Bookmarks in Grammar-Compressed Strings

Patrick Hagge Cording[1⋆], Pawel Gawrychowski[2], and Oren Weimann[2]

[1] Technical University of Denmark, DTU Compute
[2] University of Haifa

Abstract. We consider the problem of storing a grammar of size n compressing a string of size N, and a set of positions $\{i_1, \ldots, i_b\}$ (*bookmarks*) such that any substring of length l crossing one of the positions can be decompressed in $O(l)$ time. Our solution uses space $O((n + b) \max\{1, \log^* n - \log^*(\frac{n}{b} + \frac{b}{n})\})$. Existing solutions for the bookmarking problem either require more space or a super-constant "kick-off" time to start the decompression.

1 Introduction

Textual databases for e.g. biological or web-data are growing rapidly, and it is often only feasible to store the data in compressed form. However, compressing the data comes at a price: it may be necessary to decompress the entire file in order to retrieve just a small portion of it. Inserting *bookmarks* in the compressed file can accommodate this problem. A bookmark in a compressed string is a position i from which any substring of length l crossing position i can be decompressed in $O(l)$ time.

A popular technique for compressing a string is to instead store a small grammar that generates the string (and only the string). The idea dates back far and has received much attention in the theory community while also being widely used in practice. In particular, popular compression schemes such as LZ78 [15], LZW [13], Re-pair [9], and Sequitur [11] produce grammars. Even the LZ77 [14] compression scheme that does not produce a grammar, can be converted to a grammar with only a logarithmic overhead in the space [5,12]. For our purposes, we consider Straight Line Programs (SLPs). These are context-free grammar in Chomsky Normal Form that generate exactly one string. SLPs capture any grammar-based compression scheme.

For the *bookmarking problem*, we are given an SLP $\mathcal{S}$ of size n compressing a string S of size N and a set of positions $\{i_1, \ldots, i_b\}$, and we want to construct a data structure that supports linear-time decompression of substrings crossing any of the b positions.

⋆ Supported by the Danish Research Council under the Sapere Aude Program (DFF 4005-00267).

Related work. Gagie et al. [6] presented a bookmarking data structure that uses $O(n + b \log^* N)$ space[3] for *balanced* SLPs (i.e., SLPs whose parse tree is balanced). When the SLP is unbalanced, we may use an algorithm to balance it at the cost of adding nodes [5, 12], and as a result the space usage of their data structure increases to $O(n \log \frac{N}{n} + b \log^* N)$.

A more general problem is to support random access to the compressed string (i.e., access to a single character of S without decompression). This does not require any bookmarks to be predefined, but in turn incurs a "kick-off" time when decompressing a substring. If we allow the kick-off time to be $O(\log N)$ (i.e., $O(l + \log N)$ time to decompress a substring of length l), we may use the $O(n)$-space data structure of Bille et al. [4]. For a faster kick-off time of $O(\log_\tau N)$, for any $2 < \tau \leq \log N$, we may instead apply the data structure of Belazzougui et al. [1] at the cost of increasing the space to $O(n\tau \log_\tau \frac{N}{n})$. The data structure of Belazzougui et al. [2] supports random access to any character of the compressed string in $O(1)$ time and thereby allows decompression of any substring in time linear in the substring's length. However, this data structure uses space $O(n^{1-\epsilon} N^\epsilon)$ for some constant $0 < \epsilon \leq 1$.

The compressed finger search problem is somehow a hybrid of the bookmarking problem and the random access problem. For this problem, we place a set of *fingers*, and now we may answer random access queries in $O(\log D)$ time, where D is the distance from a given finger to the index we query for [3]. Using this data structure, we get a bookmarking data structure of $O(n)$ space that can decompress any substring of length l in $O(l \log l)$ time.

Our results. In this paper we present a bookmarking data structure for SLP-compressed strings that uses space $O((n+b) \max\{1, \log^* n - \log^*(\frac{n}{b} + \frac{b}{n})\})$ and supports decompression of length-l substrings crossing bookmarks in $O(l)$ time. The space is measured in words and we assume the standard RAM model of computation.

The general idea is to make τ copies of the SLP for some parameter τ. Each copy is modified so that the decompression kick-off takes less time but only supports decompression of substrings up to a certain length and from certain positions. At query time, we then select the copy of the SLP that provides a kick-off time of $O(l)$.

2 Preliminaries

Let S be a string of length $|S|$ consisting of characters from an alphabet of size σ. We use $S[i, j]$, $1 \leq i \leq j \leq |S|$, to denote the substring starting in position i and ending in position j of S.

A Straight Line Program (SLP) $\mathcal{S}$ is a context-free grammar in Chomsky normal form with n production rules that derives a single string S of size N. We

[3] The bound is in fact $O(z + b \log^* N)$, where z is the size of the LZ77 parse of S. Since it is known that $z \leq n' \leq n$ [12], where n' is the size of the smallest SLP generating S, we replace z by n for clarity.

represent the SLP as a rooted, ordered, and node-labelled directed acyclic graph (DAG) with outdegree 2 and we will refer to production rules as nodes where it is appropriate. We denote by $v = uw$ that node v in the DAG has left-child u and right-child w. A depth-first left-to-right traversal starting from a node v in the DAG produces the string $S(v)$. As a shorthand we sometimes use $|v|$ instead of $|S(v)|$.

All logarithms in this paper are base 2. As a shorthand to denote the logarithm applied i times to a number n we write $\log^{(i)} n$, e.g., $\log^{(3)} n = \log\log\log n$. The iterated logarithm $\log^* n$ is equal to the number of times the logarithm can be applied to n before the result is less than 1, i.e., $\log^* n = \arg\min_i\{\log^{(i)} n \leq 1\}$. We also need the up-arrow notation of Knuth [8] defined as follows: $2 \uparrow\uparrow 0 = 1$ and $2 \uparrow\uparrow (k+1) = 2^{2\uparrow\uparrow k}$. Observe that $k = \log^* n$ if and only if $2 \uparrow\uparrow (k-1) < n \leq 2 \uparrow\uparrow k$.

3 A Simple Solution

In this section we give a simple data structure to the bookmarking problem with the following bounds.

Theorem 1. *Given an SLP for $S[1, N]$ with n rules and positions $i_1, \ldots, i_b$ in S, we can store S in $O(n + b + \min\{n, b\}\log N)$ space such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $S[i, i + l]$ in $O(l)$ time.*

Our solution builds on the following data structure by Bille et al. [4].

Lemma 1 ([4]). *Let S be a string of length N compressed by an SLP $\mathcal{S}$ of size n. There is data structure of size $O(n)$ that, given a node v in $\mathcal{S}$, supports decompression of a substring $S(v)[i, i + l]$ in $O(\log |v| + l)$ time.*

Notice that when $l \geq \log N$ the decompression time in Lemma 1 is dominated by the $O(l)$ term. This means we only need to focus on the case where $l < \log N$.

To obtain a $O(n + b\log N)$-space solution, since $l < \log N$, we can simply store the substring $S[i-\log N, i+\log N]$ for each bookmark $i \in \{i_1, \ldots, i_b\}$ along with the data structure of Lemma 1.

In the case where $n < b$, to obtain a $O(n \log N + b)$-space solution, we show that it is sufficient to store n substrings each of length $O(\log N)$. For this we use the following lemma, stating that any substring of S is the concatenation of a suffix of $S(u)$ and a prefix of $S(w)$ for some node v whose left child is u and right child is w. The observation was first used for compressed pattern matching [10]. For the sake of completeness, we will give a proof using our terminology.

Lemma 2 ([10]). *Let S be a string of length N compressed by an SLP $\mathcal{S}$ of size n. Let $r(v) = S(u)[\max\{1, |u| - k\}, |u|]S(w)[1, \min\{1, k - 1\}]$ be the relevant substring with respect to k of a node $v = uw$ in $\mathcal{S}$. Then any substring of S of length at most k is also a substring of some string in $\{r(v) \mid v \in \mathcal{S} \wedge |v| \geq k\}$.*

Proof. The proof is by induction. For the base case, consider a node $v = uw$ where $|v| \leq 2k-2$ and $|u| < k$ and $|w| < k$. Since $r(v) = S(v)$ this obviously contains every substring of length k. For the inductive step we again consider some node $v = uw$ and we know that $S(v) = S(u) \circ S(w)$. Assume that $|u| \geq k$ and $|w| \geq k$, then by the induction hypothesis it holds that the set of strings $\{r(u') \mid u' \in \mathcal{S}(u) \wedge |u'| \geq k\} \cup \{r(w') \mid w' \in \mathcal{S}(w) \wedge |w'| \geq k\}$ contains all substrings of length k in $S(u)$ and $S(w)$. The substrings of length k starting in $S(u)$ and $S(w)$ are not guaranteed to be in this set, but since $r(v)$ contains exactly all these, they will be after adding $r(v)$ to the set. For the cases when $|u| < k$ or $|w| < k$ the same argument holds. □

For our data structure, we set $k = 2 \log N$ and store the strings $r(v)$ for all $v \in \mathcal{S}$. For each bookmark i we store the deepest node that generates the string $S[i - \log N, i + \log N]$. Furthermore, we build the data structure of Lemma 1 for use for the case where $l \geq \log N$.

Since $|r(v)| \leq 4 \log N - 2$ and we store $O(1)$ words (pointers) for each bookmark, and the data structure of Lemma 1 uses $O(n)$ space, our data structure uses $O(n \log N + b)$ space in total. This concludes the proof of Theorem 1.

4 A Leveled Solution

We now describe a data structure that seeks to reduce the $\log N$ factor of the space usage in Theorem 1. The time to decompress a substring of length l crossing some bookmark is still $O(l)$. The key to our solution is a technique due to Gawrychowski [7] captured by the following lemma.

Lemma 3 ([7]). *Let S be a string of length N compressed into an SLP $\mathcal{S}$ of size n. We can choose an arbitrary ℓ and modify $\mathcal{S}$ in $O(N)$ time by adding $O(n)$ new variables such that we can write S as $S = S(v_1)S(v_2)\dots S(v_m)$ with $m = O(N/\ell)$ and $|S(v_i)| \leq 2\ell - 2$. Furthermore, for any substring $S[i, i+\ell]$ there are a constant number of nodes $v_1, \dots, v_c$ such that $S[i, i+\ell]$ is a substring of $S(v_1)\dots S(v_c)$.*

The lemma says that we can restructure the given SLP such that for any substring $S[i, i+\ell]$ we can find $O(1)$ nodes whose concatenation has total length $O(\ell)$ and contains $S[i, i+\ell]$ as a substring. We now describe how to apply this restructuring procedure to get a bookmarking data structure using almost linear space. In the description we use the parameter τ which is later to be minimized subject to n and b.

Construction. First we make τ copies of $\mathcal{S}$, denoted by $\mathcal{S}_1, \dots, \mathcal{S}_\tau$. We then apply the restructuring procedure for $\ell = \log N, \log^{(2)} N, \dots, \log^{(\tau)} N$ to the τ copies of $\mathcal{S}$ to get $\mathcal{S}'_1, \dots, \mathcal{S}'_\tau$. Next, we build the data structure of Lemma 1 for each SLP $\mathcal{S}'_1, \dots, \mathcal{S}'_\tau$. For each $\mathcal{S}'_j$, let a *block node* be a node v for which $|S(v)| = \Theta(\log^{(j)} N)$. For each SLP $\mathcal{S}'_j$ and for each bookmark i we store the $O(1)$ block nodes generating the string containing $S[i - \log^{(j)} N, i + \log^{(j)} N]$. We also store the relative index of position i in the string generated by the first

block node. On the lowest level (i.e., for $\mathcal{S}'_\tau$) we apply the technique from the previous section. I.e., if $b \leq n$ we use $O(n + b \log^{(\tau)} N)$ space and if $b > n$ we use $O(n \log^{(\tau)} N + b)$ space.

Decompression. To decompress a substring of length l from a bookmark position $i \in \{i_1, \ldots, i_b\}$ we do the following.

If $\log^{(j+1)} N < l \leq \log^{(j)} N$ for some $j < \tau$. We locate the block node that contains i in $\mathcal{S}'_j$ and decompress the string starting in the relative position stored for the current bookmark using the data structure of Lemma 1. If we reach the end of the string generated by the current block node, we move on to the next node that we stored and repeat the process from relative position 1. When we decompress from a block node v in $\mathcal{S}'_j$, the query time of Lemma 1 becomes $O(\log \log^{(j)} N + l) = O(l)$ since $\log^{(j+1)} N < l$. We visit $O(1)$ block nodes so the total time to decompress $S[i, i + l]$ becomes $O(l)$.

If on the other hand $l < \log^{(\tau)} N$, then we use the solution chosen for the bottom level, which according to Theorem 1 yields a decompression time of $O(l)$.

Analysis. Our data structure creates τ copies of $\mathcal{S}$. Each has size $O(n)$ after the restructuring of Lemma 3 and the application of Lemma 1, i.e, this requires $O(\tau n)$ space. For each bookmark, we store references to $O(1)$ nodes in each copy for a total of $O(\tau b)$ space. For $\mathcal{S}'_\tau$ we need $O(\min\{n, b\} \log^{(\tau)} N)$ space as stated in Theorem 1. Hence, the total space usage is $O(\tau(n + b) + \min\{n, b\} \log^{(\tau)} N)$, which is equal to $O(\tau(n + b) + \min\{n, b\} \log^{(\tau)} n)$, because $n \leq N \leq 2^n$ in any SLP. It remains to choose τ as to minimize this expression.

We define $x = \frac{n+b}{\min\{n,b\}} \geq 1$. Then, the goal is to minimize $\min\{n, b\} f(\tau)$, where $f(\tau) = x \cdot \tau + \log^{(\tau)} n$, over all $\tau \geq 1$. We claim that $f(\tau)$ is minimized (up to a constant multiplicative factor) for $\tau = \max\{1, \log^* n - \log^* x + 1\}$, when $f(\tau) = O(x \max\{1, \log^* n - \log^* x\})$. If $\log^* n - \log^* x + 1 < 2$ then $x > \log n$, so the expression is minimized for $\tau = 1$. Otherwise, define $p = \log^* n$ and $q = \log^* x$, where $t = p - q + 1 \geq 2$. By the properties of iterated log, $2 \uparrow\uparrow (p-1) < n \leq 2 \uparrow\uparrow p$ and $2 \uparrow\uparrow \leq (q - 1) < x \leq 2 \uparrow\uparrow q$. Hence $\log^{(p-q+1)} n \leq 2 \uparrow\uparrow (q - 1) < x$ and $f(t) \leq x(t + 1) \leq 2x \cdot t$. We claim that, for any $\tau \geq 1$, $f(\tau) \geq \frac{1}{4} x \cdot t$, that is, $\tau = t$ is the (asymptotically) best choice.

If $\tau \geq \frac{1}{4}t$ then clearly $f(\tau) \geq x \cdot \tau \geq \frac{1}{4} x \cdot \tau$. It remains to analyze the case $\tau < \frac{1}{4}t$. We will prove that, for any $\tau < \frac{1}{4}t$, $\log^{(\tau)} n \geq \frac{1}{4} x \cdot t$. Because $\log^{(\tau)} n$ is monotone in τ, it is enough to prove that $\log^{(\frac{1}{4}t-1)} n \geq \frac{1}{4} x \cdot t$, or by the properties of iterated log $2 \uparrow\uparrow (p - \frac{1}{4}t) \geq 2 \uparrow\uparrow q \cdot \frac{1}{4}t$. Because $p - \frac{1}{4}t \geq q + \frac{1}{4}t$ by the assumption that $p - q \geq 1$, this reduces to showing that $2 \uparrow\uparrow (q + \frac{1}{4}t) \geq 2 \uparrow\uparrow q \cdot \frac{1}{4}t$.

Lemma 4. *For any $x, y \geq 0$, $2 \uparrow\uparrow (x + y) \geq 2 \uparrow\uparrow x \cdot y$.*

Proof. If $x = 0$, we need to show that $2 \uparrow\uparrow y \geq y$, which holds for any $y \geq 0$. From now on, we assume that $x \geq 1$ and apply induction on $y \geq 0$.

For $y = 0$, the left side is positive and the right side is zero. For $y = 1$, $2^{2 \uparrow\uparrow x} \geq 2 \uparrow\uparrow x$ holds for all $x \geq 0$. For $y = 2$, $2 \uparrow\uparrow (x + 2) > 2^{2 \uparrow\uparrow x} \geq 2 \uparrow\uparrow x \cdot 2$ holds for all $x \geq 0$.

Now assume that $2 \uparrow\uparrow (x+y) \geq 2 \uparrow\uparrow x \cdot y$ for some $y \geq 2$. Then

$$2 \uparrow\uparrow (x+y+1) = 2^{2\uparrow\uparrow(x+y)} \geq 2^{2\uparrow\uparrow x \cdot y} \geq (2 \uparrow\uparrow x)^y.$$

So now it is enough to show that $(2 \uparrow\uparrow x)^{y-1} \geq y+1$. But $x \geq 1$, so this reduces to $2^{y-1} \geq y+1$, which holds for any $y \geq 3$. □

In conclusion, choosing $\tau = \max\{1, \log^* n - \log^* x + 1\}$ gives us the total space usage of $O((n+b)\max\{1, \log^* n - \log^* x\})$. By rewriting this expression to remove x we get the following Theorem.

Theorem 2. *Given an SLP for $S[1,N]$ with n rules and positions $i_1, \ldots, i_b$ in S, we can store S in space $O((n+b)\max\{1, \log^* n - \log^*(\frac{n}{b} + \frac{b}{n})\})$ such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $S[i, i+l]$ in $O(l)$ time.*

5 Conclusion

We have shown a bookmarking data structure that uses a little more than linear space. If $b \leq \frac{n}{\log^{(c)} N}$ or $n \log^{(c)} N \leq b$ the space becomes $O(n+b)$. Furthermore, $O(n+b)$ space can be achieved for any n and b if we are willing to pay a $O(\log^{(c)} N)$ kick-off time for decompression. It remains open whether there exists a bookmarking data structure that uses $O(n+b)$ space and supports linear time decompression, regardless of the relationship between n and b.

References

1. Belazzougui, D., Cording, P.H., Puglisi, S.J., Tabei, Y.: Access, rank, and select in grammar-compressed strings. In: ESA, pp. 142–154 (2015)
2. Belazzougui, D., Gagie, T., Gawrychowski, P., Kärkkäinen, J., Ordónez, A., Puglisi, S.J., Tabei, Y.: Queries on LZ-bounded encodings. In: DCC. pp. 83–92 (2014)
3. Bille, P., Christiansen, A.R., Cording, P.H., Gørtz, I.L.: Finger search in grammar-compressed strings. CoRR abs/1507.02853 (2015)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM Journal on Computing 44(3), 513–539 (2015)
5. 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)
6. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: Lz77-based self-indexing with faster pattern matching. In: LATIN, pp. 731–742 (2014)
7. Gawrychowski, P.: Faster algorithm for computing the edit distance between SLP-compressed strings. In: SPIRE. pp. 229–236 (2012)
8. Knuth, D.E.: Mathematics and computer science: coping with finiteness. Science 194(4271), 1235–1242 (1976)
9. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. DCC 88(11), 1722–1732 (2000)

10. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: CPM. pp. 1–11 (1997)
11. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: A linear-time algorithm. J. Artif. Intell. Res.(JAIR) 7, 67–82 (1997)
12. Rytter, W.: Application of Lempel–Ziv factorization to the approximation of grammar-based compression. Theoret. Comput. Sci. 302(1), 211–222 (2003)
13. Welch, T.A.: A technique for high-performance data compression. Computer 6(17), 8–19 (1984)
14. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. Information Theory, IEEE Trans. Inf. Theory 23(3), 337–343 (1977)
15. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. Information Theory, IEEE Trans. Inf. Theory 24(5), 530–536 (1978)