Balancing Straight-Line Programs for Strings and Trees

Markus Lohrey(✉)

University of Siegen, Siegen, Germany
lohrey@eti.uni-siegen.de

Abstract. The talk will explain a recent balancing result according to which a context-free grammar in Chomsky normal form of size m that produces a single string w of length n (such a grammar is also called a straight-line program) can be transformed in linear time into a context-free grammar in Chomsky normal form for w of size $\mathcal{O}(m)$, whose unique derivation tree has depth $\mathcal{O}(\log n)$. This solves an open problem in the area of grammar-based compression, improves many results in this area and greatly simplifies many existing constructions. Similar balancing results can be formulated for various grammar-based tree compression formalism like top DAGs and forest straight-line programs. The talk is based on joint work with Moses Ganardi and Artur Jeż. An extended abstract appeared in [11]; a long version of the paper can be found in [12].

1 Grammar-Based String Compression

In *grammar-based compression* a combinatorial object (e.g. a string or tree) is compactly represented using a grammar of an appropriate type. In many grammar-based compression formalisms such a grammar can be exponentially smaller than the object itself. A well-studied example of this general idea is grammar-based string compression using context-free grammars that produce only one string each, which are also known as *straight-line programs*, SLPs for short. Formally, we define an SLP as a context-free grammar $\mathcal{G}$ in Chomsky normal form such that (i) $\mathcal{G}$ is acyclic and (ii) for every nonterminal A there is exactly one production, where A is the left-hand side. The *size* $|\mathcal{G}|$ of the SLP $\mathcal{G}$ is the number of nonterminals in $\mathcal{G}$ and the *depth* of $\mathcal{G}$ (depth($\mathcal{G}$) for short) is the height of the unique derivation tree of $\mathcal{G}$. Since we assume that $\mathcal{G}$ is in Chomsky normal form, we have depth($\mathcal{G}$) $\geq \log n$ if n is the length of the string produced by $\mathcal{G}$.

The goal of grammar-based string compression is to compute from a given string an SLP of small size. Grammar-based string compression is tightly related to dictionary-based compression: the famous LZ78 algorithm can be viewed as a particular grammar-based compressor. Moreover, the number of phrases in the LZ77-factorization is a lower bound for the smallest SLP for a string [24], and an LZ77-factorization of length m can be converted to an SLP of size $\mathcal{O}(m \cdot \log n)$

M. Anselmo et al. (Eds.): CiE 2020, LNCS 12098, pp. 296–300, 2020.
https://doi.org/10.1007/978-3-030-51466-2_26

where n is the length of the string [8,18,19,24]. For various other aspects of grammar-based string compression see [8,20].

2 Balancing String Straight-Line Programs

An advantage of grammar-based compression is that SLPs are well-suited for further algorithmic processing. There is an extensive body of literature on algorithms for SLP-compressed strings, see e.g. [20] for a survey. For many of these algorithms, the depth of the input SLP is an important parameter. Let us give a simple example: in the *random access problem* for an SLP-compressed string s an SLP $\mathcal{G}$ for s is given. Let n be the length of s. The goal is to produce from $\mathcal{G}$ a data structure that allows us to compute for a given position i ($1 \leq i \leq n$) the i-th symbol of s. As observed in [6] one can solve this problem in time $\mathcal{O}(\text{depth}(\mathcal{G}))$ (assuming arithmetic operations on numbers from the interval $[0, n]$ need constant time): in the preprocessing phase one computes for every nonterminal A of $\mathcal{G}$ the length n_A of the string produced from A; this takes time $\mathcal{O}(\mathcal{G})$ and produces a data structure of size $\mathcal{O}(\mathcal{G})$ (assuming a number from $[0, n]$ fits into a memory location). In the query phase, one computes for a given position $i \in [1, n]$ the path from the root to the i-th leaf in the derivation tree of $\mathcal{G}$. Only the current nonterminal A and a relative position in the string produced from A has to be stored. Using the pre-computed numbers n_A the whole computation needs time $\mathcal{O}(\text{depth}(\mathcal{G}))$. Recall that $\text{depth}(\mathcal{G}) \geq \log n$. In [6] it is shown that one can compute from $\mathcal{G}$ a data structure of size $\mathcal{O}(|\mathcal{G}|)$ which allows to access every position in time $\mathcal{O}(\log n)$, irrespective of the depth of $\mathcal{G}$. The algorithm in [6] is quite complicated and used several sophisticated data structures. An alternative approach to obtain access time $\mathcal{O}(\log n)$ is to *balance* the input SLP $\mathcal{G}$ in the preprocessing phase, i.e., to reduce its depth. This is the approach that we will follow.

It is straightforward to show that any string s of length n can be produced by an SLP of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$. A more difficult problem is to balance a given SLP: assume that the SLP $\mathcal{G}$ produces a string of length n. Several authors have shown that one can restructure $\mathcal{G}$ in time $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ into an equivalent SLP $\mathcal{H}$ of size $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ and depth $\mathcal{O}(\log n)$ [8,19,24]. Applied to the random access problem, these balancing procedures would yield access time $\mathcal{O}(\log n)$ at the cost of building a data structure of size $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ during the preprocessing. Our main result shows that SLP balancing is in fact possible with a constant blow-up in SLP-size:

Theorem 1. *Given an SLP $\mathcal{G}$ producing a string of length n one can construct in linear time an equivalent SLP $\mathcal{H}$ of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\log n)$.*

As a corollary we obtain a very simple and clean algorithm for the random access problem with access time $\mathcal{O}(\log n)$ that uses a data structure of size $\mathcal{O}(m)$ (in words of bit length $\log n$). We can also obtain an algorithm for the random access problem with access time $\mathcal{O}(\log n / \log \log n)$ using a data structure with $\mathcal{O}(m \cdot \log^{\epsilon} n)$ words for any $\epsilon > 0$; previously this bound was only shown for

SLPs of height $\mathcal{O}(\log n)$. [1] The paper [11] contains a list of further applications of Theorem 1, which include the following problems on SLP-compressed strings: rank and select queries [1], subsequence matching [2], computing Karp-Rabin fingerprints [4], computing runs, squares, and palindromes [17], real-time traversal [14,23] and range-minimum queries [15]. In all these applications we either improve existing results or significantly simplify existing proofs by replacing $\mathrm{depth}(\mathcal{G})$ by $\mathcal{O}(\log n)$ in time/space bounds.

3 Proof Strategy

The proof of Theorem 1 consists of two main steps. Take an SLP $\mathcal{G}$ for the string s of length n and let m be the size of $\mathcal{G}$. We consider the derivation tree t for $\mathcal{G}$; it has size $\mathcal{O}(n)$. The SLP $\mathcal{G}$ can be viewed as a DAG for t of size m. We decompose this DAG into node-disjoint paths such that each path from the root to a leaf intersects $\mathcal{O}(\log n)$ paths from the decomposition. Each path from the decomposition is then viewed as a string of integer-weighted symbols, where the weights are the lengths of the strings derived from nodes that branch off from the path. For this weighted string we construct an SLP of linear size that produces all suffixes of the path in a certain weight-balanced way. Plugging these SLPs together yields the final balanced SLP.

Some of the ideas in our algorithm can be traced back to the area of parallel algorithms: the path decomposition for DAGs is related to the centroid path decomposition of trees [9], where it is the key technique in several parallel algorithms on trees. Moreover, the SLP of linear size that produces all suffixes of a weighted string can be seen as a weight-balanced version of the optimal prefix sum algorithm.

Our balancing procedure involves simple arithmetics on string positions, i.e., numbers of order n. Therefore we need machine words of bit-length $\Omega(\log n)$ in order to achieve a linear running time in Theorem 1; otherwise the running time increases by a multiplicative factor of order $\log n$. Note that such an assumption is realistic and standard in the field: machine words of bit length $\Omega(\log n)$ are needed, say, for indexing positions in the represented string. On the other hand, our procedure works in the pointer model regime.

4 Balancing Forest Straight-Line Programs and Top DAGs

Grammar-based compression has been generalized from strings to ordered node-labelled trees. In fact, the representation of a tree t by its smallest directed acyclic graph (DAG) is a form of grammar-based tree compression. This DAG is obtained by merging nodes where the same subtree of t is rooted. It can be seen as a regular tree grammar that produces only t. A drawback of DAG-compression is that the size of the DAG is lower-bounded by the height of the tree t. Hence, for deep narrow trees (like for instance caterpillar trees), the DAG-representation

cannot achieve good compression. This can be overcome by representing a tree t by a linear context-free tree grammar that produces only t. Such grammars are also known as *tree straight-line programs* in the case of ranked trees (where the number of children of a node is uniquely determined by the node label) [7,21,22] and *forest straight-line programs* in the case of unranked trees [13]. The latter are tightly related to *top DAGs* [3,5,10,13,16], which are another tree compression formalism, also akin to grammars. Our balancing technique works similarly for these compression formalisms:

Theorem 2. *Given a top DAG/forest straight-line program/tree straight-line program $\mathcal{G}$ producing the tree t one can compute in time $\mathcal{O}(|\mathcal{G}|)$ a top DAG/forest straight-line program/tree straight-line program $\mathcal{H}$ for t of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\log |t|)$.*

For top DAGs, this solves an open problem from [5], where it was shown that from an unranked tree t of size n, whose minimal DAG has size m (measured in number of edges in the DAG), one can construct in linear time a top DAG for t of size $\mathcal{O}(m \cdot \log n)$ and depth $\mathcal{O}(\log n)$. It remained open whether the additional factor $\log n$ in the size bound can be avoided. A negative answer for the specific top DAG constructed in [5] was given in [3]. On the other hand, Theorem 2 yields another top DAG of size $\mathcal{O}(m)$ and depth $\mathcal{O}(\log n)$. To see this note that one can easily convert the minimal DAG of t into a top DAG of roughly the same size, which can then be balanced. This also gives an alternative proof of a result from [10], according to which one can construct in linear time a top DAG of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ for a given tree of size n containing σ many different node labels.

Let us finally mention that Theorems 1 and 2 are instances of a general balancing result that applies to a large class of circuits over algebraic structures, see [12] for details.

References

1. Belazzougui, D., Cording, P.H., Puglisi, S.J., Tabei, Y.: Access, rank, and select in grammar-compressed strings. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 142–154. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_13
2. Bille, P., Cording, P.H., Gørtz, I.L.: Compressed subsequence matching and packed tree coloring. Algorithmica **77**(2), 336–348 (2017)
3. Bille, P., Fernstrøm, F., Gørtz, I.L.: Tight bounds for top tree compression. In: Fici, G., Sciortino, M., Venturini, R. (eds.) SPIRE 2017. LNCS, vol. 10508, pp. 97–102. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67428-5_9
4. Bille, P., Gørtz, I.L., Cording, P.H., Sach, B., Vildhøj, H.W., Vind, S.: Fingerprints in compressed strings. J. Comput. Syst. Sci. **86**, 171–180 (2017)
5. Bille, P., Gørtz, I.L., Landau, G.M., Weimann, O.: Tree compression with top trees. Inf. Comput. **243**, 166–177 (2015)
6. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM J. Comput. **44**(3), 513–539 (2015)

7. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML document trees. Inf. Syst. **33**(4–5), 456–474 (2008)
8. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005)
9. Cole, R., Vishkin, U.: The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. Algorithmica **3**, 329–346 (1988)
10. Dudek, B., Gawrychowski, P.: Slowing down top trees for better worst-case compression. In: Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018, volume 105 of LIPIcs, pp. 16:1–16:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
11. Ganardi, M., Jeż, A., Lohrey, M.: Balancing straight-line programs. In: Proceedings of the 60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, pp. 1169–1183. IEEE Computer Society (2019)
12. Ganardi, M., Jeż, A., Lohrey, M.: Balancing straight-line programs. CoRR, abs/1902.03568 (2019)
13. Gascón, A., Lohrey, M., Maneth, S., Reh, C.P., Sieber, K.: Grammar-based compression of unranked trees. In: Fomin, F.V., Podolskii, V.V. (eds.) CSR 2018. LNCS, vol. 10846, pp. 118–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90530-3_11
14. Gasieniec, L., Kolpakov, R.M., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proceedings of the 2005 Data Compression Conference, DCC 2005, p. 458. IEEE Computer Society (2005)
15. Gawrychowski, P., Jo, S., Mozes, S., Weimann, O.: Compressed range minimum queries. CoRR, abs/1902.04427 (2019)
16. Hübschle-Schneider, L., Raman, R.: Tree compression with top trees revisited. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 15–27. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20086-6_2
17. Tomohiro, I., et al.: Detecting regularities on grammar-compressed strings. Inf. Comput. **240**, 74–89 (2015)
18. Jeż, A.: Approximation of grammar-based compression via recompression. Theor. Comput. Sci. **592**, 115–134 (2015)
19. Jeż, A.: A really simple approximation of smallest grammar. Theor. Comput. Sci. **616**, 141–150 (2016)
20. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups Complex. Cryptol. **4**(2), 241–299 (2012)
21. Lohrey, M.: Grammar-based tree compression. In: Potapov, I. (ed.) DLT 2015. LNCS, vol. 9168, pp. 46–57. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21500-6_3
22. Lohrey, M., Maneth, S., Mennicke, R.: XML tree structure compression using RePair. Inf. Syst. **38**(8), 1150–1167 (2013)
23. Lohrey, M., Maneth, S., Reh, C.P.: Constant-time tree traversal and subtree equality check for grammar-compressed trees. Algorithmica **80**(7), 2082–2105 (2018)
24. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. **302**(1–3), 211–222 (2003)