Relative Select

Christina Boucher[1], Alexander Bowe[2], Travis Gagie[3,⋆],
Giovanni Manzini[4], and Jouni Sirén[5]

[1] Colorado State University, USA
[2] National Institute of Informatics, Japan
[3] Helsinki Institute for Information Technology,
University of Helsinki, Finland
[4] University of Eastern Piedmont, Italy
[5] Wellcome Trust Sanger Institute, UK

Abstract. Motivated by the problem of storing coloured de Bruijn graphs, we show how, if we can already support fast select queries on one string, then we can store a little extra information and support fairly fast select queries on a similar string.

1 Introduction

Many compressed data structures for strings rely on three fundamental queries: access, rank and select. The query $S.\mathsf{access}(i)$ on a string S returns its ith character; the query $S.\mathsf{rank}_a(i)$ returns the number of occurrences of character a in the prefix of S of length i; and the query $S.\mathsf{select}_a(j)$ returns the position of the jth leftmost occurrence of a in S. Suppose we have a data structure supporting these queries on a string S_1 and we want another data structure supporting them on a similar string S_2. It is not difficult to store S_2 in small space and support access to it via access to S_1. For example, we can find a longest common subsequence of S_1 and S_2, store two bitvectors with 1s marking their characters not in that subsequence, and store the characters marked in S_2. The total number of 1s in the two bitvectors is at most twice the standard edit distance d between S_1 and S_2 (i.e., the number of single-character insertions, deletions and substitutions needed to change one into the other) so we can store them in $\mathcal{O}(d)$ space and support rank and select on them using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time using a sparse-bitvector implementation [6]. To access $S_2[i]$, we check whether it appears in the common subsequence: if so, we use rank and select queries on the bitvectors to find the corresponding character in S_1, which we access; if not, we find $S_2[i]$'s rank among characters marked in S_2 and look it up.

Last year, when describing their relative FM-index data structure, Belazzougui et al. [1] showed how to store $\mathcal{O}(d)$ extra words and support any rank query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a rank query on S_1. In this paper we show how to store $\mathcal{O}(d)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1. We call

⋆ Supported by grants 268324 and 250345 (CoECGR) from the Academy of Finland.

this *relative select* and we expect it to be useful when storing compressed data structures for navigating in coloured de Bruijn graphs [8].

Belazzougui el al. were interested in saving space when storing FM-indexes [5] for many genomes from the same species. An FM-index for a genome is essentially just a data structure supporting access and rank on the Burrows-Wheeler Transform [4] (BWT) of that genome. The BWT sorts the characters of a string into the lexicographic order of the suffixes that immediately follow them. The edit distance between two genomes from the same species tends to be small relative to their lengths and in practice the edit distance between their BWTs also tends to be small. Therefore, if we store the FM-index for one genome normally, we can use Belazzougui et al.'s result to save space when storing FM-indexes for other genomes from the same species (at the cost of higher query times).

It is possible to support nearly all the functionality of an FM-index without using select queries on the underlying BWT, so Belazzougui et al. did not consider relative select. When the FM-index is used in a compressed suffix tree, however, select queries are needed for computing suffix links and for certain other operations. Our interest in relative select comes from Bowe et al.'s [3] (see also [2]) compressed representation of de Bruijn graphs — which is based on something like an FM-index and uses select queries to find nodes' predecessors, and which we call the BOSS representation for the authors' initials — and the possibility of extending it to coloured de Bruijn graphs. Our plan for future work is to view a coloured de Bruijn graph as a union of normal de Bruijn graphs, and relatively compress the BOSS representations of those graphs. Due to space constraints, we provide a brief summary of the BOSS representation and coloured de Bruijn graphs as an appendix. In Section 2 we describe how we implement relative select, and in Section 3 we show experimentally that our implementation is practical. For simplicity and because we are interested mainly in working with DNA, we assume throughout that the size of the alphabet is constant, and we work in the word-RAM model with $\Omega(\log(|S_1| + |S_2|))$-bit words.

2 Design

Although our implementation of relative select is made up of steps that are individually very simple, the overall effect might be confusing. To mitigate this, we break our presentation into pieces: first, we consider the case when S_2 is a subsequence of S_1; then, we consider the case when S_2 is a supersequence of S_1; and finally, we combine our solutions for these special cases to obtain a general solution. We close this section with a small example.

Lemma 1. *Given a select data structure for a string S_1, and a subsequence S_2 of S_1, we can store $\mathcal{O}(|S_1| - |S_2|)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1.*

Proof. We store a bitvector $B[1..|S_1|]$ with 1s marking the characters of S_1 that do not appear in S_2. For each distinct character x, we store a bitvector $B_x[1..\mathsf{occ}(x, S_1)]$, where $\mathsf{occ}(x, S_1)$ is the number of occurrences of x in S_1, with

1s marking the occurrences of x in S_1 that do not appear in S_2. We use the same sparse-bitvector implementation as in Section 1, so this takes a total of $\mathcal{O}(|S_1| - |S_2|)$ extra words and lets us compute

$$S_2.\mathsf{select}_x(i) = B.\mathsf{rank}_0(S_1.\mathsf{select}_x(B_x.\mathsf{select}_0(i)))$$

using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1. To see why this equality holds, consider that $B_x.\mathsf{select}_0(i)$ returns the rank in S_1 of the ith x that appears in S_2; $S_1.\mathsf{select}_x(B_x.\mathsf{select}_0(i))$ returns the position of that x in S_1; and $B.\mathsf{rank}_0(S_1.\mathsf{select}_x(B_x.\mathsf{select}_0(i)))$ returns the position of that x in S_2. □

Lemma 2. *Given a select data structure for a string S_1, and a supersequence S_2 of S_1, we can store $\mathcal{O}(|S_2| - |S_1|)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on S_1.*

Proof. We store a bitvector $B[1..|S_2|]$ with 1s marking the characters of S_2 that do not appear in S_1, and a select data structure for the subsequence D of S_2 consisting of those marked characters. For each distinct character x, we store a bitvector $B_x[1..\mathsf{occ}(x, S_2)]$ with 1s marking the occurrences of x in S_2 that do not appear in S_1. We use a sparse-bitvector implementation again, so this takes a total of $\mathcal{O}(|S_2| - |S_1|)$ extra words and lets us compute

$$S_2.\mathsf{select}_x(i) = \begin{cases} B.\mathsf{select}_0(S_1.\mathsf{select}_x(B_x.\mathsf{rank}_0(i))) & \text{if } B_x[i] = 0, \\ B.\mathsf{select}_1(D.\mathsf{select}_x(B_x.\mathsf{rank}_1(i))) & \text{if } B_x[i] = 1. \end{cases}$$

using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on S_1. To see why this equality holds, suppose the ith x in S_2 also appears in S_1, so $B_x[i] = 0$. Consider that $B_x.\mathsf{rank}_0(i)$ returns the rank of that x in S_1; $S_1.\mathsf{select}_x(B_x.\mathsf{rank}_0(i))$ returns the position of that x in S_1; and $B.\mathsf{select}_0(S_1.\mathsf{select}_x(B_x.\mathsf{rank}_0(i)))$ returns the position of that x in S_2. Now suppose the ith x in S_2 does not appear in S_1, so $B_x[i] = 1$. Consider that $B_x.\mathsf{rank}_1(i)$ returns the rank of that x in D; $D.\mathsf{select}_x(B_x.\mathsf{rank}_1(i))$ returns the position of that x in D; and $B.\mathsf{select}_1(D.\mathsf{select}_x(B_x.\mathsf{rank}_1(i)))$ returns the position of that x in S_2. □

Theorem 1. *Given a select data structure for a string S_1, and another string S_2, we can store $\mathcal{O}(d)$ extra words, where d is the edit distance between S_1 and S_2, and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1.*

Proof. Consider a sequence of d single-character insertions, deletions and substitutions that turns S_1 into S_2. Let C be the common subsequence of S_1 and S_2 consisting of characters left unchanged by these d edits (or a longer common subsequence if we can find one). By Lemma 1, we can store $\mathcal{O}(d)$ extra words and support any select query on C using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1. By Lemma 2, we can then store $\mathcal{O}(d)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on C. Therefore, we can store $\mathcal{O}(d)$ extra words on top of the select data structure for S_1 and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1. □

For example, consider the strings $S_1 = \mathsf{TCTGCGTAAAAGGTGC}$ and $S_2 = \mathsf{TGCTCGTAAAACGCG}$ (the BWTs of $\mathsf{GCACTTAGAGGTCAGT}$ and $\mathsf{GCACTAGACGTCAGT}$, respectively, from the running example in Belazzougui et al.'s paper). Their edit distance is 5 and their longest common subsequence is $C = \mathsf{TCTCGTAAAAGG}$. If we already have a select data structure for S_1 and we want one for S_2, we first add support for relative select on C by the bitvectors $B, B_{\mathsf{A}}, \ldots, B_{\mathsf{T}}$, shown below; then we add support for relative select on S_2 by storing bitvectors $B', B'_{\mathsf{A}}, \ldots, B'_{\mathsf{T}}$, also shown below, and a select data structure for $D = \mathsf{GCC}$. We note that if we have a relative FM-index for S_2 with respect to S_1, then it already includes B, B' and D.

$$B[1..16] = 0001000000010101 \qquad B'[1..15] = 010000000001010$$

$$B_{\mathsf{A}}[1..4] = 0000 \qquad B'_{\mathsf{A}}[1..4] = 0000$$
$$B_{\mathsf{C}}[1..3] = 001 \qquad B'_{\mathsf{C}}[1..4] = 0011$$
$$B_{\mathsf{G}}[1..5] = 10100 \qquad B'_{\mathsf{G}}[1..4] = 1000$$
$$B_{\mathsf{T}}[1..4] = 0001 \qquad B'_{\mathsf{T}}[1..3] = 000$$

To compute $S_2.\mathsf{select}_{\mathsf{C}}(4)$, for instance, we check $B'_{\mathsf{C}}[4]$ and see it is 1, meaning the fourth C in S_2 does not appear in C. Since $B'_{\mathsf{C}}.\mathsf{rank}_1(4) = 2$, it is the second C in D. Since $D.\mathsf{select}_{\mathsf{C}}(2) = 3$, it is the third character in D. Finally, since $B'_1.\mathsf{select}_1(3) = 14$, it is the 14th character in S_2, meaning $S_2.\mathsf{select}_{\mathsf{C}}(4) = 14$.

To compute $S_2.\mathsf{select}_{\mathsf{G}}(3)$, we check $B'_{\mathsf{G}}[3]$ and see it is 0, meaning the third G in S_2 also appears in C. Since $B'_{\mathsf{G}}.\mathsf{rank}_0(3) = 2$, it is the second G in C. Since

$$C.\mathsf{select}_{\mathsf{G}}(2) = B.\mathsf{rank}_0(S_1.\mathsf{select}_{\mathsf{G}}(B_{\mathsf{G}}.\mathsf{select}_0(2))) = 11\,,$$

it is the 11th character in C. Finally, since $B'_1.\mathsf{select}_0(11) = 13$, it is the 13th character in S_2, meaning $S_2.\mathsf{select}_{\mathsf{G}}(3) = 13$.

3 Experiments

We augmented our implementation of the Relative FM-index with the new select structure.[6] The implementation is written in C++ and based on the Succinct Data Structures Library 2.0 [6]. We used g++ version 4.8.1 to compile the code, and ran the experiments on a system with two 16-core AMD Opteron 6378 processors and Linux kernel 2.6.32. We used a single core for the query tests.

As our reference sequence, we used the 1000 Genomes Project's version of the GRCh37 human reference genome, both with (3.096 Gbp) and without (3.036 Gbp) chromosome Y. For a target sequence, we chose the maternal haplotypes of the 1000 Genomes Project's individual NA12878 (3.036 Gbp) [11]. We built a plain FM-index for the reference sequences and the target sequence, as well as relative FM-indexes for the target sequence relative to both references and with and without structures for relative select; the lengths of the common subsequences used were 2.992 Gbp and 2.991 Gbp, respectively. In all cases, we used

[6] https://github.com/jltsiren/relative-fm

Table 1. Average query times for 100 million random LF and Ψ queries on NA12878 stored relative to the human reference genome, with and without chromosome Y.

	FM-index			**Relative FM-index**			**+ Relative Select**	
ChrY	**space**	LF	Ψ	**space**	LF	Ψ	**total space**	Ψ
yes	1090 MB	0.55 µs	1.22 µs	218 MB	3.95 µs	48.0 µs	382 MB	6.11 µs
no	1090 MB	0.55 µs	1.11 µs	181 MB	3.84 µs	44.8 µs	331 MB	6.12 µs

plain bitvectors in the wavelet trees and entropy-compressed bitvectors [10] for marking the common subsequences.

To test the performance of relative select, we ran 100 million random $\Psi(i) = \mathsf{BWT.select}_c(i - \mathsf{C}[c])$ queries on the BWT of the target sequence, using a plain FM-index and Relative FM-indexes with and without relative select. (Character c is the ith character in the BWT in sorted order, while $\mathsf{C}[c]$ is the number of occurrences of characters smaller than c in the BWT.) The implementation of Ψ in the Relative FM-index without relative select was based on binary searching with rank queries. As a comparison, we also ran $\mathsf{LF}(i) = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{BWT.rank}_{\mathsf{BWT}[i]}(i)$ queries. Table 1 shows the results: the relative FM-indexes without relative select are each about a fifth the size of the normal FM-indexes but rank queries are about seven times slower and select queries are about forty times slower; the relative FM-indexes with relative select are about a third the size of the normal FM-indexes but select queries are only about five times slower (rank queries are unaffected).

References

1. Belazzougui, D., Gagie, T., Gog, S., Manzini, G., Sirén, J.: Relative FM-indexes. In: Proc. SPIRE. pp. 52–64 (2014)
2. Boucher, C., Bowe, A., Gagie, T., Puglisi, S., Sadakane, K.: Variable-order de Bruijn graphs. In: Proc. DCC (2015)
3. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: Proc. WABI. pp. 225–235 (2012)
4. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
5. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52, 552–581 (2005)
6. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Proc. SEA. pp. 326–337 (2014)
7. Idury, R., Waterman, M.: A new algorithm for DNA sequence assembly. J. Comput. Biol. 2, 291–306 (1995)
8. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nature Genetics 44, 226–232 (2012)
9. Pevzner, P., Tang, H., Waterman, M.: An Eulerian path approach to DNA fragment assembly. Proc. Nat. Acad. Sci. 98, 9748–9753 (2001)

10. Raman, R., Raman, V., Rao Satti, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3, 43 (2007)
11. Rozowsky, J., et al.: AlleleSeq: analysis of allele-specific expression and binding in a network framework. Molecular Systems Biology 7, 522 (2011)

A de Bruijn Graphs

In biology, the (edge-centric) *kth-order de Bruijn graph* for a set of strings (e.g., DNA reads) is the graph whose nodes are those strings' k-mers (substrings of length k), with a directed edge (u, v) from u to v if at least one of the strings contains a substring of length $k+1$ with u as a prefix and v as a suffix. We label (u, v) with the last character of v. Almost all state-of-the-art DNA assemblers build contigs via Eulerian assembly [7, 9] on de Bruijn graphs, making their space- and time-efficient representation an important problem in bioinformatics.

Bowe et al. add certain dummy nodes and edges, sort the edges into the right-to-left lexicographic order of the nodes they leave, and take the last column of the matrix whose rows are the edges in sorted order (or, equivalently, take the last character in each edge). The result is like a BWT in which edges correspond to characters and nodes correspond to the substrings containing all their out-edges' characters. For example, for the string TACGTCGACGACT and $k = 3$, Bowe et al. derive the edge-BWT TCCGTGGATAA$C. (This example is from [2].) With some auxiliary data structures, we can use rank and select queries on this edge-BWT to navigate forward and backward in the graph.

For the two strings TACGTCGACGACT and TACGACGCGACT and $k = 3$, the de Bruijn graph is 2 nodes larger than the graphs for strings separately. If we store whether each edge occurs in the first string, the second string, or both, then the result is a *coloured de Bruijn graph*. Coloured de Bruijn graphs were introduced by Iqbal et al. [8] for detecting variations between individuals' genomes, and are now also used in other areas of genomics. We can view the coloured de Bruijn graph as the union of each graph consisting of edges of the same colour. In a future paper we will show how to combine the BOSS representations of the individual de Bruijn graphs to obtain a representation of the coloured de Bruijn graph, and also how to relatively compress the auxiliary data structures for the BOSS representations of the individual graphs.

We can use Belazzougui et al.'s result to relatively compress the edge-BWTs of the individual graphs while still supporting rank over them. For example, the edge-BWTs for TACGTCGACGACT and TACGACGCGACT with $k = 3$ are TCCGTGGATAA$C and TCCGTGGACAA$, respectively. They are so close — edit distance 2 — because most of the strings' 4-tuples are common to both and, thus, most of their de Bruijn graphs' edges are common to both. We note that, for reasonable values of k, most of the $(k + 1)$-mers in genomes from the same species should also be common to most of the genomes. In this paper we showed how to support relative select on similar strings, which we will eventually need to navigate backward across edges in our representation of coloured de Bruijn graphs.