Notes on Compression Schemes

Patrick Hagge Cording

1 Compression Schemes

These notes will introduce the compression schemes known as the Straight Line Program (SLP), LZ77 [24], and LZ78 [25]. The latter are named after its inventors Lempel and Ziv in the years 1977 and 1978. These three compression schemes are closely related. For instance, an LZ77 compressed file can be converted to an SLP with logarithmic overhead and an LZ78 file can be converted with only constant overhead. The conversion can be done in time nearly linear in the compressed size of the files. Consequently, if we design a data structure for an SLP of size n we also have a data structure for LZ77 of size $O(n \log \frac{N}{n})$ and a data structure for LZ78 of size $O(n)$. An overview of algorithms to convert between files from different compression schemes is given in [11].

1.1 Straight Line Programs

The SLP is a widely studied compression scheme because of its simplicity and it is known to model many other compression schemes. An SLP is a context-free grammar in Chomsky normal form that generates one string only. A production rule either has two other rules on its righthand side if it is non-terminal or a single character if it is terminal. Rules are unambiguous and non-recursive.

For convenience we assume that the SLP is represented as a directed acyclic graph (DAG). Nodes in the DAG are labeled with the names of production rules. Non-terminals have out-degree 2 and a terminal has one outgoing edge to a node labelled by one character. An example of an SLP is shown in Figure 1.

$$
\begin{aligned}
X_1 &= \mathtt{a}\\
X_2 &= \mathtt{b}\\
X_3 &= X_1X_2\\
X_4 &= X_3X_1\\
X_5 &= X_4X_3\\
X_6 &= X_5X_4\\
X_7 &= X_6X_5
\end{aligned}
$$

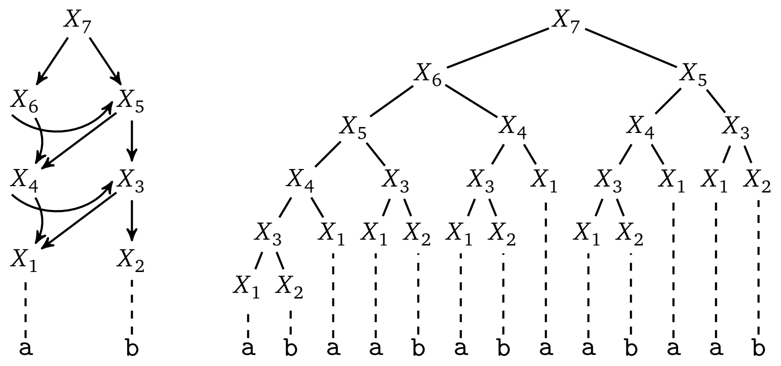

Figure 1: An SLP compressing the string abaababaabaab, its corresponding DAG, and its parse tree.

Let $\mathscr{S}$ denote an SLP with n nodes compressing a string S of size N. The size of the SLP will lie in the range $\log N \leq n \leq N$. To decompress a string we start a top-down left-to-right traversal starting from the node representing the start rule of the grammar. The tree emerging from this is the parse tree of the SLP and the leaves on the tree (read from left to right) is the uncompressed string. The height h of $\mathscr{S}$ is the longest path from its root to a leaf in the parse tree of $\mathscr{S}$.

If we want to support decompression of substrings without having to decompress the entire string, then we need to add auxiliary data to the SLP. The simplest approach requires $O(n)$ space and supports decompression of an arbitrary substring of length l in $O(h + l)$ time. Using a much more elaborate data structure, this can be improved to $O(\log N + l)$ time while retaining $O(n)$ space [4]. Using non-linear space, decompression can be done in $O(\log N/ \log\log N + l)$ time [3]. In some cases, the "kick-off" time can be avoided. We may decompress the prefix or suffix of length l of a string generated by some node in $O(l)$ time, provided that we have already found said node [10].

Finding the smallest SLP representing a string is NP-hard [5]. However, several approximation algorithms exist [20, 5, 13, 14] and efficient practical algorithms have been developed [19, 16].

1.2 LZ77

The LZ77 factorization of a string S of size N is a sequence of factors $f_1, \ldots, f_z$ such that these generate S when expanding them left to right. A factor has the form $f_k = (i, j, \alpha)$, where the string $S[i, j]$ is a substring of the string obtained from expanding $f_1, \ldots, f_{k-1}$ and α is a character. Some factors may expand to just α and will then have null values instead of i and j. The LZ77 factorization defined here is not self-referential. In the self-referential version of LZ77, a factor f_k is allowed to refer to a substring that is a concatenation of a suffix of $f_1, \ldots, f_{k-1}$ and a prefix of itself. Since we at most double the size of the string for every factor we add, the achievable compression for LZ77 is $\log N \leq z \leq N$. The following is an example of an LZ77 factorization of a string.

$$(-,-,\mathtt{a})(-,-,\mathtt{b})(1,1,\mathtt{a})(2,3,\mathtt{b})(3,6,\mathtt{a})(-,-,\mathtt{b})$$

Figure 2: The LZ77 factorization of the string `abaababaabaab`.

To decompress a string we process the factors from f_1 to f_z. When processing $f_k = (i, j, \alpha)$ we append $S[i, j]$ to the string that we have already decompressed from $f_1, \ldots, f_{k-1}$ and then append α.

The greedy LZ77 parse maximizes the substring of $f_1, \ldots, f_{k-1}$ referred to from f_k, and is known to produce the optimal LZ77 factorization [7, 6, 21] when measuring the number of factors produced[1]. The optimal factorization may be found in $O(N)$ time given the suffix tree/array[2] of S.

Given the LZ77 factorization of S we may build an SLP of size $O(z \log \frac{N}{z})$ for S [20]. Hence, a data structure using $f(n)$ space for SLPs is also a data structure using $O(f(z \log \frac{N}{z}))$ space for an LZ77 compressed file of size z. If we want to convert an SLP to the LZ77 factorization, this can be done in $O(\text{poly}(n))$ time [11].

Unlike the SLP, LZ77 is used widely in data compression software in practice (often combined with other methods and/or in derivative variants). For example, it is the basis of the gzip, png, rar, and zip file formats.

1.3 LZ78

Much like LZ77, the LZ78 factorization is a sequence of factors $f_1, \ldots, f_z$ that generate S when expanded left to right. However, in this scheme a factor has the form $f_k = (f_i, \alpha)$, where $1 \leq i < k$ and α is a character. As with LZ77, factors can have a null value instead of f_i and in this case only expand to one character. With LZ78 compression we may achieve a compression in the range $\sqrt{N} \leq z \leq N$. The following shows the LZ78 factorization of a string.

$$(-,\mathtt{a})(-,\mathtt{b})(1,\mathtt{a})(2,\mathtt{a})(4,\mathtt{a})(5,\mathtt{b})$$

Figure 3: The LZ78 factorization of the string `abaababaabaab`.

LZ78 can be seen as a restricted version of LZ77 where the strings that are referred to from factors are restricted to start and end in substrings that correspond to the expansion of a single preceding factor. Contrary to LZ77, the LZ78 parse may be seen as a grammar, and it converts to an SLP with less overhead. For each unique character we create a terminal node with an edge to that character. For each factor $f_k = (f_i, \alpha)$ we create a node v_k with v_i as its left child and the terminal node representing α as its right child. Finally, we build a binary tree with the sequence $v_1, \ldots, v_z$ as leaves. The resulting SLP has size $O(z)$. Recent advances also show that we can convert a string compressed by an SLP into its LZ78 factorization in almost linear time [2, 1].

LZ78 is used in e.g. the gif file format among others.

[1]The greedy parsing strategy is no longer optimal when measuring the output in number of bits required [9].
[2]It is assumed for this chapter that the reader is familiar with the suffix tree or array. Otherwise, see [18, 22, 23, 8, 12] or [17, 15].

References

[1] H. Bannai, P. Gawrychowski, S. Inenaga, and M. Takeda. Converting SLP to LZ78 in almost linear time. In *Proc. 24th CPM*, pages 38–49, 2013.

[2] H. Bannai, S. Inenaga, and M. Takeda. Efficient LZ78 factorization of grammar compressed text. In *Proc. 19th SPIRE*, pages 86–98, 2012.

[3] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Rank, select and access in grammar-compressed strings. *arXiv preprint arXiv:1408.3093*, 2014.

[4] P. Bille, G. Landau, R. Raman, K. Sadakane, S. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proc. 22nd SODA*, pages 373–389, 2011.

[5] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

[6] M. Cohn and R. Khazan. Parsing with suffix and prefix dictionaries. In *Proc. 6th DCC*, pages 180–180, 1996.

[7] M. Crochemore, A. Langiua, and F. Mignosi. Note on the greedy parsing optimality for dictionary-based text compression. *Theoret. Comp. Sci.*, 525(0):55 – 59, 2014.

[8] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143, 1997.

[9] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel–Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.

[10] L. Gąsieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.

[11] K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda. Restructuring compressed texts without explicit decompression. *arXiv preprint arXiv:1107.2729*, 2011.

[12] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[13] A. Jeż. Approximation of grammar-based compression via recompression. In *Proc. 24th CPM*, pages 165–176, 2013.

[14] A. Jeż. A really simple approximation of smallest grammar. In *Proc. 25th CPM*, 2014.

[15] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[16] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[17] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[18] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[19] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *J. Artificial Intelligence Res.*, 7:67–82, 1997.

[20] W. Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302(1):211–222, 2003.

[21] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[22] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[23] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th FOCS (SWAT)*, pages 1–11, 1973.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.