Optimal-Time Contextual Pattern Matching in Compressed Space*

Gonzalo Navarro[1,2][0000−0002−2286−741X] and Francisco Olivares[1,2][0000−0001−7881−9794]

[1] Centre for Biotechnology and Bioengineering, Chile
[2] Department of Computer Science, University of Chile, Chile
{gnavarro,folivares}@uchile.cl

Abstract. Contextual pattern matching is the task of, given a pattern $P[1,m]$, a context length λ, and a text $T[1,n]$, find all the *occ* distinct contexts in which P occurs in T, the context being the λ symbols preceding and the λ symbols following the occurrence; a text position where each context occurs must be output. While the problem can be solved in optimal time $O(m+occ)$ using $O(n)$-space precomputed data structures on T, this type of search is particularly relevant on large repetitive text collections, where $O(n)$ space can be prohibitive. We present the first optimal-time solution that runs in compressed space, namely that of a symmetric CDAWG (SCDAWG) of T. Further, we show how the set of *occ* solutions can be enumerated with $O(\log\log\lambda)$ delay after $O(m)$-time preprocessing of P. To achieve this, we develop an improved linear-space distance-sensitive weighted ancestor data structure.

Keywords: Compressed text indexing · CDAWGs · Contextual Pattern Matching · Weighted ancestors in trees

1 Introduction

Given a text $T[1,n]$ and a pattern $P[1,m]$, the classical Pattern Matching problem consists of finding all of the positions of T where P occurs. The Contextual Pattern Matching problem is a recently proposed variant [18] better suited for searching large collections of similar documents (e.g., genome collections, versioned documents, etc.). In such a case, if P occurs inside a piece of text that is repeated many times along the collection, it may be wasteful to list all those occurrences individually, as these are, for many purposes, essentially the same occurrence. We might want to list, for example, all the different contexts in which some DNA motif occurs in a large genomic collection: while most occurrences can appear within identical contexts, different contexts may indicate genetic variants or diseases. As another example, we might want to track the changes on what is said about a proper name is mentioned along the different versions of

* Funded by Basal Funds FB0001 and AFB240001, and Fondecyt Grant 1260080, ANID, Chile.

a Wikipedia page by listing the different snippets around its occurrences, which are likely to be identical across many versions.

In Contextual Pattern Matching, the query provides also a *context* length λ and asks for one occurrence position of each distinct context around P: the context is formed by the λ characters preceding and the λ characters following the occurrence. This is formally stated as follows.

Definition 1 ([18]). *The Contextual Pattern Matching problem on a text $T[1,n]$ is, given a pair $(P[1,m],\lambda)$, return a position in T for each of the occ distinct strings XPY occurring in T, for all X, Y such that $|X| = |Y| = \lambda$. To capture the occurrences near the extremes of T, assume T is preceded and followed by λ copies of the special symbol \$, which does not appear in P.*

In that paper [18], it was shown that the problem can be solved in optimal time, $O(m + occ)$, by using $O(n)$-size data structures (essentially, suffix trees). This is not so satisfactory, however, because many repetitive collections are too large for $O(n)$ space to be affordable. Instead, one desires some *compressed indexing* solution, which provides search functionality within space bounded by some measure of compressibility [20,17].

In the same paper (later corrected and improved [19]) they provide one such compressed indexing solution: Letting r be the number of runs in the Burrows-Wheeler Transform of T, and $\overline{r}$ the sum of the values of r for T and its reverse, the problem can be solved in time $O(m + occ \log n)$ with an index of size $O(\overline{r} \log \frac{n}{\overline{r}})$. Later, Abedin et al. [1] obtained $O(m + occ \log \lambda \log \frac{n}{r})$ time and $O(r \log \frac{n}{r})$ space. Using bidirectional indexes, the problem can be solved in time $O((m + \lambda \cdot occ) \log \log n)$, within $O(\overline{r})$ space [19].

A natural question is whether one can obtain optimal time, $O(m + occ)$, within space proportional to some measure of repetitiveness (like r and $\overline{r}$, in the current solutions). In this paper we present the first index of this kind, whose space is related to measure e, which is the size of the CDAWG of T [7]. The CDAWG is a directed acyclic graph obtained by merging identical subtrees of the suffix tree: since the suffix tree of repetitive text collections tends to have many identical subtrees, e is a meaningful measure of repetitiveness. We actually build on a Symmetric CDAWG (SCDAWG) [12], which includes the $\overline{e}$ edges of the CDAWGs of T and of the reverse of T (the nodes are the same for both CDAWGs). Although it always holds $\overline{e} \geq r$, $\overline{e}$ is incomparable with $r \log \frac{n}{r}$ [20], which is the best space of previous solutions [19,1]. Concretely, our index uses space $O(\overline{e})$ and performs Contextual Pattern Matching in optimal time, $O(m + occ)$.

We then turn our attention to the enumeration problem: since the number of occurrences can be huge, we wish to provide an efficient enumerator. This is a data structure that, after some (ideally small) preprocessing time, can deliver any new occurrence within a bounded *delay*. For example, it is easy to support enumeration for Pattern Matching, using a suffix tree with threaded leaves, with $O(m)$ preprocessing and $O(1)$ delay, but this is not so easy for Contextual Pattern Matching, even with $O(n)$ space. Our optimal-time $O(m + occ)$ algorithm

naturally yields an enumerator with $O(\lambda)$ delay, but we can do better: We give the first enumerator for Contextual Pattern Matching that, using $O(\overline{e})$ space, provides $O(m)$ preprocessing time and $O(\log\log\lambda)$ delay.

To obtain this delay, we develop an improved linear-space distance-sensitive data structure for weighted ancestor queries [8,13,6], which is of independent interest. Its query time is loglogarithmic on the difference of weights between the query and answer nodes. We build our solution on top of previous work [13], which incurs an $O(\log^* n)$ extra penalty factor in order to achieve linear space. Using new techniques from the realm of compact data structures, we reduce this extra factor to constant (another solution to remove the $O(\log^* n)$ term exists [11], but we believe ours is simpler).

2 Preliminaries

By $[i,j]$ we denote $\{i, i+1, \ldots, j\}$, if $i \leq j$, and $[i,j] = \emptyset$, otherwise. By $[n]$ we refer to $[1,n]$. $T[1,n]$ denotes a length-n string over an alphabet $\Sigma = \{1,\ldots,\sigma\}$. By $|T|$ we refer to the length of T. $T[i]$ denotes the i-th symbol of T, and $T[i,j] = T[i]\cdots T[j]$ if $[i,j] \neq \emptyset$, otherwise ($j < i$) $T[i,j] = \epsilon$, which is the only string satisfying $|\epsilon| = 0$. $T[1,i]$ is the prefix of T ending at position i and $T[j,n]$ is the suffix of T starting at position j. We assume T ends with a special symbol \$ that only occurs at $T[n] = \$$ and that is smaller than any other symbol $c \in \Sigma$. To enforce that the special symbol \$ always appears at the end of our input strings, we define the reverse T^{rev} of T as $T^{\text{rev}} = T[n-1]\cdots T[1]\$$.

Given $T \in \Sigma^*$, we say that $T[i,j]$ is a right-maximal substring of T if $j = n$ or there exists $a, b \in \Sigma$ ($a \neq b$) such that $T[i,j]a$ and $T[i,j]b$ are both substrings of T; $T[i,j]a$ and $T[i,j]b$ are said to be right-extensions. Similarly, we say that $T[i,j]$ is a left-maximal substring of T if $i = 1$ or there exists $c, d \in \Sigma$ ($c \neq d$) such that $cT[i,j]$ and $dT[i,j]$ are both substrings of T; $cT[i,j]$ and $dT[i,j]$ are called left-extensions. We say that $T[i,j]$ is maximal if it is both right-maximal and left-maximal.

2.1 Suffix trees

The *suffix tree* [22,15,21] of a string T over alphabet Σ, $\text{STree}(T) = (V^T_{\text{STree}}, E^T_{\text{STree}})$ is the compacted trie of all suffixes of T, where V^T_{STree} is the set of nodes and E^T_{STree} is the set of edges. Every node with out-degree greater than zero is called an *internal node*. The remaining nodes are called *leaves*. The elements of E^T_{STree} are tuples of the form (u,s,v) where the origin u is an internal node, the destination node $v \neq u$ cannot be the root, and $s \in \Sigma^+$ is the label of the edge; we call $|s|$ the edge length. We denote by $\ell(v)$ the string obtained by concatenating the edges' labels in the path from the root to a node v. It is well-known that each $\ell(v)$ corresponds to a different right-maximal substring of T, and each right-maximal substring s of T is the label $\ell(v)$ for some v, so there is a bijection between nodes in V^T_{STree} and right-maximal substrings of T.

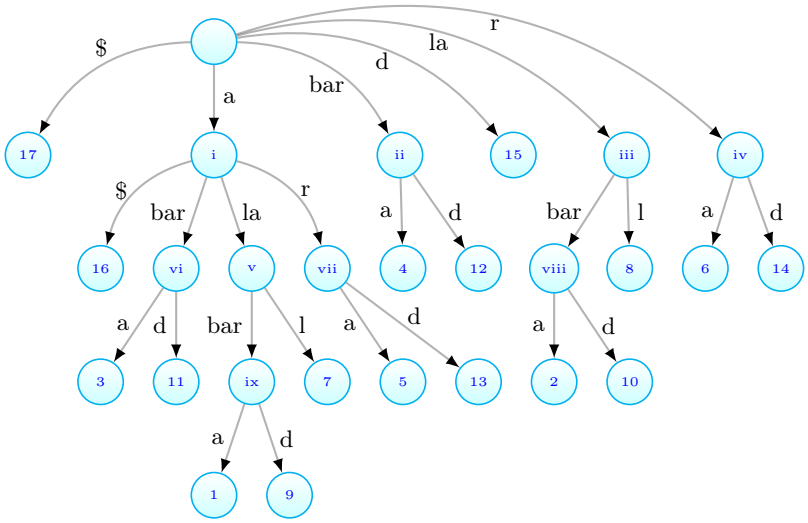

Fig. 1. Suffix tree for the string "alabaralalabarda". Edges leading to leaves show only the first symbol of their label, which continues spelling the text until the $ terminator. Leaves are labeled with the starting position of the corresponding suffix, and internal nodes with Roman numbers.

From the latter we can see that every internal node has at least two outgoing edges, and the labels of all the edges leaving a node start with a different symbol. Additionally, the edges leaving a node are sorted lexicographically by their labels, so concatenating the labels of the edges in the path from the root to the i-th leaf spells out the i-th suffix of T in lexicographically sorted order. Also, the leaves store the position where the suffix it represents starts in T and collecting those positions in leaves from left to right we obtain the *suffix array* [14].

The *locus* of P is the highest node (i.e., whose path connecting it with the root has the least number of edges) v such that P is a prefix of $\ell(v)$. Having found the locus u of P by descending from the root matching its symbols, the positions of the occ occurrences of P can be reported in $O(occ)$ time by collecting all the leaves under u.

We assume the labels of edges (u, s, v) are not stored explicitly but just the indices i, j of one occurrence of s in T (namely, $s = T[i, j]$). Since the number of nodes and edges is proportional to the number of right-maximal substrings in T, then by storing the input string T, STree(T) consumes $O(n)$ space.

Figure 1 shows the suffix tree for the string "alabaralalabarda".

2.2 CDAWGs

The *Compact Directed Acyclic Word Graph (CDAWG)* of T [7] is a directed acyclic graph $\text{CDAWG}(T) = (V^T_{\text{CDAWG}}, E^T_{\text{CDAWG}})$ that represents the suffixes of

T. It is well-known that in CDAWG(T), a node $v \in V^T_{\text{CDAWG}}$ represents a set of nodes from V^T_{STree}, such that all of them are roots of isomorphic sub-trees of STree(T). So, if a node $u \in V^T_{\text{CDAWG}}$ represents nodes $u_1, \ldots, u_k \in V^T_{\text{STree}}$ then $\ell(u_a)$ is a suffix of $\ell(u_b)$, for any $|\ell(u_a)| \leq |\ell(u_b)|$, with $a, b \in [k]$. The label of node u is then defined as $\ell(u) = \ell(u_a)$ where u_a has the longest label ℓ among all nodes represented by u. Therefore, $\ell(u)$ is a maximal substring of T, and there is a bijection between the nodes in V^T_{CDAWG} and maximal substrings of T. Moreover, for each edge $(u, s, v) \in E^T_{\text{CDAWG}}$, the label s is a right-maximal substring of T, its first symbol $s[1]$ is a right-extension of the maximal string $\ell(u)$, and $\ell(u) \cdot s$ is a suffix of $\ell(v)$. In the same way as in STree(T), in CDAWG(T) the edges leaving a node are lexicographically sorted by their label. In CDAWG(T) there is a single node with out-degree zero, which is called the *sink*, and, unlike STree(T), the leaves are the edges whose endpoint is the sink. We define $|u|$ as the number of leaves reachable from node u. We will refer to the string length of a path as the sum of the lengths of the edges in the path.

The locus u of a string $P[1, m]$ in CDAWG(T) is the endpoint of the path that spells out P from the root. Unlike in the suffix tree, where P must prefix $\ell(u)$, in a CDAWG P occurs at some position i in $\ell(u)$, that is, $P = \ell(u)[i, m + i - 1]$. We find u by descending from the CDAWG root; the process is analogous to the descending walk from the suffix tree root. By storing the position in T of the suffix $\ell(v) \cdot s$ for each leaf (v, s, w) (so, w is the sink), all the occurrences of P can be retrieved by collecting the leaves reachable from node u and adding the offset i.

The size e_T of CDAWG(T) is defined as the number of edges in E^T_{CDAWG}, so e_T is the number of right-extensions of maximal strings in T. Similarly to E^T_{STree}, the labels in the edges are not explicitly stored, but as the indices of one occurrence of the label, so CDAWG(T) uses $O(e_T)$ space on top of T.

Example 1. Figure 2 shows the CDAWG for the string "alabaralalabarda". Observe that, in the STree for the same string (Figure 1), the root, node i and node iii have edges labeled bar that reach nodes ii, vi and $viii$, whose subtrees are isomorphic. In the CDAWG, there are corresponding edges labeled bar, reaching node 4, from nodes 1, 2 and 3. The subtrees of iv and vii are also isomorphic, and $\ell(iv) = r$ and $\ell(vii) = ar$ are suffixes of bar. The CDAWG has corresponding edges labeled r to node 4, from nodes 1 and 2. Thus, $\ell(4) = alabar$ and all the paths from the root to node 4 spell out a suffix of $\ell(4)$. For example, node 4 is the locus of $P = aba$, an P (always) occurs somewhere inside $\ell(4)$. Instead, the locus of P in the STree is node vi, and P is (always) a prefix of its locus $\ell(vi) = abar$. □

2.3 SCDAWGs

The *Symmetric* CDAWG of T [12], SCDAWG(T) = $(V^T_{\text{CDAWG}}, E^T_{\text{SCDAWG}})$ is the CDAWG of T enhanced with *left-edges* (u, s, v) such that $s[|s|]$ is a left-extension of the maximal string $\ell(u)$ and $s \cdot \ell(u)$ is a prefix of $\ell(v)$ [7]. The left-edges leaving a node are sorted co-lexicographically by edges' labels (namely, right-to-left).

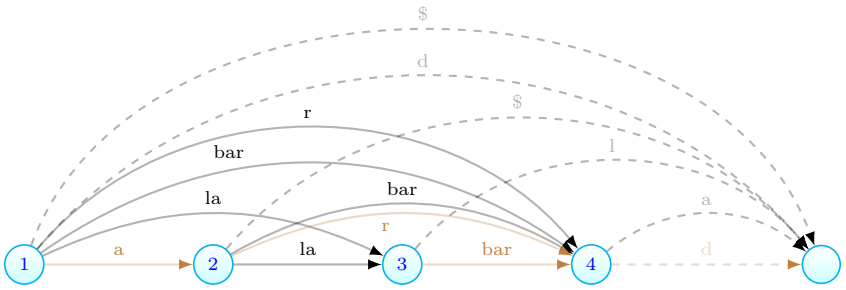

Fig. 2. CDAWG for the string "alabaralalabarda". Dashed edges represent leaves. The root is node 1 and the non-numbered node is the sink. Edges leading to the sink (the leaves) show only the first symbol of their label, which continues spelling the text until the $ terminator. For ease to draw, edges are not shown in sorted order of labels. The edges chosen to be "default" to build tree $\mathcal{T}$ in Figure 4 are colored in brown.

For a given left-edge $(u, s, v) \in E^T_{\text{SCDAWG}}$, since $\ell(u)$, $\ell(v)$ are maximal strings, then in CDAWG(T^{rev}) there exist exactly two nodes u_l, v_l such that $\ell(u_l) = \ell(u)^{\text{rev}}$, $\ell(v_l) = \ell(v)^{\text{rev}}$ and $\ell(u_l) \cdot s^{\text{rev}}$ is a suffix of $\ell(v_l)$, so in $E^{T^{\text{rev}}}_{\text{CDAWG}}$ there is an edge $(u_l, s^{\text{rev}}, v_l)$. Then, there is a one-to-one correspondence between nodes in CDAWG(T) and CDAWG(T^{rev}), and the left-edges in E^T_{SCDAWG} are exactly the edges in $E^{T^{\text{rev}}}_{\text{CDAWG}}$ (yet, note that the edge labels are s, not s^{rev}). It naturally follows that the size of SCDAWG is $\overline{e_T} = e_T + e_{T^{\text{rev}}}$. Figure 3 shows the SCDAWG for the string "alabaralalabarda".

SCDAWG(T) can be built in $O(n \log \sigma)$ time by computing CDAWG(T) and CDAWG(T^{rev}) in $O(n)$ time [12] and mapping each node by following the path of each maximal string s and s^{rev} in CDAWG(T) and CDAWG(T^{rev}), respectively. Further, Blumer et al. [7] give an $O(n)$-time construction, and Inenaga et al. [12] give an online linear-time algorithm.

3 Optimal Contextual Pattern Matching in $O(\overline{e})$ Space

3.1 Sub-optimal solution

Let u be the CDAWG node that is the locus of P and let $h \geq m$ be, initially, the string length of the path we followed from the CDAWG root to u. Note that u itself is the only right-context locus if $h \geq m + \lambda$, so below we assume the non-trivial case, where we still need to match $m + \lambda - h > 0$ symbols from the right context. The loci of the right-contexts of P are then all the closest descendants v of u reached by paths of string length $\geq m + \lambda - h$. So, for each right-edge (u, s, v) we must evaluate if $m + \lambda \leq h + |s|$, in which case v is the locus of a right-context of P, and P occurs at position $|\ell(v)| - (h + |s|) + 1$ of $\ell(v)$. Otherwise, we continue recursively looking from right-context loci from v, updating h to $h + |s|$.

Let us now find the left-contexts for each right-context found at a node v. Note that we have reached v by a path of length $h \geq m + \lambda$, which matches

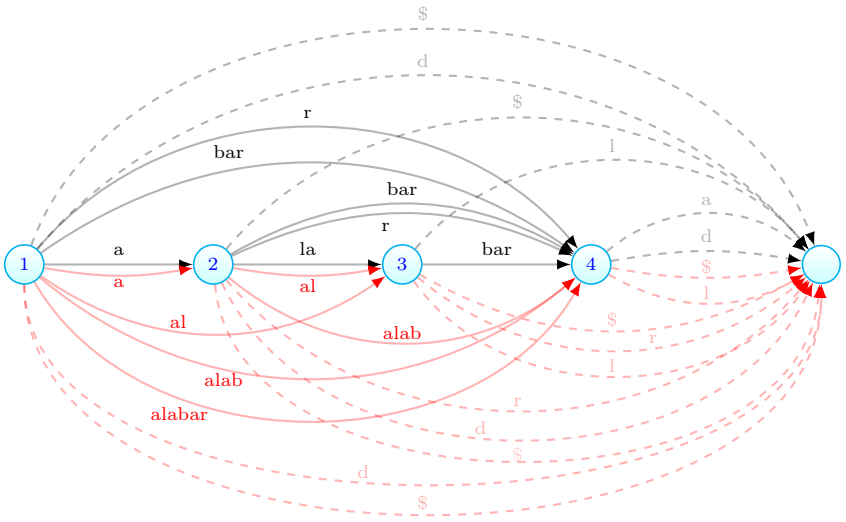

Fig. 3. SCDAWG for the string "alabaralalabarda". Black edges represent right-edges, and red edges represent left-edges. Dashed edges represent leaves. The root is node 1 and the non-numbered node is the sink. Right-edges leading to the sink show only the first symbol of their label, which continues spelling the text until the $ terminator, and left-edges leading to the sink show only the last symbol of their label, which continues spelling the text to the left until the first symbol of the string (assuming a $ terminator to the left of it). For ease to draw, edges are not shown in sorted order of labels.

P and a right-context, so h is the length of the suffix of $\ell(v)$ starting with the occurrence of P. If $|\ell(v)| - h \geq \lambda$, then, all those right-contexts occur with the same left-context and we are done. Otherwise, we set $h' = |\ell(v)| - h$ and start traversing left-edges to find left-contexts. For each left-edge (v, t, w) we evaluate if $\lambda \leq h' + |t|$ to determine if we have matched a left-context of P.

Note that, if for some right-edge (resp., left-edge), it happens that the destination is the sink, then this edge is a leaf and this contextual occurrence does not share a length-λ right-context (resp., left-context) with any other occurrence of P, so we can report it immediately.

Example 2. In Figure 3, for the right-edge $(2, bar, 4)$ we have $\ell(2) = a$, $\ell(4) = alabar$, and $\ell(2) \cdot bar$ is a suffix of $\ell(4)$. Note also that following this right-edge extends the contexts of the string a to the right and to the left, i.e., we get $al \cdot a \cdot bar$.

Similarly, following the left-edge $(2, alab, 4)$ we have that $alab \cdot \ell(2)$ is a prefix of $\ell(4)$. Further, following this edge extends the contexts of the string a to the right and to the left, i.e., we get $alab \cdot a \cdot r$. □

Algorithm 1 gives the procedure to report all the contextual occurrences of P once we have found its locus u and the string length h of the path followed to u. It uses $O(\overline{e}_T)$ space by using a SCDAWG augmented with some fields in

Algorithm 1: Function `contextual(`u, h`)` reports all the λ-contextual occurrences of $P[1, m]$ from its locus u and the string length h of the path followed to u. Function `leftContextual(`v, h'`)` reports all the λ-contextual occurrences reachable by traversing $\lambda - h'$ symbols labeling left-edges from v. The fields $e.occ$ and $v.occ$ are text positions where the strings represented by leaves e or nodes v, respectively, occur in T. We report the positions where P occurs.

```
1  function contextual(u, h)
2  |  if h ≥ m + λ then leftContextual(u, |ℓ(u)| − h) ;
3  |  else
4  |  |  for each right-edge e = (u, s, v) do
5  |  |  |  if v is the sink then report e.occ + (|ℓ(u)| − h + 1);
6  |  |  |  ; else contextual(v, h + |s|);

7  function leftContextual(v, h')
8  |  if h' ≥ λ then report v.occ + (h' + 1);
9  |  else
10 |  |  for each left-edge e = (v, t, w) do
11 |  |  |  if w is the sink then report e.occ + (h' + 1 + |t|);
12 |  |  |  else leftContextual(w, h' + |t|);
```

nodes and edges: For every node $u \in V^T_{\text{SCDAWG}}$, we store the length of $\ell(u)$ and the position $u.occ$ of one occurrence of $\ell(u)$ in T. Additionally, for every right-edge $e = (u, s, v)$, we store the edge length $|s|$, and if e is a leaf, we store the position $e.occ$ of the suffix of T that e represents. We store analogous fields for left-edges.

To find the locus of $P[1, m]$, we descend by following right-edges while matching substrings of P in SCDAWG. We start by setting u to the SCDAWG root, use a variable h (initially set to 0) to mark the part of P already compared. We binary search right-edges of u to find the edge $e = (u, s, v)$ such that $s[1] = P[h+1]$. If neither s is a prefix of $P[h + 1, m]$ or vice-versa, then P does not occur in T. Otherwise, we set $h \leftarrow h + |s|$ and $u \leftarrow v$. We repeat this process until $m \leq h$, so u is the locus of P. Then we call `contextual(`u, h`)` of Algorithm 1.

The search for the locus of P takes $O(m \log \sigma)$ time. After finding it, for each edge traversed from a node to all its children we increment the number of contextual occurrences to report, so it takes $O(occ)$ time to find all occ contextual occurrences of P.[3] The overall time is thus $O(m \log \sigma + occ)$. The space usage is $O(\overline{e}_T)$ on top of T, but it still needs T to compare P with the edges of SCDAWG, so we use $O(\overline{e}_T + n) = O(n)$ space. We next solve this space issue, and also improve the locus finding time.

[3] As an alternative amortized argument, note that the tree of recursive calls to `contextual` and `leftContextual` reports one occurrence at each leaf and has at least two children per node, so there are less than two calls per occurrence reported.

3.2 Optimal-time solution

We now improve the above solution in both time and space. Belazzougui and Cunial [4] give a data structure of size $O(e_T)$ that determines if a pattern $P[1,m]$ occurs in a string $T[1,n]$ in $O(m)$ time. Using this structure, we first determine whether P occurs in T. If P does not occur in T, then there are no contextual occurrences. Otherwise, we search for the locus of P through a *blind search* [4]. To do this, we add at each edge $e = (u, s, v)$ of our data structure the first character of the label s in a variable $e.label = s[1]$ (we now do not store s, but still store $|s|$). Now, with u starting from the root and $h = 0$, we follow the edge e of u such that $e.label = P[h+1]$ (we use perfect hashing [10] to determine which edge to follow in $O(1)$ time), and update h, and u as before. Since we know that P occurs in T, we can skip comparing the whole label of e against $P[h+1,m]$. We continue that way until $m \leq h$, which means u is the locus of P. With this change, the overall time is now $O(m + occ)$ and we can drop the text T (because labels are not needed anymore).

Theorem 1. *The occ solutions to Contextual Pattern Matching for $P[1,m]$ can be listed in optimal time $O(m + occ)$, on a data structure of size $O(\overline{e})$.*

4 Enumerators for Contextual Pattern Matching

A standard traversal of the *occ* results, though taking $O(occ)$ time, may spend $O(\lambda)$ time between some result and the next. We now describe an enumerator that, after $O(m)$ time preprocessing, can deliver each new solution to the Contextual Pattern Matching problem within $O(\log\log\lambda)$ time.

The idea is inspired by a recent result by Adamson et al. [2], for enumerating the paths of a certain length in a graph. Let us first describe our solution on CDAWGs for simplicity. We arbitrarily choose exactly one (right-)edge leaving each (non-sink) node as its *default* edge. The set of all default edges then defines an "enumerator" tree $\mathcal{T}$, where the target of the default edge is the parent of the source and the CDAWG sink is the root of $\mathcal{T}$. The nodes u of $\mathcal{T}$ have weight $w(u)$ defined as the sum of edge lengths from the root of $\mathcal{T}$ to u. The size of $\mathcal{T}$ is the number of CDAWG nodes. Figure 4 shows one possible enumerator tree for the right-edges of the SCDAWG of the string "alabaralalabarda".

Let u be the CDAWG node that is the locus of P, found in $O(m)$ time as explained in Section 3.2. Let $h \geq m$ be the string length of the path we followed from the CDAWG root to u, computed as in Section 3.1. Our first task is, again, to enumerate the loci of the right-contexts of P, that is, the closest descendants v of u by paths of string length $\geq m + \lambda - h$.

We will first find one of those loci, u^*, which is reached from u by default edges, that is, u^* is an ancestor of u in $\mathcal{T}$. Node u^* is reached from u in $\mathcal{T}$ using a *weighted ancestor query* on the node weights: u^* is the lowest ancestor of u such that $w(u) - w(u^*) \geq m + \lambda - h$, or equivalently, $w(u^*) \leq w(u) - (m + \lambda - h)$.[4]

[4] To capture contexts shorter than λ near the end of T, we set $w(\cdot) = -\infty$ for the sink node.

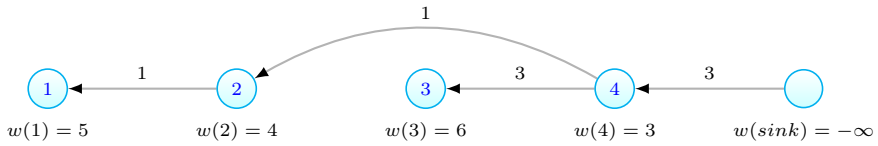

Fig. 4. A possible enumerator tree $\mathcal{T}$ for the right-edges of the SCDAWG of the string "alabaralalabarda". The default edges chosen are those colored in brown in Figure 2. Edges (u, s, v) are labeled by their edge length $|s|$.

We show in Section 5 how to solve this query in time $O(\log\log\lambda)$, by exploiting the fact that the difference between $w(u)$ and the weight of the child of u^* is at most λ.

Once we have obtained our first locus, u^*, we *visit* all of its CDAWG ancestors x by default edges, from $x = u$ to (but not including) u^*. *Visiting* x means descending from x by all the non-default edges (x, s, y) and repeating the process recursively from every such y, by regarding y in $\mathcal{T}$, performing a weighted ancestor query to find a locus y^*, and *visiting* all the nodes in the path from y to the parent of y^*.

Along this recursive procedure, we deliver each new locus node v with delay $O(\log\log\lambda)$: our recursion first performs an $O(\log\log\lambda)$-time weighted ancestor query and immediately enumerates the locus u^*. Then it visits all the nodes x in the path between u and u^*. There is exactly one default child of each node x and at least one non-default child, so all the non-default children y of all nodes x in the path are found with constant delay. Further, there is at least one other occurrence to report below each such y. The argument then applies inductively in the subtree rooted at each child y (we find a first occurrence y^* in $O(\log\log\lambda)$ time by the default path, report it, visit the nodes in between, and so on).

Example 3. Consider, in Figure 2, the process of finding the right-contexts of $P =$"a" with $\lambda = 2$. We arrive at node $u = 2$ with $h = 1$. A weighted ancestor query (see Figure 4) for the closest node at distance $\geq m+\lambda-h = 2$ yields u^* as the sink, where we report the right-context "rd". We now traverse the path in $\mathcal{T}$ from u to u^*, visiting the nodes 2 and 4. Node 2 has three other right-edges that do not lead to node 4 by "r" (the default edge): one leads to the sink by "$", another to node 3 by "la", and the third to node 4 by "bar". The first case yields the empty right-context corresponding to the final "a" in the text. In the other cases we exceed the context length λ immediately, yielding the contexts "la" and "ba", respectively. We now descend by the default edge to node 4 by "r". From it, we descend by the only non-default edge to the sink, by "a", leading to the remaining right-context, "ra". □

To complete the process, we must obtain all the left-contexts that correspond to each right-context. Those are obtained with essentially the same technique, now using the left-edges of the SCDAWG. We build another enumerator tree,

Algorithm 2: Iterator `enumerate`(u, h) enumerates all the λ-contexts from the locus u of $P[1, m]$; $h \geq m$ is the string length of the path to the locus. Auxiliary enumerator `leftEnumerate`(v, h') enumerates all the left-contexts reachable by traversing $\lambda - h'$ symbols from v. $\mathcal{T}$ and $\mathcal{T}'$ are the enumerator trees for right- and left-edges, respectively, and w is the node weight in those trees. Keyword **report** enumerates each new context; we report the positions where P occurs.

```
1  function enumerate(u, h)
2      if h ≥ m + λ then leftEnumerate(u, |ℓ(u)| − h) ;
3      else
4          u* ← wancestor(T, u, w(u) − (m + λ − h)) ;
5          leftEnumerate(u*, |ℓ(u*)| − (h + w(u*) − w(u))) ;
6          x ← u ;
7          while x ≠ u* do
8              (x, s', y') ← default right-edge leaving x ;
9              for each right-edge (x, s, y) ≠ (x, s', y') do
10                 enumerate(y, h + |s|) ;
11             x ← y' ;
12             h ← h + |s'| ;

13 function leftEnumerate(v, h')
14     if h' ≥ λ then report v.occ + (h' + 1) ;
15     else
16         v* ← wancestor(T', v, w(v) − (λ − h')) ;
17         report v*.occ + (h' + w(v*) − w(v) + 1);
18         y ← v ;
19         while y ≠ v* do
20             (y, t', z') ← default left-edge leaving y ;
21             for each left-edge (y, t, z) ≠ (y, t', z') do
22                 leftEnumerate(z, h' + |t|) ;
23             y ← z' ;
24             h' ← h' + |t'| ;
```

$\mathcal{T}'$, on the left-edges, and at query time repeat the process from the locus v of each new right-context. If the path followed to v was of string length $h \geq m + \lambda$, then the left-context loci are the closest nodes reached from v by left-edge paths of string length $\geq \lambda - (\ell(v) - h)$. When all the left-contexts of v are reported, we switch to the next right-context. The total delay is still $O(\log \log \lambda)$.

Algorithm 2 gives the pseudocode. Though we describe recursive procedures for enumeration, those are easily turned into iterators by maintaining the state of the computation and returning to the caller with each new result. The memory required by the enumeration is $O(\lambda)$, as λ is the maximum height of the recursion stack (just as with the standard traversal procedure).

Theorem 2. *The solutions to Contextual Pattern Matching for $P[1,m]$, with context length λ, can be enumerated with $O(m)$ preprocessing time and $O(\log\log\lambda)$ delay, on a data structure of size $O(\overline{e})$. The enumerator requires $O(\lambda)$ space.*

5 Distance-Sensitive Weighted Ancestor Queries

The weighted ancestor problem on trees assumes that nodes have integer weights $w(\cdot)$ that are increasing towards descendants. Given a tree node u and a target weight t, the query finds the lowest ancestor u^* of u such that $w(u^*) \leq t$. It has been shown that this problem can be reduced essentially to (a constant number of) predecessor queries [8,13], and thus it can be solved in $O(n)$ space and $O(\log\log n)$ time on a tree of n nodes.

We are interested in exploiting the fact that $w(u) - t$ may be small (it is at most λ in the preceding section), aiming at time $O(\log\log(w(u) - t))$. This does not immediately stem from previous work [13], as they include an additive term of the form $O(\log^* n)$, necessary to achieve linear space. We follow their basic scheme, using a distance-sensitive predecessor data structure, and use a different way to achieve linear space (Gawrychowski et al. [11] give an alternative way to remove this $O(\log^* n)$ additive term using micro/macro trees, yet still do not aim at the distance-sensitive time we obtain in this section).

We partition the tree into paths; those can be heavy paths [8] or centroid paths [13] indistinctly for us. The important property is that every node reaches the root with only $O(\log n)$ changes from one path to another. Each node stores the weights of the highest nodes in each of those paths that connect it to the root. A first part of the search for u^* is to determine in which of those paths it lies. Since this is a predecessor query for t on $O(\log n)$ integer weights, the correct path π is found in constant time using fusion trees [9].

On each path π we build a linear-space distance-sensitive predecessor data structure [3], so we again look for the predecessor of t and finish. The query time of this data structure is $O(\log\log\Delta)$, where Δ is the minimum between $t - w(u^*)$ and $w(u') - t$, where u' is the child of u^* in its path to u. Since $w(u') - t \leq w(u) - t$, we have the desired time complexity.

The problem of this scheme is that its space complexity is $O(n \log n)$. Just as in previous work [13], we prune the tree, turning into leaves the lowest nodes whose subtree has more than $\log n$ nodes. The pruned tree then has $O(n/\log n)$ nodes and we can afford to use the general scheme on it within $O(n)$ space. On the small detached subtrees, with root x, we first check if $w(x) > t$, in which case we rerun the query from the leaf x that corresponds to in the main tree. If $w(x) \leq t$, instead, we must run the query on the small subtree.

To do this efficiently, we repeat the same scheme on the small trees. Now there are $O(\log(\log n))$ paths between a node and its root, so the extra space is $O(n \log\log n)$. To maintain linear space, we prune those small subtrees, turning into leaves their nodes whose subtree sizes exceed $\log\log n$. We now must show how to solve queries on the tiny detached subtrees of size $O(\log\log n)$.

Instead of continuing this process for $O(\log^* n)$ rounds [13], we resort to universal tables at this point. Let $k = O(\log \log n)$ be the maximum size of the tiny trees. We remap their weights to the interval $[1, k]$, and store a fusion tree on those k elements that allows us map t to the corresponding value in constant time. Now, $2k$ bits suffice to describe the tree topology [16] and $k\lceil \log_2 k \rceil$ bits suffice for the weights. There are k^2 possible weighted ancestor queries (k nodes and k target weights). Therefore, a table of $2^{k(2+\lceil \log_2 k \rceil)} \times k^2 = o(n)$ entries can store all the precomputed answers to queries on every possible tiny tree.

Theorem 3. *Weighted ancestor queries (u, t) on a tree of n nodes with integer weights $w(\cdot)$ can be solved in time $O(\log \log(w(u) - t))$ using an $O(n)$-space data structure.*

6 Conclusions and Future Work

We have presented the first compressed index (for highly repetitive text collections) that supports contextual pattern matching in optimal time, $O(m + occ)$. The space, $O(\overline{e})$, is proportional to the size of the SCDAWG of the text. We have also presented an enumerator for the results, which after $O(m)$ time delivers each new contextual occurrence in time $O(\log \log \lambda)$, for a context length λ. In our way, we uncover a new, improved, linear-space distance-sensitive data structure for weighted ancestor queries on trees.

One interesting challenge is to obtain constant delay, which can be achieved by running weighted ancestor queries in constant time. This has been shown to be possible on suffix trees [11,5]; the results could perhaps be translated to our enumerator trees $\mathcal{T}$ and $\mathcal{T}'$. Another interesting challenge is to solve contextual pattern matching, even if not in optimal time, within space bounded by a stronger measure of repetitiveness, such as the size of a context-free grammar representing the text.

Acknowledgements

We thank the reviewers for their careful reading and detailed comments to improve our presentation.

References

1. Abedin, P., Chubet, O., Gibney, D., Thankachan, S.V.: Contextual pattern matching in less space. In: Proc. 33rd Data Compression Conference (DCC). pp. 160–167 (2023). https://doi.org/10.1109/DCC55655.2023.00024
2. Adamson, D., Gawrychowski, P., Manea, F.: Enumerating m-length walks in directed graphs with constant delay. In: Proc. 16th Latin American Symposium on Theoretical Informatics (LATIN), Part I. pp. 35–50 (2024). https://doi.org/10.1007/978-3-031-55598-5_3

3. Belazzougui, D., Boldi, P., Vigna, S.: Predecessor search with distance-sensitive query time. CoRR **1209.5441** (2012). https://doi.org/10.48550/arXiv.1209.5441
4. Belazzougui, D., Cunial, F.: Fast label extraction in the CDAWG. In: Proc. 24th String Processing and Information Retrieval (SPIRE). pp. 161–175 (2017). https://doi.org/10.1007/978-3-319-67428-5_14
5. Belazzougui, D., Kosolobov, D., Puglisi, S.J., Raman, R.: Weighted ancestors in suffix trees revisited. In: Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 8:1–8:15 (2021). https://doi.org/10.4230/LIPIcs.CPM.2021.8
6. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Rao, S.S., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM Journal on Computing **44**(3), 513–539 (2015). https://doi.org/10.1137/130936889
7. Blumer, A., Blumer, J., Haussler, D., McConnell, R.M., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34**(3), 578–595 (1987). https://doi.org/10.1145/28869.28873
8. Farach, M., Muthukrishnan, S.: Perfect hashing for strings: Formalization and algorithms. In: Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 130–140 (1996). https://doi.org/10.1007/3-540-61258-0_11
9. Fredman, M., Willard, D.: Surpassing the information theoretic bound with fusion trees. Journal of Computer and Systems Science **47**(3), 424–436 (1993). https://doi.org/10.1016/0022-0000(93)90040-4
10. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $O(1)$ worst case access time. Journal of the ACM **31**(3), 538–544 (1984). https://doi.org/10.1145/828.1884
11. Gawrychowski, P., Lewenstein, M., Nicholson, P.: Weighted ancestors in suffix trees. In: Proc. 22nd Annual European Symposium on Algorithms (ESA). pp. 455–466 (2014). https://doi.org/10.1007/978-3-662-44777-2_38
12. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S.: On-line construction of symmetric compact directed acyclic word graphs. In: Proc. 8th Symposium on String Processing and Information Retrieval (SPIRE). pp. 96–110 (2001). https://doi.org/10.1109/SPIRE.2001.989743
13. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 565–574 (2007). https://doi.org/10.5555/1283383.1283444
14. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
15. McCreight, E.: A space-economical suffix tree construction algorithm. Journal of the ACM **23**(2), 262–272 (1976). https://doi.org/10.1145/321941.321946
16. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing **31**(3), 762–776 (2001). https://doi.org/10.1137/S0097539799364092
17. Navarro, G.: Indexing highly repetitive string collections, part II: Compressed indexes. ACM Computing Surveys **54**(2), article 26 (2021). https://doi.org/10.1145/3432999
18. Navarro, G.: Contextual pattern matching. In: Proc. 27th String Processing and Information Retrieval (SPIRE). pp. 3–10 (2020). https://doi.org/10.1007/978-3-030-59212-7_1
19. Navarro, G.: Contextual pattern matching. CoRR **2010.07076** (2020), https://arxiv.org/abs/2010.07076

20. Navarro, G.: Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Computing Surveys **54**(2), article 29 (2021). https://doi.org/10.1145/3434399
21. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). https://doi.org/10.1007/BF01206331
22. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory. pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13