Improved Compressed String Dictionaries*

Nieves R. Brisaboa
brisaboa@udc.es
Universidade da Coruña,
Centro de investigación CITIC, Databases Lab.
A Coruña, Spain

Ana Cerdeira-Pena
acerdeira@udc.es
Universidade da Coruña,
Centro de investigación CITIC, Databases Lab.
A Coruña, Spain

Guillermo de Bernardo
gdebernardo@udc.es
Universidade da Coruña,
Centro de investigación CITIC, Databases Lab.
A Coruña, Spain

Gonzalo Navarro
gnavarro@dcc.uchile.cl
IMFD, DCC, University of Chile
Santiago, Chile

ABSTRACT

We propose a new family of data structures to store and query string dictionaries in main memory. We combine hierarchical Front-coding with ideas from longest-common-prefix computation in suffix arrays. We focus on two application domains where string dictionaries are extensively used: URL collections, used in Web graphs, and collection of URIs and literals used in RDF datasets. We test our proposals in real-world dictionaries, showing that our data structures yield relevant space-time tradeoffs, achieving very good compression and competitive query times.

CCS CONCEPTS

• **Information systems** → **Data compression**; **Dictionaries**.

KEYWORDS

compression, data structures, string dictionaries

1 INTRODUCTION

String dictionaries, that map strings to unique identifiers, are widely used in applications that must work with large collections of strings. For instance, in Web graphs or RDF datasets, a usual technique is to associate consecutive integers to each node label, and then build a representation of the graph structure that uses those integer identifiers (ids) instead of the original strings. Then, queries are solved by translating query strings to ids, executing the query on ids and mapping the results back to strings. The two basic operations needed are *lookup(s)* , that receives a string and returns the string id, and *access(i)* that returns the string from the id.

In this work we focus on the compact representation of URL dictionaries, used in Web graphs, and URIs and literals dictionaries, used in RDF datasets. We combine well-known techniques such as Front-coding or Re-pair compression with adapted binary-search algorithms to obtain a family of compressed string dictionaries with good performance in those applications.

Several solutions have been proposed for this problem. Martinez-Prieto et al. [4] developed, among other proposals, a solution based on a fixed partition of the strings and compression of each block with Front-coding and other techniques; their solution provides a wide space/time tradeoff varying the block size. Grossi and Ottaviano proposed path-decomposed tries (PDT) [3], a compact trie representation that achieves good compression and very consistent query times.

2 OUR PROPOSAL

Our techniques follow the same ideas of differential compression based on Front-coding proposed in previous work [4], but we design a new arrangement that aims at improving the space-time tradeoff of previous solutions based on sampling and linear search.

Our representation starts from a collection of strings, that are lexicographically sorted. We add two markers at the limits of the collection, and initialize two arrays $llcp$ and $rlcp$, that will be set to 0 for those markers. Then, we select the middle point m of our array, set the value $llcp[m]$ to the longest common prefix (lcp) between the string at the middle and the left limit, and $rlcp[m]$ to the lcp between the middle and the rightmost string. Then, we recursively repeat the process in each half of the collection, until all lcps have been computed. We remove from each string the prefix corresponding to the maximum of its two associated lcps. Figure 1 displays the resulting elements for a sample collection of strings. Our representation consists of the arrays $llcp$ and $rlcp$, and the tails of the strings after removing the prefixes.

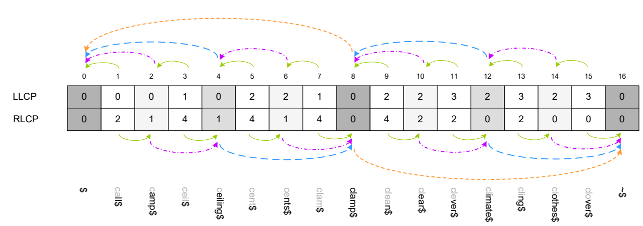

Figure 1: Conceptual dictionary structure.

Using those structures, to look up a string, we could binary search our collection and compare the query string with the string tail at the midpoint at each step. We have instead devised an improved binary search procedure that, keeping track of lcp values, is able to save some string comparisons. To recover the string for an id, we reverse the procedure used in lookups. We recover the

*This extended abstract summarizes work previously published in CIKM 2019 [1]
"Copyright ©2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)."

string from the end, starting at the given position and moving to the left/right "parent" to keep decoding new characters, until we reach a position with lcp 0 and hence the string decoding is complete. The complete pseudocode for both algorithms can be found in the full article.

We call our basic proposal IBiS, but we have proposed and tested a number of implementation variants for the same conceptual representation, varying the data structure and compression techniques applied to $llcp$, $rlcp$ and the string tails S: IBiS stores the arrays as sequences of fixed-size integers, and the string tails are stored in a single string Str. We add a bitmap B that marks the beginning of each string tail in Str. IBiS^{RP} applies Re-Pair compression to Str and encodes the resulting integer sequence as an array. IBiS^{RP+DAC} is similar to IBiS^{RP}, but uses DACs [2] to store $llcp$ and $rlcp$. IBiS$^{RP+DAC-VLS}$ is also similar to IBiS^{RP} but uses another variant of DACs to store the array resulting from Re-Pair compression. IBiS$^{RP+DAC+DAC-VLS}$ combines the two previous approaches, using DACs to $llcp$, $rlcp$ and the Re-Pair-compressed strings.

We also tested other variants that provide interesting tradeoffs. First, if we consider end-of-string markers in Str to locate the end of each string we can speed up some comparisons, but we have to store an extra byte per string; we can omit those to save significant space, at the cost of slightly more complex searches. We also build variants that use a single lcp array ($llcp$ or $rlcp$) instead of two; in those variants, we avoid storing one of the arrays, but our string tails are longer, and search operations may not save as many string comparisons, so a space-time tradeoff is obtained.

3 EXPERIMENTAL EVALUATION

We tested our proposal with Web graph and RDF datasets. We summarize here the full results, that can be found on the full paper. We compare our results with previous solutions [4] based on Front-coding (RPFC and RPHTFC) and binary search (RPDAC and HASHRPDAC), and with path-decomposed tries (PDT).

Our preliminary results suggest a number of trends among our variants. For instance, among single-lcp implementations, those using only $llcp$ ($-L$) are consistently the most efficient, so we will show detailed results for those. Variants with no end-of-string markers ($-nt$) also obtain the best performance in general, since query times are slightly affected but compression improves significantly, so we will restrict our explanation to them.

Figure 2 shows the space/time tradeoff on the Web graph dataset UK. Results in other datasets follow similar trends: our techniques achieve the best compression and are competitive in query times with most alternatives. Techniques like PDT or RPDAC are faster, but significantly larger in most cases; previous solutions based on Front-coding provide a wide tradeoff, but our variants can provide similar tradeoffs and achieve better compression for similar query times. Additionally, our variants provide different tradeoffs for lookup and access that could be useful in specific applications.

4 CONCLUSIONS

Our new family of compressed data structures can efficiently compress string dictionaries and provide competitive query times. Our

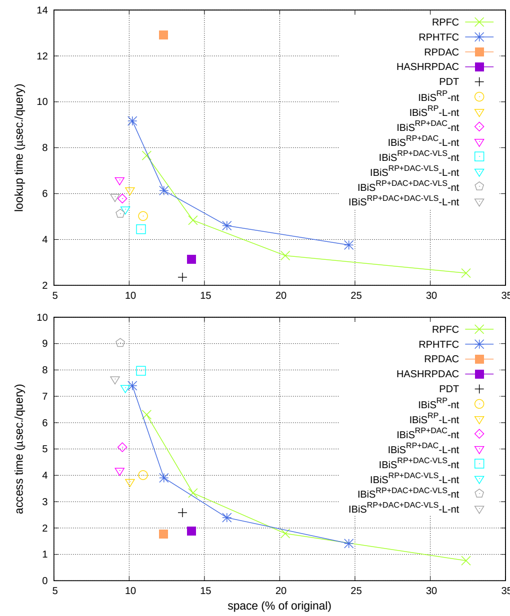

Figure 2: Space and query times on Web graph UK

evaluation with Web graphs and RDF data suggests that our techniques improve the compression obtained by state-of-the-art solutions. Moreover, the variants tested provide several interesting space/time tradeoffs for compression, lookup and access performance. Our proposal is so far limited to a static scenario, and works well in datasets with relatively long strings on average. We plan to explore the possibilities to adapt our solutions to other types of string dictionaries, as well as to the dynamic dictionary problem.

REFERENCES

[1] Nieves R. Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, and Gonzalo Navarro. 2019. Improved Compressed String Dictionaries. In *Proc. 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM'19)*. 29–38. https://doi.org/10.1145/3357384.3357972

[2] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2013. DACs: Bringing Direct Access to Variable-length Codes. *Information Processing and Management* 49, 1 (Jan. 2013), 392–404. https://doi.org/10.1016/j.ipm.2012.08.003

[3] Roberto Grossi and Giuseppe Ottaviano. 2015. Fast Compressed Tries Through Path Decompositions. *Journal of Experimental Algorithmics* 19, Article 3.4 (Jan. 2015), 11 pages. https://doi.org/10.1145/2656332

[4] Miguel A. Martínez-Prieto, Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. 2016. Practical Compressed String Dictionaries. *Information Systems* 56, C (Mar. 2016), 73–108. https://doi.org/10.1016/j.is.2015.08.008