trie's edges, we conclude that all those functions $\mathcal{C}_e$ use $O(s)$ space overall. The same reasoning holds for functions $x^+_{i,j}$. Overall, the w-ary z-fast trie for X uses $O(s)$ words of space.

The structure of Definition 21 can be used to solve predecessor queries in $O(\log\log_w u)$ time ($u = w^H$). In the following subsection we prove a more general result that will allow us achieving distance-sensitive time $O(\log\log_w \Delta)$.

A.1 Extending to obtain distance-sensitive times

In order to support distance-sensitive predecessor queries, we augment the z-fast trie of Definition 21 with one additional data structure.

▶ **Definition 22.** *Given $q \in [0, w^H)$ and $k \in [0, \log H]$, we denote by q_k the string $q[1,H]$ devoid of its suffix of length $2^k - 1$, that is, $q_k = q[1, H - 2^k + 1]$.*

Note that, in Definition 22, it holds $k \in O(\log H) = O(\log\log_w u)$, where $u = w^H$ is the universe size, and q_k is a prefix of q that gets shorter as k increases. In particular, $q_0 = q$ and $q_{\log H} = q[1]$. As said above, q_k will be treated equivalently as an integer formed by the $H - 2^k + 1$ most significant digits of q in base w (this will allow us to perform integer subtractions on it).

▶ **Definition 23.** *Let $\mathcal{H} : [0, w)^* \to [0, w)^*$ be the (partial) function defined as follows. For each $k \in [0, \log H]$ and each $q \in X$, $\mathcal{H}(q_k) = |h_e|$ is the length of the longest handle h_e that is a prefix of q_k. If no such edge e exists, then $\mathcal{H}(q_k) = 0$.*

We store $\mathcal{H}$ with perfect hashing. Observe that k in Definition 23 satisfies $k \in O(\log H) = O(\log \log_w 2^w) \subseteq O(\log w)$, therefore $\mathcal{H}$ is defined on $O(s \log w)$ distinct arguments. Similarly, $\mathcal{H}(q_k) = |h_e| \leq H \in O(\log_w 2^w)$, therefore it can be packed in $O(\log w)$ bits. We conclude that $\mathcal{H}$ can be stored in $O(s(\log w)^2) \subseteq O(sw)$ bits, or $O(s)$ words.

▶ **Lemma 24.** *Functions $\mathcal{Z}$ and $\mathcal{H}$ take $O(s)$ words of space and allow determining in $O(1)$ time whether $p \in pref(X)$, for any $p \in [0, w)^{H-2^k+1}$ and $k \in [0, \log H]$.*

Proof. Let $p \in [0, w)^{H-2^k+1}$, with $k \in [0, \log H]$. We consider two cases.

(1) $\mathcal{H}(p) = 0$. If $p \in pref(X)$, then p must prefix the label $x_i[1, r]$ of one of the edges exiting from the trie's root, for some i, r. To check that this is the case, we get the label $x_i[1, r] = \mathcal{C}_\varepsilon(p[1])$ of that (candidate) edge. If $|p| > r$ then $p \notin pref(X)$. Otherwise, $p \in pref(X)$ if and only if $x_i[1, |p|] = p$.

(2) $\mathcal{H}(p) > 0$. We know the length $|h_e| = \mathcal{H}(p)$ of the handle of some edge e labeled with $x_i[l, r]$ maximizing r and such that h_e prefixes p – up to collisions in the perfect hash representing $\mathcal{H}$, which we now show how to detect. If $|h_e| > |p|$ then this is indeed a hash collision and $p \notin pref(X)$. Otherwise, we compute $h_e = p[1, |h_e|]$ in constant time and get $\mathcal{Z}(h_e) = (r, \mathcal{C}_e, x^-_{i,r}, x^+_{i,r})$, also in constant time. If $|p| \leq r$, then we check in constant time that $p = x_i[1, |p|]$ (recall that $\mathcal{Z}(h_e)$ allows us retrieving $x_i[1, r]$); this test succeeds if and only if $p \in pref(X)$. Otherwise ($|p| > r$), let $c = p[r + 1]$. We compute in constant time $\mathcal{C}_e(c) = x_{i'}[1, r']$, which at this point must be an extension of p if p does indeed belong to $pref(X)$. We check in constant time that $p = x_{i'}[1, |p|]$; this test succeeds if and only if $p \in pref(X)$. ◀

Running Lemma 24 for $k = 0, 1, 2, \ldots$ corresponds to running an exponential search for the longest prefix of q matching an element in $pref(X)$, obtaining the following result. Searching also for $q_k - 1$ will be needed later to solve predecessor queries.

▶ **Corollary 25** (exponential search)**.** *Let $X = \{x_1, \ldots, x_s\} \subseteq [1, w^H)$. We can build a data structure on X taking $O(s)$ words of space that, given any $q \in [1, w^H)$, returns the pair (k, q') defined as follows. Letting $P_k = \{q_k, q_k - 1\} \cap (pref(X) \cup \{0\})$, k is the smallest integer from $[0, \log H]$ such that $P_k \neq \emptyset$, and $q' = \max P_k$. If no such integer k exists, it returns $(-1, -1)$. This query is answered in $O(1 + k)$ time.*

In Corollary 25, note that we allow q_k (or q_{k-1}) to prefix also 0. This means that we stop looking for the smallest k even if q_k (or q_{k-1}) prefixes 0 but no other element in X. Additionally, observe that it is implicit that if $q_k = 0$, then $q_k - 1 = -1$ does not belong to $pref(X) \cup \{0\}$ (this avoids treating that particular case separately). Observe that Corollary 25 finds k by linear search on k (that is, $k = 0, 1, 2, \ldots$). This is enough to reach the desired running times for predecessor queries, as we will prove that $k \in O(\log \log_w \Delta)$.

▶ **Lemma 26** (w-ary z-fast trie with hints)**.** *Let $X = \{x_1, \ldots, x_s\} \subseteq [1, w^H)$. There exists a data structure on X taking $O(s)$ words of space supporting the following query. Given $q \in [1, w^H)$ and the smallest $k \in [0, \log H]$ (we call k the* hint*) such that $q_k \in pref(X)$, find the predecessor q^- of q in $X \cup \{0\}$. This query is solved in $O(1 + k)$ time.*

Proof. We know that there exists an integer $q' \in X$ prefixed by $q_k = q[1, H - 2^k + 1]$. If $k = 0$ then $q = q_k \in X$, hence we just return q and we are done, so in the following we assume $k > 0$.

We show how to use classic fat binary search on the z-fast trie to find the longest handle of length in $[H-2^k+2, H-1]$ that is a prefix of q (note that we can exclude length H since $k>0$) in the claimed running time. A first observation is that (since H is excluded from that interval) the binary representation of any number in $[H-2^k+2, H-1]$ has no more than k trailing zeros. This enables finding the answer by $O(1+k)$ steps of fat binary search as follows. We check, using map $\mathcal{Z}$, whether $q[1, f(H-2^k+2, H-1)]$ is a handle. If the answer is positive, then we recurse in interval $[f(H-2^k+2, H-1)+1, H-1]$. Otherwise, we recurse in interval $[H-2^k+2, f(H-2^k+2, H-1)-1]$. Since each recursive step removes the 2-fattest number from the interval, it also decreases the number of trailing zeros of the 2-fattest number in the interval by one unit. We conclude that this procedure finds the longest handle y prefixing q in $O(1+k)$ time.

Let $\ell = lcp(q, X) \in [0, w)^{<H}$ denote the longest prefix of q belonging to $pref(X)$. From the handle y found in the previous step, using map $\mathcal{Z}$ one can get in constant time (1) the label $x_i[1, r]$ ending at the end of trie edge e labeled with $x_i[l, r]$ (for some i, l, r) such that $l \leq |\ell| \leq r$ and $\ell = x_i[1, |\ell|]$, as well as (2) dictionary $\mathcal{C}_e$, (3) value $x^-_{i,r}$ and (4) function $x^+_{i,r}$. In other words, by reading ℓ from the trie's root we end up inside edge e. We distinguish three cases.

(1) If $x_i[1, r] < q[1, r]$, then q^- is the largest number in X being prefixed by $x_i[1, r]$, that is, $x^+_{i,r}(\varepsilon)$.

(2) If $x_i[1, r] > q[1, r]$, then $q^- = x^-_{i,r}$.

(3) Finally, it could be $x_i[1, r] = q[1, r]$, meaning that ℓ matches until the end of edge e. In that case, let $c = q[r+1]$ be the mismatching character (note that c is well-defined since $r = |\ell| < H$). We find the predecessor c^- of c among the characters in the domain of $\mathcal{C}_e$, that is, characters extending edge $e = x_i[l, r]$ in the trie. This operation takes constant time, since $\mathcal{C}_e$ is represented with a fusion tree. If such c^- exists, then $q^- = x^+_{i,r}(c^-)$. Otherwise, c is smaller than all those characters and q^- is again $x^-_{i,r}$ as in case (2). ◀

A.2 Putting everything together

Given $q \in [1, w^H)$, we apply Corollary 25 and find, in time $O(1+k)$, the smallest value k such that either q_k or $q_k - 1$ prefixes some integer in $X \cup \{0\}$, while (if $k > 0$) both q_{k-1} and $q_{k-1} - 1$ do not. We prove that such a value k is distance-sensitive.

▶ **Lemma 27.** *Let k be the value identified by Corollary 25. Then, $k \in O(\log\log_w \Delta)$.*

Proof. If $k = 0$ then the claim is trivially true, so we assume $k > 0$. Observe that by the definition of k it must be $P_{k-1} = \{q_{k-1}, q_{k-1} - 1\} \cap (pref(X) \cup \{0\}) = \emptyset$, hence in particular $q_{k-1} > 0$ and $q_{k-1} - 1 \neq 0$ hold. This allows us to conclude that $q_{k-1} - 1 > 0$.

Integers prefixed by either q_{k-1} or $q_{k-1} - 1$ form a contiguous integer range $R = R_1 \cup R_2$ where $R_1 = [(q_{k-1}-1)\cdot w^{2^{k-1}-1}, q_{k-1}\cdot w^{2^{k-1}-1})$ and $R_2 = [q_{k-1}\cdot w^{2^{k-1}-1}, (q_{k-1}+1)\cdot w^{2^{k-1}-1})$ are disjoint: $R_1 \cap R_2 = \emptyset$. Since no element in $X \cup \{0\}$ is prefixed by neither q_{k-1} nor $q_{k-1} - 1$, we have that (A) $R \cap (X \cup \{0\}) = \emptyset$. Moreover, since q is prefixed by q_{k-1}, then (B) $q \in R_2$. Observations (A) and (B) imply that q^- comes strictly before the left bound of R_1, that is, $q^- < (q_{k-1} - 1) \cdot w^{2^{k-1}-1}$. Then, $\Delta = q - q^- > |R_1| = w^{2^{k-1}-1}$. This implies $k < 1 + \log(1 + \log_w \Delta) \in O(\log\log_w \Delta)$. ◀

Let k be the value identified by Corollary 25. We distinguish two cases.

(1) If $q_k - 1$ prefixes some integer in $X \cup \{0\}$ but q_k does not (in particular, $q_k - 1 \geq 0$), then q^- is the largest integer in $X \cup \{0\}$ being prefixed by $q_k - 1$. We find such a value as follows. We check if $q_k - 1 \in pref(X)$ using Lemma 24. If $q_k - 1$ does not prefix any integer

in X, then $q^- = 0$ and we are done. Otherwise, using maps $\mathcal{H}$ and $\mathcal{Z}$ we identify in constant time the edge e labeled with $x_i[l, r]$ such that $x_i[1, r]$ is prefixed by $q_k - 1$ but $x_i[1, l-1]$ is not, as well as its associated map $x^+_{i,r}$. Then, $q^- = x^+_{i,r}(\varepsilon)$.

(2) Otherwise, q_k prefixes some integer in $X \cup \{0\}$. If $q_k \notin \mathit{pref}(X)$ (a property we can check using Lemma 24), then $q^- = 0$. Otherwise, we can apply Lemma 26 on query q with hint k and find q^- in $O(1 + k)$ time.

Overall, finding q^- takes $O(1 + k)$ time. By Lemma 27, this is $O(1 + k) = O(\log \log_w \Delta)$. This proves Theorem 13.