Compressed String Dictionaries via Data-Aware Subtrie Compaction*

Antonio Boffa[0000−0002−8178−135X], Paolo Ferragina[0000−0003−1353−360X], Francesco Tosoni[0000−0001−8457−3866], and Giorgio Vinciguerra[0000−0003−0328−7791]

Department of Computer Science, University of Pisa, Italy
antonio.boffa@phd.unipi.it, paolo.ferragina@unipi.it, francesco.tosoni@phd.unipi.it, giorgio.vinciguerra@di.unipi.it

Abstract. String dictionaries are a core component of a plethora of applications, so it is not surprising that they have been widely and deeply investigated in the literature since the introduction of tries in the '60s. We introduce a new approach to trie compression, called COmpressed COllapsed Trie (CoCo-trie), that hinges upon a data-aware optimisation scheme that selects the best subtries to collapse based on a pool of succinct encoding schemes in order to minimise the overall space occupancy. CoCo-trie supports not only the classic lookup query but also the more sophisticated rank operation, formulated over a sorted set of strings. We corroborate our theoretical achievements with a large set of experiments over datasets originating from a variety of sources, e.g., URLs, DNA sequences, and databases. We show that our CoCo-trie provides improved space-time trade-offs on all those datasets when compared against well-established and highly-engineered trie-based string dictionaries.

Keywords: String dictionaries · Tries · Compressed data structures.

1 Introduction

Let $\mathcal{S}$ be a *sorted set* of n variable-length strings $s_1, s_2, \ldots, s_n$ drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. The String Dictionary problem consists of storing $\mathcal{S}$ in a *compressed format* while supporting the `rank` operation that returns the number of strings in $\mathcal{S}$ lexicographically smaller than or equal to a pattern $P[1, p]$.

Some other classic operations such as `lookup`(P) (returning a unique stringID for P if $P \in \mathcal{S}$, and -1 otherwise), `access`(i) (returning the string in $\mathcal{S}$ having stringID i), `predecessor`(P) (returning the lexicographically largest string in $\mathcal{S}$ smaller than P), `prefix_range`(P) (returning all strings in $\mathcal{S}$ that are prefixed

* This version of the contribution has been accepted for publication in the Proceedings of the 29th International Symposium on String Processing and Information Retrieval (SPIRE), after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-20643-6_17. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use.

by P), `longest_prefix_match`(P) (returning the longest prefix of P which is shared with one of the strings in $\mathcal{S}$) can be implemented through the `rank` operation, possibly using compact auxiliary data structures [19].

String dictionaries constitute a core component of a plethora of applications such as query auto-completion engines [23, 26, 29], RDF and key-value stores [36, 50], computational biology tools [5, 33], and n-gram language models [27, 42], just to mention a few. They are typically approached via the *trie* data structure, which dates back to the '60s [31, §6.3]. Since then, researchers have put a lot of effort to improve the time and space efficiency of the naïve pointer-based implementation. Some solutions compact paths [19, 24, 28, 37] or subtrees [4,7,10,25,40,45,46], succinctly encode node fan-outs [9,15,32,34,43,48], apply sophisticated string transformations [20, 21] or proper disk-based layouts [8,18,22]. Many recent results aim at reducing further the space occupancy of tries without impairing their efficient query time via sophisticated compression techniques (see e.g. [4,6,10,11,14,28,35,43,45,46,50]). As a result, this plethora of proposals offers different space-time trade-offs over various datasets, but without a clear winner. Choosing the appropriate storage solution is indeed quite a daunting task, requiring specific expertise and accurate analysis of the input datasets.

In this paper, we tackle this long-standing problem by introducing a fully-new approach that exploits a principled and data-aware optimisation strategy to collapse and compress subtries. More precisely, we make the following contributions:

- We revisit the subtrie compaction technique by introducing a novel representation that encodes a collapsed subtrie via standard integer compressors. Then, by means of a concrete motivating example, we observe that the effectiveness of this compressed representation depends upon the "shape" of the collapsed subtrie and its possibly long "edge labels" (Section 2).
- In light of this, we propose a new data structure, called CoCo-trie, which stands for COmpressed and COllapsed trie.[1] It orchestrates three main tools: the above novel representation for collapsed subtries, a pool of succinct encoding schemes to compress the edge labels, and an optimisation procedure that selects the *best* subtries to collapse into macro-nodes to minimise the overall occupied space while guaranteeing efficient queries due to the shorter trie traversal and the efficiently-searchable encoding schemes (Section 3).
- We corroborate our theoretical results with an experimental evaluation on several datasets with different characteristics originating from a variety of sources (e.g. URLs, XML, DNA sequences, and databases) and against five well-established and highly-engineered competitors (namely, ART [32], CART [49], ctrie++ [46], FST [50], and PDT [24]). To the best of our knowledge, this is the very first work experimenting with all these implementations together over a wide variety of datasets. Our results show that CoCo-trie is a robust and flexible data structure since: in two cases, it significantly improves the space-time performance of all competitors; in two other cases, it is on the Pareto frontier of the best approaches (thus offering new competitive space-time trade-offs); and, in the last case, it is very close to the Pareto frontier (Section 4).

[1] The source code is publicly available at `https://github.com/aboffa/CoCo-trie`.

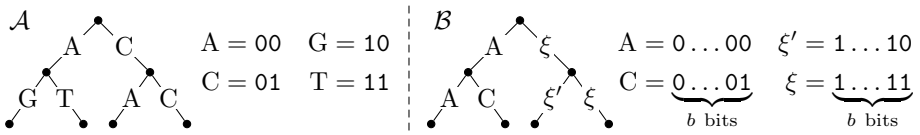

Fig. 1. Two tries $\mathcal{A}$ and $\mathcal{B}$ built on two sets of four strings each: $\{\mathrm{AG}, \mathrm{AT}, \mathrm{CA}, \mathrm{CC}\}$ on the left, and $\{\mathrm{AA}, \mathrm{AC}, \xi\xi', \xi\xi\}$ on the right. $\mathcal{A}$ uses just four alphabet symbols, and $\mathcal{B}$ uses a much larger alphabet in which ξ' and ξ are the last two symbols.

2 A motivating example

"Subtrie compaction" is a common technique in the design of compressed string dictionaries. However, it has been mainly investigated in the restricted context of either bounding the subtrie height, to fit the branching substring into one machine word [10, 45, 46]; or when bounding the macro-node fan-out so that more space-time efficient data structures can be used for it [7, 11, 25].

In what follows, we firstly introduce a novel macro-node representation, and then we provide a concrete example of the impact this technique can have on the space-time efficiency of the resulting trie representation. Our technique consists of properly choosing (i) the heights of the subtries to collapse into macro-nodes, and (ii) the coding mechanisms to represent the corresponding branching substrings (associated with the collapsed edge labels).

Consider the tries $\mathcal{A}$ and $\mathcal{B}$ of Figure 1 built respectively on the string sets $S_1 = \{\mathrm{AG}, \mathrm{AT}, \mathrm{CA}, \mathrm{CC}\}$ and $S_2 = \{\mathrm{AA}, \mathrm{AC}, \xi\xi', \xi\xi\}$, where ξ denotes the last symbol in a (potentially large) alphabet Σ, and ξ' denotes the symbol preceding ξ in Σ. In $\mathcal{A}$, the alphabet $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ consists of just 4 symbols, so we need 2 bits to represent them. In $\mathcal{B}$, the alphabet is assumed to be $\Sigma = \{\mathrm{A}, \mathrm{C}, \ldots, \xi', \xi\}$ and its symbols can be represented with $b = \lceil \log_2 |\Sigma| \rceil$ bits.

Let us now consider two scenarios for both of the tries above: one in which the trie $\mathcal{T} \in \{\mathcal{A}, \mathcal{B}\}$ succinctly encodes the individual branching symbols; the other one in which the two levels of $\mathcal{T}$ are *collapsed* at the root node, thereby creating a macro-root $\mathcal{T}^c$ with branching macro-symbols of length 2 symbols. For evaluating the space cost of encoding $\mathcal{T}$ and $\mathcal{T}^c$ we consider the following succinct scheme: for every node in level order, we store the first branching symbol explicitly and then encode the *gap* between successive symbols using some coding tool, say γ-code (see Appendix A for the definition of γ-code and the full calculations). If we refer, say, to the root of $\mathcal{A}$, its two branching symbols, namely A and C, are encoded in 3 bits as $enc(A) = \mathtt{00}$, followed by $\gamma(\mathrm{C} - \mathrm{A}) = \gamma(\mathtt{01} - \mathtt{00}) = \mathtt{1}$.

The encoding of $\mathcal{A}$ takes 9 bits, while if we collapse the two levels of $\mathcal{A}$ in the root of $\mathcal{A}^c$, this root gets four children whose edge labels are $\{\mathrm{AG}, \mathrm{AT}, \mathrm{CA}, \mathrm{CC}\}$, and their succinct representation takes 7 bits. Hence, the representation of $\mathcal{A}$ takes more space than the one of $\mathcal{A}^c$. Surprisingly, one comes to the opposite conclusion with $\mathcal{B}$, despite having the same topology of $\mathcal{A}$. Here, the larger alphabet together with the different distribution of the edge labels changes the optimal choice. Indeed, the encoding of $\mathcal{B}$ takes at most $5b + 1$ bits, while the one of $\mathcal{B}^c$

takes $6b-1$ bits. Hence, it is better not to collapse $\mathcal{B}$ because its succinct encoding takes b bits less than the one of $\mathcal{B}^c$, and b can make this gap arbitrarily large.

This example shows there is no *a priori* best choice about which subtrie to collapse, thus opening a significant deal of possible improvements to the known trie representations. In particular, the "best" choice depends upon several features, such as the trie structure, the number of distinct branching symbols at each node and their distribution among the trie edges. Consequently, designing a principled approach to finding that "best" choice for each trie node is quite a complex task, that we rigorously investigate throughout the rest of the paper.

3 CoCo-trie: Compressed Collapsed Trie

The simplest and most used approach to collapsing tries is to obtain the trie $\mathcal{T}_\ell$ by collapsing ℓ levels of the subtries rooted at the nodes whose distances from the root of $\mathcal{T}$ are multiple of ℓ. In this way, one can seek for a pattern $P[1,p]$ over $\mathcal{T}_\ell$ by traversing at most p/ℓ (macro-)nodes and edges, that is, p/ℓ branches over (macro-)characters (e.g., in [10, 45, 46], ℓ is the number of characters that fit into a RAM word). Obviously, increasing ℓ reduces the number of branching steps, but it may increase (i) the computational cost of each individual step, given that the number and the length of the branching characters increase; and, (ii) the space occupancy of the overall trie, given that shared paths within the collapsed subtries are turned into distinct substrings by macro-characters (see e.g. the paths "e$" and "es" descending from v in Figure 2, which share "e").

Our proposal addresses three main questions:

Q1: Can we tackle in a principled algorithmic way the issues (i) and (ii) above as ℓ increases?
Q2: How does the choice about the number ℓ of levels to collapse depend on the dictionary of strings?
Q3: Should the choice of ℓ be *global*, and thus unique to the entire trie, or should it be *local*, and thus vary among trie nodes?

These questions admit surprising answers in theory, which have equally-surprising impacts in practice. In particular, we will:

- answer Q1 affirmatively, by resorting to a pool of succinct encoding schemes to compress the possibly long edge labels (i.e., branching macro-characters);
- show for Q2 that the choice for ℓ has to account for the topology and edge labelling of the trie $\mathcal{T}$, and thus the characteristics of its indexed strings;
- show for Q3 that one has to find locally, i.e., node by node, the best value of ℓ, via a suitably-designed optimisation procedure aimed at minimising the overall space occupancy.

Our algorithmic answer consists of six main steps. Firstly, we introduce a novel compressed encoding for the collapsed subtries (Section 3.1). Secondly, we provide an optimisation procedure to choose the subtries to collapse (Section 3.2). Thirdly, we show how to select the best compression scheme for each

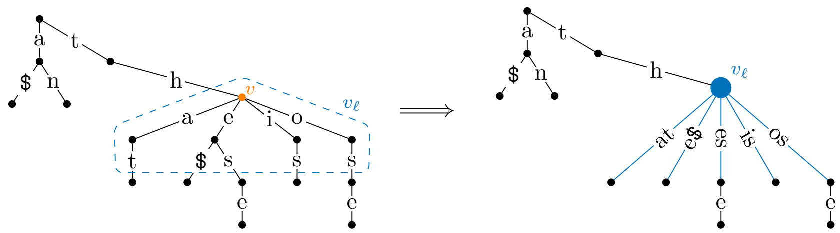

Fig. 2. Collapsing $\ell = 2$ levels of the subtrie rooted at v.

collapsed subtrie in a data-aware manner (Section 3.3). Fourthly, we present a further compression step that exploits the local alphabet of the edge labels in the collapsed subtrie (Section 3.4). Fifthly, we show how to trade space occupancy with query time (Section 3.5). Sixthly, we describe how to implement the `rank` operation over the resulting compressed trie structure (Section 3.6).

3.1 Compressed encoding of collapsed subtries

Let us be given a trie $\mathcal{T}$ whose edges are drawn from an integer alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ and sorted increasingly at each node. The special character 0 (indicated with `$`) is the string terminator. We formalise the notion of collapsed subtries as follows.

Definition 1. *Given an internal node v of a trie $\mathcal{T}$ and an integer $\ell \geq 1$, the* collapsing *of ℓ levels of the subtrie of $\mathcal{T}$ rooted at v consists in replacing this subtrie with a* macro-node *v_ℓ such that (i) the substrings branching out of v_ℓ are the ones corresponding to the paths of length ℓ descending from v in $\mathcal{T}$, and (ii) the children of v_ℓ are the nodes at distance ℓ from v. If branching substrings are shorter than ℓ, we pad them with the character `$`.*

This is depicted in Figure 2, where five paths of length $\ell = 2$ are collapsed to form the five branching edges $\{\text{at}, \text{e\$}, \text{es}, \text{is}, \text{os}\}$ of v_ℓ.

To encode a string s branching out of v_ℓ, we initially right-pad it with $\ell - |s|$ characters `$` if $|s| < \ell$; then, we assign it the integer

$$enc_\ell(s) = \sum_{i=1}^{\ell} s[i] \cdot \sigma^{\ell - i}. \tag{1}$$

Intuitively, we interpret $enc_\ell(s)$ as a branching *macro-character* of v_ℓ drawn from the integer alphabet $\Sigma^\ell = \{0, \ldots, \sigma^\ell - 1\}$.

Furthermore, we observe that enc_ℓ is monotonic, i.e., given two strings s' and s'' such that s' is lexicographically smaller than s'', then $enc_\ell(s') < enc_\ell(s'')$.

After computing each branching macro-character, we need to define a compression scheme that guarantees efficient access to edge labels, so to support fast pattern searches over the resulting collapsed trie. Let us assume that a macro-node v_ℓ has m branching macro-characters, indicated with $c^i_{v_\ell}$ for $i = 1, 2, \ldots, m$.

We explicitly encode the first macro-character $x = c^1_{v_\ell}$ using a fixed-size representation taking $\log \sigma^\ell$ bits,[2] and we represent the other $m-1$ macro-characters by encoding the sequence $c^i_{v_\ell} - x$ for $i = 2, \ldots, m$ with Elias-Fano (EF) [16,17], which takes $(m-1)(2+\log \frac{u}{m-1})$ bits, where $u = c^m_{v_\ell} - c^1_{v_\ell}$ is the universe size of the sequence. To decompress the EF sequence, we also need to store some small metadata taking $\log\log \frac{u}{m} \leq \log\log \frac{\sigma^\ell}{m}$ bits.

One should notice that other integer encoding schemes could be used in place of EF, and indeed we do so in Section 3.3.

Summing up, the space occupancy of the collapsed and compressed macro-node v_ℓ is (excluding EF's metadata)

$$C(v_\ell) = \log \sigma^\ell + (m-1)\left(2 + \log \frac{u}{m-1}\right) + 2 \text{ bits}, \tag{2}$$

where the first term corresponds to the space for the first macro-character, the second term accounts for the space to store the $(m-1)$ EF-coded integers, and the last 2 bits account for the contribution of the node v_ℓ to the space required by a succinct trie representation (we use LOUDS [39, §8.1]).

We underline that the subtraction of x has a subtle (yet paramount) impact on the space occupancy of our trie representation. It indeed removes any possible redundancy given by the longest common prefix (shortly, lcp) among the branching macro-characters. For instance, if we have $\ell = 2$ and the four branching macro-characters $\{\text{ha}, \text{he}, \text{hi}, \text{ho}\}$, then our encoding scheme stores $x = enc_\ell(\text{ha})$ explicitly as the integer $\text{h} \cdot \sigma^1 + \text{a} \cdot \sigma^0$, and it encodes the following three branching macro-characters $\{\text{he}, \text{hi}, \text{ho}\}$ as the difference with x, i.e., it encodes "he" as $enc_\ell(\text{he}) = (\text{h} \cdot \sigma^1 + \text{e} \cdot \sigma^0) - x = (\text{h} \cdot \sigma^1 + \text{e} \cdot \sigma^0) - (\text{h} \cdot \sigma^1 + \text{a} \cdot \sigma^0) = \text{e} - \text{a}$. So our encoding scheme stores the lcp "h" only once in x, thereby getting rid of much redundancy in the edge labels, and saving a big deal of space, especially when ℓ gets longer. As a matter of fact, we are reducing the value of the integers $enc_\ell(c^i_{v_\ell})$, which are upper-bounded by σ^ℓ, to the values $enc_\ell(c^i_{v_\ell}) - enc_\ell(c^1_{v_\ell})$, which are upper-bounded by $\sigma^{\ell - lcp}$.

3.2 On the choice of the subtries to collapse

We now get down to the details of our algorithm that, given an input trie $\mathcal{T}$, identifies which subtries of $\mathcal{T}$ to collapse (and for which height ℓ each one), in order to minimise the space occupancy of the resulting representation.

Our algorithm performs a post-order traversal of $\mathcal{T}$, starting from the root. Let $h(v)$ denote the height of the subtrie rooted at v (and reaching its descending leaves in $\mathcal{T}$). For each node v, the algorithm evaluates the cost of encoding the entire subtrie descending from v by taking into account the space cost $C(v_\ell)$ of Eq. (2) referring to the subtrie of v limited to height ℓ, plus the optimal space cost $C^*(d)$ of encoding recursively the entire subtries hanging from the nodes d descending from v at distance ℓ. We vary $\ell = 1, \ldots, h(v)$, thereby determining the

[2] We omit ceilings for the sake of simplicity.

minimum space occupancy $C^*(v)$. Formally, if $desc(v, \ell)$ is the set of descendants of v at distance ℓ (recall that $\ell \leq h(v)$), we have

$$C^*(v) = \min_{\ell=1,\ldots,h(v)} \left\{ C(v_\ell) + \sum_{d \in desc(v,\ell)} C^*(d) \right\}. \quad (3)$$

Note that if v is a leaf, we simply set $C^*(v) = C(v_1) = 2$, since a leaf cannot be collapsed and its cost in the LOUDS representation is 2 bits. Clearly, because of the post-order visit, the values $C^*(d)$ are available whenever we compute $C^*(v)$.

When the root of $\mathcal{T}$ is eventually visited, the topology and the encoding of all (macro-)nodes of our CoCo-trie have already been fully determined. Thus, we know which subtries to collapse and for which height ℓ, which may vary from one subtrie to another. By Eq. (3), the resulting data structure is the space-optimal one using the selected encoding scheme. The following result, proved in Appendix B, bounds the space-time efficiency of this approach.

Theorem 1. *The CoCo-trie of a given input trie $\mathcal{T}$ of height h and N nodes can be computed in $\mathcal{O}(Nh)$ time and $\mathcal{O}(N)$ space.*

We finally remark that in the above optimisation process we can upper bound the maximum number ℓ of collapsed levels so that the above time cost becomes $\mathcal{O}(N)$. This is actually the approach we take in our experimental section, where we bound ℓ for each node v by setting $h(v) = w/\log\sigma$ in Eq. (3), where w is the RAM word size in bits (see also Section 3.4). This feature may remind similar mechanisms adopted in ctrie [10, 45] and ctrie++ [46], where a subtrie is packed into a machine word. However, our approach is more powerful because the height of the subtrie to collapse is not chosen in advance and equal over the whole trie, but it is adaptively chosen on a single-node basis and in a data-aware manner according to the subtrie topology and the distribution of its edge labels.

3.3 A pool of succinct encoding schemes

Thus far, we represented the $m-1$ branching macro-characters $c^i_{v_\ell}$ of a macro-node v_ℓ via the EF-encoding of the increasing integers $c^i_{v_\ell} - x$, for $i = 2, \ldots, m$, where $x = c^1_{v_\ell}$ is the first branching character we stored explicitly. This sequence of $m-1$ macro-characters is drawn from a universe of size $u = c^m_{v_\ell} - c^1_{v_\ell} + 1$. Depending on m, u and the values of the branching macro-characters, it may be beneficial in time, in space, or both, to resort to other kinds of encodings.

On the grounds of this observation and inspired by the hybrid integer-encoding literature [12, 30, 41, 44], we now equip the CoCo-trie optimisation algorithm of the previous section with an assortment of encoding mechanisms so that the compressed representation of every single node can be chosen in a *data-aware* manner. This amounts to redefining the bit cost $C(v_\ell)$ of storing the macro-node v_ℓ so as to consider the cost in bits of other compression schemes besides EF. Specifically, when evaluating $C(v_\ell)$ during the traversal, whichever compression scheme gives the minimum bit-representation size for all collapsed subtries descending from v is selected and returned as the result of $C(v_\ell)$ (see Eq. (3)).

For our experimental study of Section 4, we follow [41] and, alongside EF, we adopt packed encoding (PA), characteristic bitvectors (BV), and dense encoding (DE). PA uses a fixed amount $\log u$ of bits for each $c^i_{v_\ell}$ for a total of $(m-1)\log u$ bits. BV uses u bits initially set to `0`, and then sets to `1` the $m-1$ bits corresponding to each $c^i_{v_\ell}$. DE comes into use whenever $u = m-1$, i.e. for representing a complete sequence of consecutive macro-characters; in this case, no additional bits are required. These encoding schemes allow to implement the predecessor search easily, as needed by the `rank` of CoCo-trie (see Section 3.6).

3.4 Squeezing the universe of branching labels

We now describe an optimisation to further decrease the space requirements for the macro-characters by means of an alphabet-aware encoding. The idea lies in replacing the encoding function enc_ℓ defined in Eq. (1) with a new one that depends on the alphabet of the branching macro-characters $c^i_{v_\ell}$ local to each macro-node v_ℓ rather than on the global alphabet Σ of the whole trie.

Let $\Sigma_{v_\ell} \subseteq \Sigma$ be the alphabet of symbols occurring in the edge labels of the collapsed macro-node v_ℓ. By changing $\sigma = |\Sigma|$ in Eq. (1) with $\sigma_{v_\ell} = |\Sigma_{v_\ell}|$, we can squeeze the size of the universe of the branching macro-characters of v_ℓ from σ^ℓ to $\sigma^\ell_{v_\ell}$. This, in turn, reduces the magnitude and the distance between consecutive integers associated with the branching macro-characters and thus allows a more effective compression. Also, we reduce the first space term of Eq. (2) to $\log \sigma^\ell_{v_\ell}$.

Clearly, each macro-node v_ℓ adopting this optimisation must store a mapping between Σ and the local alphabet Σ_{v_ℓ}, e.g., via a bitvector $B[0, \sigma-1]$ where $B[i] = 1$ if symbol i appears in Σ_{v_ℓ}.We observe this optimisation requires modifying $C(v_\ell)$ to account for both the more efficient macro-characters representation due to the squeezed universe and the size of the alphabet mapping (i.e. σ bits).

Overall, the time complexity for building the CoCo-trie becomes $\mathcal{O}(Nh^2)$ since we cannot compute u_ℓ incrementally as described in the proof of Theorem 1 (in Appendix B); the space complexity instead does not change, as we do not store the bitmaps $B_\ell[1, \sigma]$, but we compute them incrementally while visiting the subtries as in the proof of Theorem 1.

3.5 On the space-time trade-off

Under some scenarios, it might be of interest to slightly readjust the optimisation procedure to take into account query performance too, while possibly giving up the space optimality. To accomplish this space-time trade-off, we rely on the intuition that collapsing more levels generally improves the query time. As a matter of fact, the more levels are collapsed, the faster each trie traversal will be. But, on the other hand, as we collapse more levels, the fan-out of each macro-node increases and so the time to traverse each individual macro-node increases as well. However, we experimentally observed that this is not a major concern, since our compressed encoding of collapsed subtries and our succinct encoding schemes are in practice extremely efficient to be navigated; thus the

time reduction given by increasing the number of collapsed levels dominates the increased access time due to the bigger node fan-out.

With this in mind, we modify the algorithm of Section 3.2 as follows. At each visited internal node v, we compute $C^*(v)$ as usual and denote by ℓ^* the value of ℓ minimising the right-hand side of Eq. (3). Then, we find the largest value $\ell \in \{\ell^*, \ell^* + 1, \ldots, h(v)\}$ that allows to represent the collapsed node adding just a constant factor $\alpha \geq 0$ overhead over the optimal space $C^*(v)$. We observe this new approach has no impact on the construction complexity. We experiment with it in Section 4, where α is expressed as a percentage.

3.6 Trie operations

The `lookup`(P) in the CoCo-trie begins from the root macro-node r_ℓ, by computing the integer $x = enc_\ell(P[1, \ell]) - c^1_{r_\ell}$. Then, we seek for x into the increasing sequence $c^i_{r_\ell}$, for $i = 2, \ldots, m$: if the search fails, we return -1; otherwise, we obtain an index j of x, and proceed with the recursion in the j-th child of the macro-node. The compressed and indexed macro-node encoding guarantees the search for x is very efficient. We iteratively consume multiple characters at once from the pattern P as we descend the CoCo-trie downwards via LOUDS. When P is exhausted, we return the unique LOUDS index of the node we reach.

As for `rank`(P), we switch to the DFUDS encoding for the trie topology as it allows us to compute the rank of a leaf efficiently, takes the same space of LOUDS, and is still efficient in navigating the trie downwards [39, §8.3]. At each macro-node v_ℓ, we seek for the largest index j such that $c^j_{v_\ell} \leq x$, and we keep searching recursively into the j-th child of r_ℓ. Again, the EF and hybrid schemes for the $c^j_{v_\ell}$s result in fast branching operations. If P is exhausted at an internal node, the navigation proceeds downwards until the leftmost descendant of that node. In any case, we eventually reach a leaf node and return its rank.

4 Experiments

We run our experiments on a machine equipped with a 2.30 GHz Intel Xeon Platinum 8260M CPU and 384 GiB of RAM, running Ubuntu 20.04.3. We compile our codebase using `g++11.1` and the `C++-20` language standard.

Datasets. We aimed at choosing very diverse datasets in terms of sources (such as the Web, bioinformatics, and databases) and features (such as the number n of strings, total number D of characters, alphabet size σ, and average/maximum length of the lcp between consecutive sorted strings) to depict a broader spectrum of the performances of the tested string dictionaries. We preprocess each dataset to keep, for each string s, the shortest prefix of s that distinguishes it from all the other strings in the dataset. It is well known that, for any trie-based structure, the remaining suffixes can be concatenated into a separate array and efficiently retrieved when needed [39, §8.5.3]. Table 1 shows the datasets and their characteristics after preprocessing.

Table 1. Datasets characteristics.

Name	Description	$n/10^6$	$D/10^6$	Avg lcp	Avg length	Max length	σ
`url`	URLs crawled from the web [13]	40.5	2 713.5	64.1	66.9	1 990	94
`xml`	rows of an XML dump of dblp [3]	2.9	107.8	34.4	36.5	248	95
`protein`	different sequences of amino acids [3]	2.9	155.6	36.7	53.3	16 191	26
`dna`	unique 12-mer from a DNA seq. [3]	13.7	164.5	10.5	11.9	12	15
`tpcds-id`	customers ids in TPC-DS-3TB [38]	30.0	446.4	13.4	14.8	15	16

State-of-the-art competitors. We consider as competitors of our CoCo-trie the following static string dictionaries implementations because they are either the state of the art or offer efficient approaches to compact trie representations:

CART: a compact version of ART [32] obtained by constructing a plain ART and converting it to a static version [1, 49, 50].

PDT: the Centroid Path Decomposed Trie [24]. We experiment with both the vbyte version that encodes the labels of the edges with vbyte [47], and the csp version that adds another layer of compression on top of the edge labels.

FST: the Fast Succinct Trie [50]. We use a slightly-modified code [2] that solves lookup queries rather than range query filtering. We show the full space-time performance of FST by varying its parameter R as 2^i for $i = 0, \ldots, 10$.

Apart from the above static data structures, we also tested ART [32] and ctrie++ [46] as representative of the dynamic approaches. In our figures we do not show ctrie++ since its space usage on our datasets is from 2.8 to 6.2× larger than ART, which in turn uses up to one order of magnitude more space than the other tested solutions, and since it is faster than ART only on `url` (by 14%).

We do not experiment with Masstree [34] because [11] shows it uses from 1.8 to 3× more space than ART. We also do not experiment with HOT [11] because their implementation only supports strings shorter than 256, while our datasets contain much longer strings. Finally, we do not experiment with the implementation provided in [14, 35] as we were unable to run its codebase in a fair environment due to some old software dependencies and incompatibilities with modern compilers.

Query workloads. Given that our competitors do not implement `rank` (despite their design does support it), we decided to measure the performance of `lookup`. We can reasonably expect that `rank` would perform similarly to `lookup`, because of the way the former can be derived from the latter in trie-based (rather than hash-based) data structures, as the ones we experimentally test here.

Given a dataset of n strings, we measure the query time by averaging the performance of 3 repetitions of a batch of size n, where half of the strings are taken from the datasets and half are generated randomly. To generate each of these latter strings, we (i) extract a randomly-chosen string belonging to the dataset

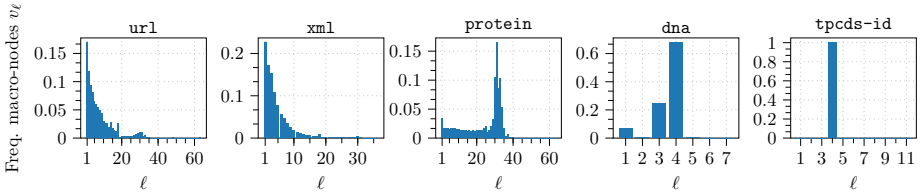

Fig. 3. Normalised frequency of macro-nodes collapsing subtries having ℓ levels.

and truncate it to the average lcp of the entire dataset, and (ii) append a random string whose length matches the average length of the strings in the dataset. This way, the queries we generate mimic a fair query workload that guarantees a balance between existent and not existent queried strings, and for the latter that the traversal does not stop at the very first steps because of a mismatch.

Experimental results. Figure 3 shows our first experimental result: the number of macro-nodes collapsing a certain amount of levels forms a non-trivial distribution whose shape differs from dataset to dataset. This provides a clear answer to both questions **Q2** and **Q3** in Section 3: the number of levels to be collapsed in a subtrie greatly depends on the strings the trie is built on, and it must be chosen locally. Therefore, the data-aware approach to subtrie compaction implemented in our CoCo-trie optimiser is essential to attain the most from these features.

In particular, on `url` and `xml`, the CoCo-trie optimiser selects many times the lowest possible values of ℓ (each horizontal axis ranges from $\ell = 1$ to the largest ℓ over all macro-nodes v_ℓ). For `protein`, instead, the CoCo-trie optimiser selects high values of ℓ (very often $\ell \approx 30$) so that, in the end, the distribution resembles a Gaussian one. On `dna`, CoCo-trie optimiser collapses at most $\ell = 7$ levels at a time and selects $\ell = 4$ for 67% of the times. The results on `tpcds-id` are also of interest for their simplicity: due to the regularity of the dataset, the CoCo-trie optimiser here creates a macro-node for the root that collapses $\ell = 11$ levels, and each of its 4096 children collapses $\ell = 4$ levels.

Figure 4 shows the results of the space and time performance of CoCo-trie and the five competitors.

Firstly, we observe that ART and CART, though fast, are generally very space-demanding (note the vertical axis is logarithmic). On `url`, they are also slower due to their large size and the high average lcp among the dictionary strings, which causes longer trie traversals and thus more cache misses.

FST is dominated in space and time performance by our CoCo-trie (and also by other data structures) on all the datasets. This is especially evident on `url`, `xml`, and `protein`. We argue that this is due to the high average lcp of these datasets that require FST to perform longer trie traversals that proceed one character at a time (indeed, FST does not compact unary paths).

PDT shows overall a good space-time performance, with the exception of `dna` and `tpcds-id`, for which the average height of the PDT nodes is 6.0 and 8.7,

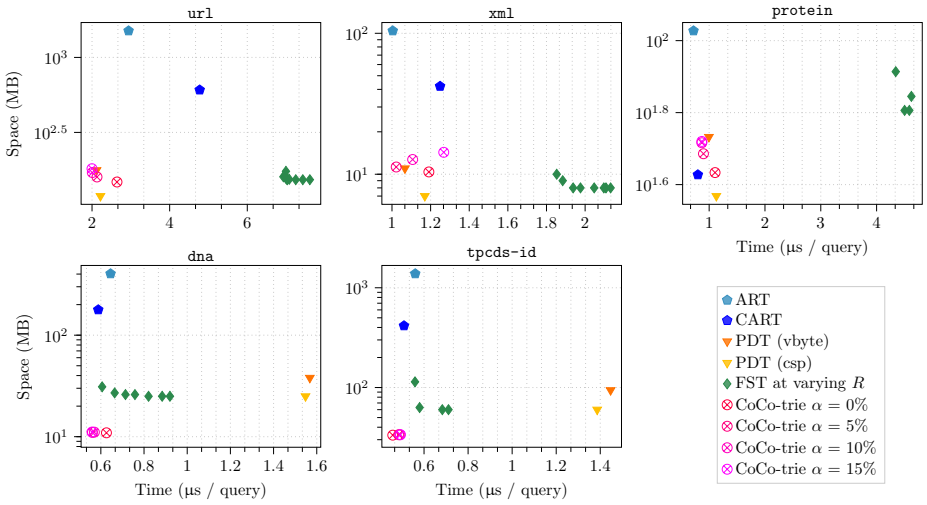

Fig. 4. Space-time performance of the lookup query for various tested approaches. Note that the y-axis is logarithmic and that the ranges of both axes change among the plots.

respectively. The average height of the macro-nodes in the CoCo-trie (see the $\alpha = 0\%$ configuration in Figure 4) is instead 3.5 and 4.0, thus requiring nearly half accesses to nodes, on average. Indeed, CoCo-trie is 2.3× faster and 3× more succinct than PDT on `dna`, and it is 2.6× faster and 2.4× more succinct than PDT on `tpcds-id`. This means that, on these two datasets, the approach of exploiting the small local alphabet at each macro-node is particularly effective. On the rest of the datasets, CoCo-trie always has some configuration on the Pareto frontier of PDT, thus offering other competitive space-time trade-offs.

In summary, with respect to the highly-engineered competitors we test on the diverse five datasets, the CoCo-trie results space-time efficient, robust and flexible: in fact, it significantly dominates the space-time performance of all competitors on `dna` and `tpcds-id`; it is on the Pareto frontier of the best competitors over `url` and `xml`; and, lastly, it is very close to the Pareto frontier for `protein`.

5 Conclusions and Future Work

We have introduced a new design of compressed string dictionaries that collapses and succinctly encodes properly-chosen subtries via novel data-aware encoding of (possibly long) edge labels and a space optimisation procedure. The experimental results over a variety of datasets and highly-engineered competitors suggest that our CoCo-trie does advance the state of the art of string dictionaries.

The novel design scheme on which CoCo-trie hinges, paves the way to further new approaches for the *multicriteria* optimisation of trie data structures that take into account possibly other encoding schemes and different multi-objective functions (e.g., over time, space, energy usage, etc.).

Acknowledgements. Supported by the Italian MUR PRIN project "Multi-criteria data structures and algorithms" (Prot. 2017WR7SHH), and by the EU H2020 projects "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (grant #871042) and "HumanE AI Network" (grant #952026).

References

1. ART and CART implementations, `https://github.com/efficient/fast-succinct-trie/tree/master/third-party/art`, accessed: June 2022
2. FST implementation, `https://github.com/kampersanda/fast_succinct_trie`, accessed: June 2022
3. Pizza&Chili corpus, `http://pizzachili.dcc.uchile.cl/texts.html`, accessed: June 2022
4. Acharya, A., Zhu, H., Shen, K.: Adaptive algorithms for cache-efficient trie search. In: Proc. International Workshop on Algorithm Engineering and Experimentation (ALENEX). pp. 300–315 (1999). https://doi.org/10.1007/3-540-48518-X_18
5. Apostolico, A., Crochemore, M., Farach-Colton, M., Galil, Z., Muthukrishnan, S.: 40 years of suffix trees. Communications of the ACM **59**(4), 66–73 (mar 2016). https://doi.org/10.1145/2810036
6. Arz, J., Fischer, J.: Lempel-ziv-78 compressed string dictionaries. Algorithmica **80**(7), 2012–2047 (2018). https://doi.org/10.1007/s00453-017-0348-7
7. Askitis, N., Sinha, R.: Engineering scalable, cache and space efficient tries for strings. The VLDB Journal **19**(5), 633–660 (2010). https://doi.org/10.1007/s00778-010-0183-9
8. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-oblivious String B-trees. In: Proc. 25th ACM Symposium on Principles of Database Systems (PODS). pp. 233–242 (2006). https://doi.org/10.1145/1142351.1142385
9. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 360–369 (1997). https://doi.org/10.5555/314161.314321
10. Bille, P., Gørtz, I.L., Skjoldjensen, F.R.: Deterministic indexing for packed strings. In: Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM). vol. 78, pp. 6:1–6:11 (2017). https://doi.org/10.4230/LIPIcs.CPM.2017.6
11. Binna, R., Zangerle, E., Pichl, M., Specht, G., Leis, V.: HOT: a height optimized trie index for main-memory database systems. In: Proc. ACM International Conference on Management of Data (SIGMOD). pp. 521–534 (2018). https://doi.org/10.1145/3183713.3196896
12. Boffa, A., Ferragina, P., Vinciguerra, G.: A learned approach to design compressed rank/select data structures. ACM Transactions on Algorithms (2022). https://doi.org/10.1145/3524060
13. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. 13th International World Wide Web Conference (WWW). pp. 595–601 (2004), `https://law.di.unimi.it/webdata/it-2004/`
14. Brisaboa, N.R., Cerdeira-Pena, A., de Bernardo, G., Navarro, G.: Improved compressed string dictionaries. In: Proc. 28th ACM International Conference on Information and Knowledge Management (CIKM). p. 29–38 (2019). https://doi.org/10.1145/3357384.3357972

15. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Software Practice and Experience **23**(3), 277–291 (1993). https://doi.org/10.1002/spe.4380230305
16. Elias, P.: Efficient storage and retrieval by content and address of static files. Journal of the ACM **21**(2), 246–260 (1974). https://doi.org/10.1145/321812.321820
17. Fano, R.M.: On the number of bits required to implement an associative memory. Memo 61. Massachusetts Institute of Technology, Project MAC (1971)
18. Ferragina, P., Grossi, R.: The String B-tree: a new data structure for string search in external memory and its applications. Journal of the ACM **46**(2), 236–280 (Mar 1999). https://doi.org/10.1145/301970.301973
19. Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: Proc. 27th ACM Symposium on Principles of Database Systems (PODS). pp. 181–190 (2008). https://doi.org/10.1145/1376916.1376943
20. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM **57**(1), 4:1–4:33 (2009). https://doi.org/10.1145/1613676.1613680
21. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Transactions on Algorithms **7**(1), 10:1–10:21 (2010). https://doi.org/10.1145/1868237.1868248
22. Ferragina, P., Venturini, R.: Compressed cache-oblivious String B-tree. ACM Transactions on Algorithms **12**(4), 52:1–52:17 (2016). https://doi.org/10.1145/2903141
23. Gog, S., Pibiri, G.E., Venturini, R.: Efficient and effective query autocompletion. In: Proc. 43rd ACM International Conference on Research and Development in Information Retrieval (SIGIR). pp. 2271–2280 (2020). https://doi.org/10.1145/3397271.3401432
24. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. ACM Journal of Experimental Algorithmics **19**(1) (2014). https://doi.org/10.1145/2656332, `https://github.com/ot/path_decomposed_tries`
25. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM Transactions on Information Systems **20**(2), 192–223 (2002). https://doi.org/10.1145/506309.506312
26. Hsu, B.J.P., Ottaviano, G.: Space-efficient data structures for top-k completion. In: Proc. 22nd International Conference on World Wide Web (WWW). p. 583–594 (2013). https://doi.org/10.1145/2488388.2488440
27. Huston, S.J., Moffat, A., Croft, W.B.: Efficient indexing of repeated n-grams. In: Proc. 4th International Conference on Web Search and Web Data Mining (WSDM). pp. 127–136 (2011). https://doi.org/10.1145/1935826.1935857
28. Kanda, S., Köppl, D., Tabei, Y., Morita, K., Fuketa, M.: Dynamic path-decomposed tries. ACM Journal of Experimental Algorithmics **25**, 1–28 (2020). https://doi.org/10.1145/3418033
29. Kang, Y.M., Liu, W., Zhou, Y.: QueryBlazer: efficient query autocompletion framework. In: Proc. 14th International Conference on Web Search and Data Mining (WSDM). pp. 1020–1028 (2021). https://doi.org/10.1145/3437963.3441725
30. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Hybrid compression of bitvectors for the FM-index. In: Proc. 24th Data Compression Conference (DCC). pp. 302–311 (2014). https://doi.org/10.1109/DCC.2014.87
31. Knuth, D.E.: The art of computer programming, vol. 3. Addison-Wesley, 2 edn. (1998)

32. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. 29th IEEE International Conference on Data Engineering (ICDE). pp. 38–49 (2013). https://doi.org/10.1109/ICDE.2013.6544812
33. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design. Cambridge University Press (2015)
34. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proc. 7th European Conference on Computer Systems (EuroSys). pp. 183–196 (2012). https://doi.org/10.1145/2168836.2168855
35. Martínez-Prieto, M.A., Brisaboa, N.R., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Information Systems **56**, 73–108 (2016). https://doi.org/10.1016/j.is.2015.08.008
36. Mavlyutov, R., Wylot, M., Cudré-Mauroux, P.: A comparison of data structures to manage URIs on the web of data. In: Proc. 12th European Semantic Web Conference (ESWC). pp. 137–151 (2015). https://doi.org/10.1007/978-3-319-18818-8_9
37. Morrison, D.R.: PATRICIA—practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM **15**(4), 514–534 (Oct 1968). https://doi.org/10.1145/321479.321481
38. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: Proc. 32nd International Conference on Very Large Data Bases (VLDB). p. 1049–1058 (2006), `http://www.tpc.org/tpcds/`
39. Navarro, G.: Compact data structures: a practical approach. Cambridge University Press (2016)
40. Nilsson, S., Tikkanen, M.: Implementing a dynamic compressed trie. In: Proc. 2nd International Workshop on Algorithm Engineering (WAE). pp. 25–36 (1998)
41. Ottaviano, G., Venturini, R.: Partitioned Elias-Fano indexes. In: Proc. 37th ACM International Conference on Research and Development in Information Retrieval (SIGIR). pp. 273–282 (2014). https://doi.org/10.1145/2600428.2609615
42. Pibiri, G.E., Venturini, R.: Efficient data structures for massive n-gram datasets. In: Proc. 40th ACM International Conference on Research and Development in Information Retrieval (SIGIR). p. 615–624 (2017). https://doi.org/10.1145/3077136.3080798
43. Poyias, A., Puglisi, S.J., Raman, R.: m-Bonsai: a practical compact dynamic trie. International Journal of Foundations of Computer Science **29**(8), 1257–1278 (2018). https://doi.org/10.1142/S0129054118430025
44. Silvestri, F., Venturini, R.: VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In: Proc. 19th ACM International Conference on Information and Knowledge Management (CIKM). pp. 1219–1228 (2010). https://doi.org/10.1145/1871437.1871592
45. Takagi, T., Inenaga, S., Sadakane, K., Arimura, H.: Packed compact tries: a fast and efficient data structure for online string processing. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **100-A**(9), 1785–1793 (2017). https://doi.org/10.1587/transfun.E100.A.1785
46. Tsuruta, K., Köppl, D., Kanda, S., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: c-Trie++: a dynamic trie tailored for fast prefix searches. In: Proc. 30th Data Compression Conference (DCC). pp. 243–252 (2020). https://doi.org/10.1109/DCC47342.2020.00032
47. Williams, H.E., Zobel, J.: Compressing integers for fast file access. The Computer Journal **42**(3), 193–201 (1999). https://doi.org/10.1093/comjnl/42.3.193
48. Yata, S.: Dictionary compression by nesting prefix/patricia tries. In: Proc. 17th Meeting of the Association for Natural Language (2011)

49. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In: Proc. ACM International Conference on Management of Data (SIGMOD). p. 1567–1581 (2016). https://doi.org/10.1145/2882903.2915222
50. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: SuRF: practical range query filtering with fast succinct tries. In: Proc. ACM International Conference on Management of Data (SIGMOD). pp. 323–336 (2018). https://doi.org/10.1145/3183713.3196931

A Calculations for the motivating example of Section 2

We first recall that the γ-code of a positive integer x consists of a number of $\texttt{0}$s equal to the number of bits minus one of the binary representation of x, followed by that binary representation, e.g. $\gamma(6) = \texttt{00110}$. Thus, $\gamma(x)$ takes $2\lfloor \log_2 x \rfloor + 1$ bits.

The case of trie $\mathcal{A}$. The succinct representation of the edge labels in $\mathcal{A}$ takes $3 + 3 + 3 = 9$ bits. In fact, the encoding of the edge labels $\{\mathrm{A}, \mathrm{C}\}$ of the root is $enc(\mathrm{A})\,\gamma(\mathrm{C} - \mathrm{A}) = \texttt{00}\,\gamma(\texttt{1}) = \texttt{001}$, then the encoding of the edge labels $\{\mathrm{G}, \mathrm{T}\}$ of the first node at the second level is $enc(\mathrm{G})\,\gamma(\mathrm{G} - \mathrm{T}) = \texttt{10}\,\gamma(\texttt{1}) = \texttt{101}$, and finally the encoding of the edge labels $\{\mathrm{A}, \mathrm{C}\}$ (again) of the second node at the second level is $enc(\mathrm{A})\,\gamma(\mathrm{C} - \mathrm{A}) = \texttt{001}$.

If instead, we collapse the two levels of $\mathcal{A}$ in the root of $\mathcal{A}^c$, this root obtains four children whose edge labels are $\{\mathrm{AG}, \mathrm{AT}, \mathrm{CA}, \mathrm{CC}\}$, and their succinct representation takes 7 bits. We indeed encode the first branching macro-symbol $enc(\mathrm{AG}) = \texttt{0010}$ as it is, followed by the encoding of the other three branching macro-symbols as: $\gamma(\mathrm{AT} - \mathrm{AG}) = \gamma(\texttt{0011} - \texttt{0010}) = \gamma(\texttt{1}) = \texttt{1}$, $\gamma(\mathrm{CA} - \mathrm{AT}) = \gamma(\texttt{0100} - \texttt{0011}) = \gamma(\texttt{1}) = \texttt{1}$, and $\gamma(\mathrm{CC} - \mathrm{CA}) = \gamma(\texttt{0101} - \texttt{0100}) = \gamma(\texttt{1}) = \texttt{1}$.

Thus in terms of space cost, $\mathcal{A}$ is worse than $\mathcal{A}^c$. This result is even more evident when accounting for the space cost for the topology, simply because $\mathcal{A}$ has more nodes than $\mathcal{A}^c$. We conclude that, under this setting, it is better to collapse the trie and keep $\mathcal{A}^c$.

The case of trie $\mathcal{B}$. Surprisingly, one comes to the opposite conclusion with $\mathcal{B}$, despite having the same topology of $\mathcal{A}$. Here, the larger alphabet together with the different distribution of the edge labels changes the optimal choice.

The succinct representation of the edge labels in $\mathcal{B}$ takes at most $5b + 1$ bits. We can indeed represent the edge labels $\{\mathrm{A}, \xi\}$ of the root with $enc(\mathrm{A})\,\gamma(|\Sigma| - 1)$ which takes at most $3b - 1$ bits; the root gets followed by the encoding of the edge labels $\{\mathrm{A}, \xi\}$ of the first node at the second level, namely $enc(\mathrm{A})\,\gamma(\texttt{1})$ which takes $b + 1$ bits, and by the encoding of the edge labels $\{\xi', \xi\}$ of the second node at the second level, which is $enc(\xi')\,\gamma(\texttt{1})$ which takes $b + 1$ bits.

Conversely, the succinct representation of $\mathcal{B}^c$ may take up to $6b - 1$ bits, since we encode AA with $2b$ bits set to 0, followed by $\gamma(\mathrm{AC} - \mathrm{AA}) = \gamma(\texttt{1}) = \texttt{1}$, then by $\gamma(\xi\xi' - \mathrm{AC}) = \gamma(\texttt{01}\ldots\texttt{101})$ (which takes $4b - 3$ bits, because the γ-encoded number consists of $2b - 1$ bits), and finally by $\gamma(\xi\xi - \xi\xi') = \gamma(\texttt{1}) = \texttt{1}$.

Hence, differently from the example on $\mathcal{A}$, here it is better not to collapse $\mathcal{B}$ because its succinct encoding takes $b-2$ bits less than the one of $\mathcal{B}^c$, and b can make this gap arbitrarily large, up to the point that the cost of representing their topology becomes negligible.

B Proof of Theorem 1

Starting from a node v of height $h(v)$, we can compute $C(v_\ell)$ for any $\ell = 1, 2, \ldots, h(v)$ by obtaining incrementally all the optimisation parameters u_ℓ and m_ℓ from the already (inductively) known $u_{\ell-1}$ and $m_{\ell-1}$.

To compute the universe size u_ℓ for v_ℓ we need to determine the enc_ℓ-code of the leftmost and rightmost length-ℓ strings descending from v, and these can be computed by extending the respective $enc_{\ell-1}$-codes computed at the previous step with one character, in constant time. This costs overall $\mathcal{O}(Nh)$ time because, for each node, we have to visit the leftmost and rightmost branching strings that are of length at most h.

To compute m_ℓ (i.e. the number of children of the collapsed macro-node v_ℓ), we need to visit once the whole subtrie rooted at v. Knowing $m_{\ell-1}$, we add to it the number of leaves at the ℓ-th level. Performing for every node v a complete visit of its whole subtrie costs overall $\mathcal{O}(Nh)$ time: indeed, each of the N nodes has at most h different ancestors and thus belongs to at most h different subtries, thereby getting visited at most h times.

For every node v we maintain just the optimal C^*-cost, thus the required space amounts to $\mathcal{O}(N)$.