Block Graphs in Practice

Travis Gagie[1,*], Christopher Hoobin[2] Simon J. Puglisi[1]

[1]Department of Computer Science, University of Helsinki, Finland
[2]School of CSIT, RMIT University, Australia
[*]Travis.Gagie@cs.helsinki.fi

Abstract

Motivated by the rapidly increasing size of genomic databases, code repositories and versioned texts, several compression schemes have been proposed that work well on highly-repetitive strings and also support fast random access: e.g., LZ-End, RLZ, GDC, augmented SLPs, and block graphs. Block graphs have good worst-case bounds but it has been an open question whether they are practical. We describe an implementation of block graphs that, for several standard datasets, provides better compression and faster random access than competing schemes.

1 Introduction

Advances in DNA sequencing technology have led to massive genomic databases, the open-source movement has led to massive code repositories, and the popularity of wikis has led to massive versioned textual databases. Fortunately, all these datasets tend to be highly repetitive and, thus, highly compressible. Compressing them is only useful, however, if we can still access them quickly afterwards. Although many papers have been published about compression schemes with fast random access (see [6] for a recent survey), most have been about schemes such as Huffman coding, LZ78, CSAs or BWT-based coding. Only relatively recently have researchers started proposing schemes with random access and LZ77-like compression, which is better suited highly-repetitive strings.

One approach uses variants of LZ77 itself. Kreft and Navarro's LZ-End [7] is practical but lacks good worst-case bounds, for both the compression and the random-access time. Kuruppu, Puglisi and Zobel's Relative Lempel-Ziv (RLZ) [8, 9] or Deorowicz and Grabowski's Genome Differential Compressor (GDC) [4] are also practical but are not general-purpose: we can apply them only when we have a good reference sequence, or can construct one. Even then, Deorowicz, Danek and

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

Grabowski [3] have observed that when compressing genomes, "the key to high compression is to look for similarities across the whole collection, not just against one reference sequence".

Another approach uses straight-line programs (SLPs). An SLP for a string s is a context-free grammar in Chomsky normal for that generates s and only s. Rytter [13] and Charikar et al. [2] showed that if s has length n and its LZ77 parse consists of z phrases, then we can build a SLP for s with $\mathcal{O}(z \log n)$ rules and height $\mathcal{O}(\log n)$. If we store the resulting SLP together with the size of each non-terminal's expansion, which takes a total of $\mathcal{O}(z \log n)$ space, then we can extract any substring of length ℓ in $\mathcal{O}(\log n + \ell)$ time. This extraction time nearly matches a lower bound by Verbin and Yiu [14]. Bille et al. [1] showed how we can store any SLP with r rules, regardless of the height, in $\mathcal{O}(r)$ space with the same time bound for extraction. These data structures are not practical, however. The most recent and practical SLP-based scheme of which we are aware is Maruyama, Tabei, Sakamoto and Sadakane's Fully-Online LCA (FOLCA) [10] but, apart from needing less resources for construction, even this is less practical than LZ-End.

In a previous paper [5] we introduced a third approach, called block graphs, and showed that they also use $\mathcal{O}(z \log n)$ space and $\mathcal{O}(\log n + \ell)$ extraction time. We did not implement these data structures for that paper, however, so it has been an open question whether they are practical. In this paper we describe an implementation that, for several standard datasets, provides better compression and faster random access than LZ-End; thus, block graphs are competitive in both theory and practice. In Section 2 we review the definition of the block graph for a string. In Section 3 we describe some details of our implementation. Finally, in Section 4 we report on our experiments.

2 Block Graphs

The block graph of the string $s[1..n]$ is a directed acyclic graph (DAG) in which each node in-degree up to 2 and out-degree up to 3. For simplicity, assume n is a power of 2. Each node of the block graph corresponds to a substring, called that node's block: the root corresponds to the whole of s; if they exist, the children of a node v correspond to the first half v's substring, the middle half of v's substring (which overlaps the first and last halves), and the last half of v's substring. In general, v shares its left and right children with its left and right siblings, respectively, because v's left child's block is both the last half of v's left sibling's block and the first half of v's left sibling's block, and v's right child's block is both the last half of v's block and the first half of v's right sibling's block. If n is not a power of 2, then we pad it so that it is, build the block graph, and prune redundant nodes. Figure 1 — which is copied from our earlier paper — shows the block graph for the eighth Fibonacci string, abaababaabaababaababa, truncated at depth 3.

We mark as an *internal node* each node whose block is the first occurrence of that substring (shown in Figure 1 as ovals). We mark as a *leaf* all nodes whose block is not unique and whose parents are internal nodes (shown in Figure 1 as rectangles). We remove leaves' children (and their descendants) and replace them by pointers, as explained in our earlier paper:

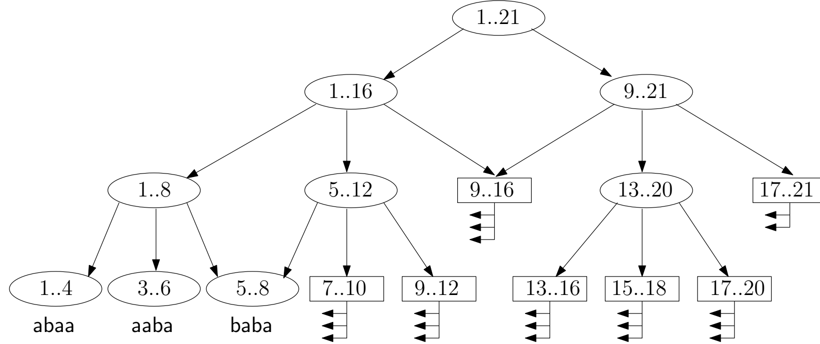

Figure 1: The block graph for the eighth Fibonacci string, abaababaabaababaababa, truncated at depth 3.

> Suppose a leaf u at depth d had a child $\langle i..j \rangle$... Consider the first occurrence $s[i'..j']$ in s of the substring $s[i..j]$. Notice that $s[i'..j']$ is completely contained within some block at depth d — this is one reason why we use overlapping blocks — and, since $s[i'..j']$ is the first occurrence of that substring in s, that block is associated with an internal node v. We replace the pointer from u to $\langle i..j \rangle$ by a pointer to v and the offset of i' in v's block.

At some depth, we store all internal nodes' blocks instead of their children; this depth is 3 in Figure 1.

In our previous paper we proved that

1. the block graph for s has $\mathcal{O}(z \log n)$ nodes and, thus, takes $\mathcal{O}(z \log n)$ space;
2. we can extract any substring of length ℓ in $\mathcal{O}(\log n + \ell)$ time;
3. if we store all of the nodes at depth d but removing all nodes above that depth, we change the space usage to $\mathcal{O}\big(z(\log n - d) + 2^d\big)$ and decrease the extraction time to $\mathcal{O}(\log n - d)$.

For more details, we refer readers to the proofs and discussion it contains.

3 Implementation

We now describe an implementation of block graphs which is efficient in practice. The main idea is to represent the shape of the graph (the internal nodes and their pointers) using bitvectors and operations from succinct data structures, and to carefully allocate space for the leaf nodes depending on their distance from the root. Below we make use of two familiar operations for bitvectors: *rank* and *select*. Given a bitvector B, a position i, and a type of bit b (either 0 or 1), $rank_b(B, i)$ returns the number of occurrences of b before position i in B and $select_b(B, i)$ returns the position of the ith b in B. Efficient data structures supporting these operations have been extensively studied (see, e.g., [11, 12]).

Recall, each level of the block graph consists of a number of nodes, either internal nodes, or leaves. Let B_d be a bitvector which indicates whether the ith node (from the left) at depth d is a leaf, $B_d[i] = 0$, or an internal node, $B_d[i] = 1$. We define another bitvector R_d, where $R_d[i] = 1$ if and only if $B_d[i] = 1$ and $B_d[i+1] = 1$ for $i < n-1$. That is, we mark a 1 bit for each instance of two adjacent internal nodes in B_d, otherwise $R_d[i] = 0$. Let L_d be an array that holds leaf nodes at depth d. The structure of a leaf node is discussed below. Finally, let T be the concatenation of the textual representation (ie. the corresponding substrings) of all internal nodes at the truncated depth, d'. As adjacent text blocks share $2^{\log n - d' - 1}$ characters, we concatenate only the last half of a new adjacent block to T. Non-adjacent blocks are fully concatenated. We utilize an R_d bitvector at this level; however, we mark $R_d[i] = 1$ if the ith node at the truncated depth is a text block.

Navigating the block graph

The main operation is to traverse from an internal node to one of its three children. Say we are currently at the jth internal node at depth d of the block graph — that is, we are at $B_d[i]$, where $i = select_1(B_d, j)$. Each internal node has three children. If these children were independent then locating the left child of the current node would be simply three times the node's position on its level, that is $3j = 3 \cdot rank_1(B_d, i)$. However, in a block graph, adjacent internal nodes share exactly one child, so we correct for this by subtracting the number of adjacent internal nodes at this depth prior to the current node — this is given by $rank_1(R_d, i)$. To find the position corresponding to the left child of a node in B_{d+1} we compute $\text{leftchild}(B_d, i) = 3 \cdot rank_1(B_d, i) - rank_1(R_d, i)$.

Given the address of the left child it is easy to find the center or right child by adding 1 or 2, respectively. If $B_d[i] = 0$ then we are at a leaf node, and its leaf information is at $L_d[rank_0(B_d, i)]$. Once we reach the truncated depth we access the text of an internal node by computing its offset in T as $T[(rank_1(B_d, i) \cdot 2^{\log n - d' - 1}) - (rank_1(R_d, i) \cdot 2^{\log n - d' - 1})]$.

Leaf nodes

In a block graph, leaves point to internal nodes. For each leaf we store two values, the position of the destination node on the current level, and an offset in the destination node pointing to the beginning of the leaf block. Note that we do not need to store the depth of the destination node. It is, by definition, on the level above the leaf, and we know this by keeping keep track of the depth during each step in a traversal. To improve compression we store leaf positions and offsets in two separate arrays. At depth d there are no more than $2^{d+1} - 1$ possible nodes, so we can store each position in $\log(2^{d+1} - 1)$ bits. Given that the length of a node at depth d is $b = 2^{\lceil \log n \rceil - d}$ and leaf nodes point to an internal node on the level above, we store each offset in $\log(2^{\lceil \log n \rceil - d - 1})$ bits.

Table 1: Size in MB of repetitive corpus files encoded with ASCII, gzip, xz, LZ-End and block graphs truncated at text length 4, 8, 16 and 32.

Collection	ASCII	GZIP	XZ	LZ-End	Bg4	Bg8	Bg16	Bg32
world_leaders	49	8.28	0.51	4.52	6.62	5.83	5.72	6.37
Escherichia_Coli	112	31.53	5.18	49.10	49.70	49.57	45.33	46.91
influenza	154	10.63	1.59	21.50	33.16	32.97	33.32	37.89
coreutils	205	49.92	3.70	35.88	42.80	33.19	30.43	33.00
kernel	257	69.39	2.07	19.34	21.21	15.69	13.84	14.05
para	429	116.07	6.09	57.41	72.39	72.13	67.84	70.66
cere	461	120.08	5.07	41.34	57.68	57.54	54.59	57.96
einstein.en.txt	467	163.66	0.33	2.24	3.52	3.07	3.01	3.19

4 Experiments

We have developed an implementation of block graphs[1] and tested it on texts from the Pizza-Chili Repetitive Corpus[2], a standard testbed for data structures designed for repetitive strings.

We compared compression achieved by the block graph to the LZ-End data structure by Kreft and Navarro [7], and to the general-purpose compressors gzip and xz; the results are shown in Table 1. gzip and xz were run with their highest compression setting -9, while LZ-End was executed with its default settings. Throughout our experiments we tested block graphs that were truncated such that the smallest blocks were 4, 8, 16 and 32 bytes. Note that gzip and xz provide compression only, not random access, and are included as reference points for achievable compression. We did not test extraction from bookmarks because Kreft and Navarro's data structure does not support it (nor does any other implemented data structure).

We then compared how quickly block graphs and LZ-End support extracting substrings of various lengths; the results are shown in Figure 2. The mean extraction speed with LZ-End never exceeded 9 million characters per second while, for sufficiently long extractions from the kernel file, the mean extraction speed with the block graph was over 480 million characters per second. Each run of extractions was performed across 10,000 randomly-generated queries. Experiments were conducted on an Intel Core i7-2600 3.4 GHz processor with 8GB of main memory, running Linux 3.3.4; code was compiled with GCC version 4.7.0 targeting x86_64 with full optimizations. Caches were dropped between runs with `sync && echo 1 > /proc/sys/vm/drop_caches`.

Although xz achieves much better compression, block graphs achieve better compression than gzip except on the Escherichia Coli and influenza files. Most importantly, our experiments show that block graphs generally achieve compression comparable to that achieved by LZ-End while supporting significantly faster substring extraction.

[1] Available at http://www.github.com/choobin/block-graph
[2] http://pizzachili.dcc.uchile.cl/repcorpus.html

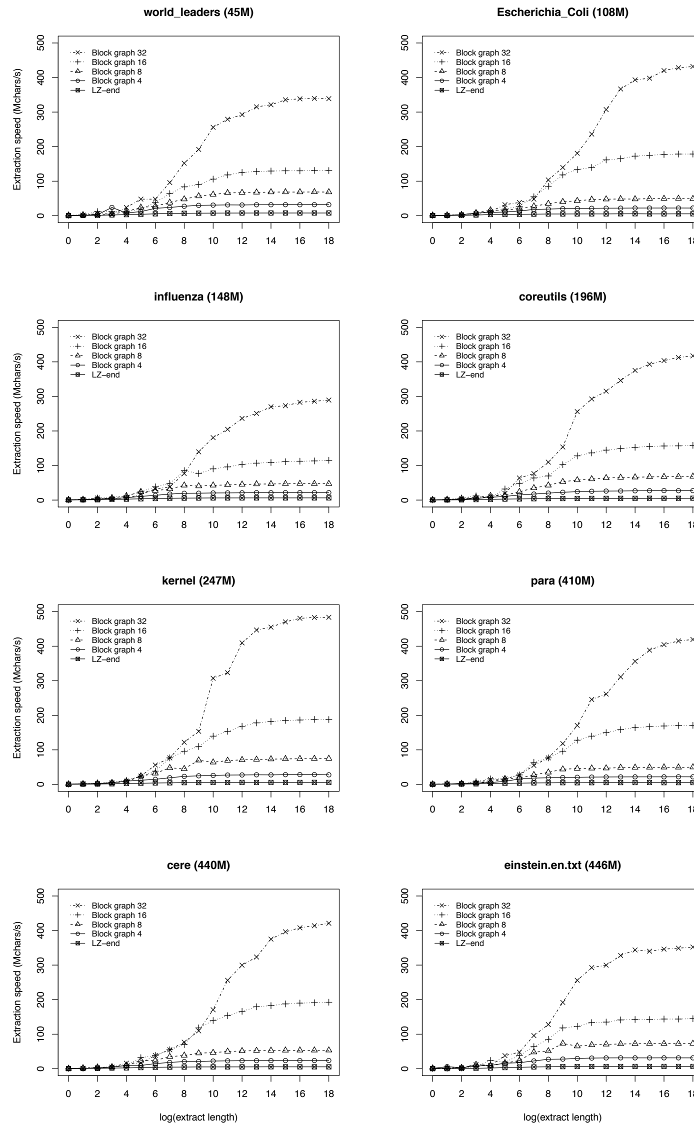

Figure 2: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.

References

[1] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA)*, pages 373–389, 2011.

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

[3] S. Deorowicz, A. Danek, and S. Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

[4] S. Deorowicz and S. Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011.

[5] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 653–662, 2011.

[6] R. Grossi. Random access to high-order entropy compressed text. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, pages 199–215. Springer-Verlag, 2013.

[7] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, to appear.

[8] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[9] S. Kuruppu, S. J. Puglisi, and J. Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proceedings of the 34th Australasian Computer Science Conference (ACSC)*, pages 91–98, 2011.

[10] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-online grammar compression. In *Proceedings of the 20th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–229, 2013.

[11] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[12] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[13] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[14] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proceeings of the 24th Symposium on Combinatorial Pattern Matching (CPM)*, pages 247–258, 2013.