Compressed Spaced Suffix Arrays*

Travis Gagie[1], Giovanni Manzini[2], and Daniel Valenzuela[1,3]

[1]Helsinki Institute for Information Technology and University of Helsinki, Finland.
[2]University of Eastern Piedmont, Italy.
[3]University of Helsinki, Finland.

August, 2015

Abstract

As a first step in designing relatively-compressed data structures — i.e., such that storing an instance for one dataset helps us store instances for similar datasets — we consider how to compresses spaced suffix arrays relative to normal suffix arrays and still support fast access to them. This problem is of practical interest when performing similarity search with spaced seeds because using several seeds in parallel significantly improves their performance, but with existing approaches we keep a separate linear-space hash table or spaced suffix array for each seed. We first prove a theoretical upper bound on the space needed to store an SSA when we already have the SA. We then present experiments indicating that our approach works even better in practice.

1 Introduction

DNA sequencing technology has advanced much more rapidly than either computing power or storage, so bioinformaticians now have more data than they can handle using traditional algorithms and data structures. Fortunately, much of this data is repetitive and, thus, highly compressible. This raises the question of when, given an instance of a data structure for one dataset — e.g., a suffix array, FM-index or suffix tree — we can build and store instances of that data structure for similar datasets using less time or space. In this paper we study how to compress spaced suffix arrays (SSAs) relative to normal suffix arrays (SAs) and still support fast random access to them. This problem seems a promising starting point for this line of research because, first, the problem has independent practical interest and, second, we can prove interesting theoretical bounds with few assumptions about the data.

In Section 2 we review spaced seeds, spaced suffix arrays and how they are used for similarity search on DNA sequences. In Section 3 we prove a theoretical bound on the space needed to store an SSA when we already have the SA, in terms of the text's length, the alphabet's size, and the properties of the spaced seed used to define the SSA. Along the way, we improve the query-time bound for Barbay et al.'s [1] compressed permutations. In Section 4 we present experiments showing that in practice our approach works even better than expected. That is, even when we implement our

*Postprint version. The final publication is available at Springer via http://dx.doi.org/10.1007/s11786-016-0283-z

data structures using simpler, theoretically sub-optimal components, we achieve better compression than our upper bound predicts. Finally, in Section 5 we briefly summarize subsequent results we have obtained for relative FM-indexes and related relative data structures since publishing the conference version of this paper.

2 Spaced Seeds and Spaced Suffix Arrays

For the problem of similarity search on DNA sequences, we are given two texts and asked to find each sufficiently long substring of the first text that is within a certain Hamming distance of some substring of the second text. (We are interested in Hamming distance and not the more general edit distance since, e.g., insertions and deletions are rarer than substitutions because they throw off the three-by-three coding of amino acids and are more likely to be deleterious.) Similarity search has many applications in bioinformatics — e.g., ortholog detection, structure prediction or determining rearrangements — and has been extensively studied (see, e.g., [24]). Researchers usually first look for short substrings of the first text that occur unchanged in the second text, called seeds, then try to extend these short, exact matches in either direction to obtain longer, approximate matches. This approach is called, naturally enough, "seed and extend". The substrings' exact matches are found using either a hash table of the substrings with the right length, or an index structure such as a suffix array (SA).

Around the turn of the millenium, Burkhardt and Kärkkäinen [8] and Ma, Tromp and Li [21] independently proposed looking for short *subsequences* of the first text that have a certain shape and occur unchanged in the second text, and trying to extend those. A binary string encoding the shape of a subsequence, with 1s indicating positions where the characters must match and 0s indicating positions where they need not, is called a spaced seed. The total number of bits in the binary string is called the seed's length, and the number of 1s is called its weight. The subsequences' exact matches are found using either a hash table of the subsequences with the right shape, or a kind of modified SA called a spaced suffix array [19] (SSA).

Peterlongo et al. [22] and Crochemore and Tischler [10] independently defined SSAs, under the names "bi-factor arrays" and "gapped suffix arrays", for the special case in which the spaced seed has the form $1^a0^b1^c$. Russo and Tischler [23] showed how to represent such an SSA in asymptotically succinct space such that we can support random access to it in time logarithmic in the length of the text. We note, however, that the spaced seeds used for most applications do not have this form. Battaglia et al. [2] used an idea similar to that of spaced seeds in an algorithm for finding motifs with don't-care symbols.

Burkhardt and Kärkkäinen, Ma et al. and subsequent authors have shown that using spaced seeds significantly improves the performance of seeding and extending. Many papers have been written about how to design spaced seeds to minimize the number of errors (see, e.g., [7, 12, 17] and references therein), with the specifics depending on the model of sequence similarity and the acceptable numbers of false positives (for which the characters indicated by 1s all match but the substrings are not similar) and false negatives (for which those characters do not all match but the substrings are still similar) for the application in question. Regardless of the particular application, however, researchers have consistently observed that the best results are obtained using more than one seed at a time. A set of spaced seeds used in combination is called a multiple seed.

Multiple seeds are now a popular and powerful tool for similarity search, but they have a lingering flaw: we keep a hash table or SSA for each seed, and each instance of these data structures takes

linear space. For example, SHRiMP2's [11] index for the human genome takes 16 GB *for each seed*. In contrast, Bowtie 2's [20] compressed SA for that genome takes only 2.5 GB. This is because a normal SA (which supports only substring matching) can be compressed such that the number of bits per character is only slightly greater than the empirical entropy of the text. Unfortunately, the techniques for compressing normal SAs do not seem to apply directly to SSAs.

Whereas the normal SA for a text lists the starting points of the suffixes of that text by those suffixes' lexicographic order, the SSA for a text and a spaced seed lists the starting points of the subsequences with the right shape by those subsequences' lexicographic order. Intuitively, if the seed starts with many 1s, the SSA will be similar to the SA. In the next section we formalize this intuition and prove a theoretical upper bound on the space needed to store an SSA when we already have the SA, in terms of the text's length, the alphabet's size, and the seed's weight and length.

3 Theory

Suppose we want to store an SSA for a text $T[0..n-1]$ over an alphabet of size σ and a spaced seed S with length ℓ and weight w. For $i < n$, let T_i be the subsequence of $T[i..n-1]$ that contains $T[j]$ if and only if $i \leq j$ and $S[j-i] = 1$. Let T'_i be the subsequence of $T[i..n-1]$ that contains $T[j]$ if and only if $i \leq j$ and $S[j-i] = 0$. Let SSA be the permutation on $\{0, \ldots, n-1\}$ in which i precedes i' if either $T_i \prec T_{i'}$, where $\prec$ indicates lexicographic precedence, or $T_i = T_{i'}$ and $T[i..n-1] \prec T[i'..n-1]$.

For example, if $T = \mathsf{abracadabra}$ and $S = 101$ then

T_0	= ar	T_6	= db	T'_0	= b	T'_6	= a
T_1	= ba	T_7	= ar	T'_1	= r	T'_7	= b
T_2	= rc	T_8	= ba	T'_2	= a	T'_8	= r
T_3	= aa	T_9	= r	T'_3	= c	T'_9	= a
T_4	= cd	T_{10}	= a	T'_4	= a		
T_5	= aa			T'_5	= d		

and so, since

$$T_{10} \prec T_3 = T_5 \prec T_0 = T_7 \prec T_1 = T_8 \prec T_4 \prec T_6 \prec T_9 \prec T_2$$

and $T[3..10] \prec T[5..10]$, $T[7..10] \prec T[0..10]$ and $T[8..10] \prec T[1..10]$, we have SSA $= [10, 3, 5, 7, 0, 8, 1, 4, 6, 9, 2]$ while SA $= [10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]$.

If $T_i \preceq T_{i'}$ and $T'_i \preceq T'_{i'}$, then i precedes i' in both SSA and SA. In particular, if $T_i = T_{i'}$ or $T'_i = T'_{i'}$, then i and i' have the same relative order in SSA and SA. In our example, $T_3 = T_5 = \mathsf{aa}$ and $T[3..10] \prec T[5..10]$, so 3 precedes 5 in both SSA and SA; $T'_2 = T'_4 = T'_6 = T'_9 = \mathsf{a}$ and $T[4..10] \prec T[6..10] \prec T[9..10] \prec T[2..10]$, so 4 precedes 6, 6 precedes 9 and 9 precedes 2 in both SSA and SA.

If we partition SSA into subsequences such that i and i' are in the same subsequence if and only if $T_i = T_{i'}$, then we can partition SA into the same subsequences. Since there are at most $\sigma^w + w$ distinct strings T_i, our partitions each consist of at most $\sigma^w + w$ subsequences. Similarly, if we partition based on T'_i and $T'_{i'}$, then our partitions each consist of at most $\sigma^{\ell-w} + \ell - w$ subsequences.

For our example, we can partition both SSA and SA into $[4, 6, 9, 2]$, for $T'_i = \mathsf{a}$; $[7, 0]$, for $T'_i = \mathsf{b}$; $[3]$, for $T'_i = \mathsf{c}$; $[5]$, for $T'_i = \mathsf{d}$; $[8, 1]$, for $T'_i = \mathsf{r}$; and $[10]$, for $T'_i = \epsilon$. In this particular case, however, we can just as well partition both SSA and SA into only two common subsequences: e.g., $[10, 7, 0]$ and $[3, 5, 8, 1, 4, 6, 9, 2]$.

Observation 1. *Let* SA *be the suffix array for a text* $T[0..n-1]$ *over an alphabet of size* σ *and let* SSA *be the spaced suffix array for* T *and a spaced seed* S *with length* ℓ *and weight* w*. We can partition* SA *and* SSA *into at most* $\min(\sigma^w + w, \sigma^{\ell-w} + \ell - w)$ *common subsequences.*

Consider the permutation $\mathrm{SA}^{-1} \circ \mathrm{SSA}$, which maps elements' positions in SSA to their positions in SA, and let ρ be the minimum number of increasing subsequences into which $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ can be partitioned. Since any subsequence common to SSA and SA corresponds to an increasing subsequence in $\mathrm{SA}^{-1} \circ \mathrm{SSA}$, we have $\rho \leq \min(\sigma^w + w, \sigma^{\ell-w} + \ell - w)$. In our example, $\mathrm{SA}^{-1} \circ \mathrm{SSA} = [0, 3, 4, 1, 2, 5, 6, 7, 8, 9, 10]$ and $\rho = 2$.

Supowit [25] gave a simple algorithm that partitions $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ into ρ increasing subsequences in $\mathcal{O}(n \lg \rho) \subseteq \mathcal{O}(n \min(w, \ell - w) \lg \sigma)$ time. When applied to $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ in our example, Supowit's algorithm partitions it into $[0, 3, 4, 5, 6, 7, 8, 9, 10]$ and $[1, 2]$. Barbay et al. [1] showed how, given a partition of $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ into ρ increasing subsequences, we can store it in $(2 + o(1))n \lg \rho \leq (2 + o(1))n \min(w, \ell - w) \lg \sigma$ bits and support random access to it in $\mathcal{O}(\lg \lg \rho)$ time. We now improve their query-time bound to $\mathcal{O}(1)$.

Lemma 2. *Given a partition of a permutation* π *on* $\{0, \ldots, n-1\}$ *into* ρ *increasing subsequences, we can store* π *in* $(2 + o(1))n \lg \rho$ *bits and support random access to it in constant time.*

Proof. For $i \leq \rho$, we replace each element in the ith subsequence by a character a_i and store the resulting string R such that we can support random access to it and partial rank queries on it in constant time. We then permute R according to π and store the resulting string R' such that we can support fast select queries on it. The partial rank query $R.\mathrm{p\_rank}(i)$ returns the number of copies of $R[i]$ in $R[0..i]$, and the select query $R'.\mathrm{select}_a(i)$ returns the position of the ith copy of a in R'. If we use the data structures by Belazzougui and Navarro [4], then we use a total of $(2 + o(1))n \lg \rho$ bits.

Barbay et al. noted that, for $i < n$,

$$\pi[i] = R'.\mathrm{select}_{R[i]}(R.\mathrm{p\_rank}(i)) \; .$$

For example, if $\pi = [0, 3, 4, 1, 2, 5, 6, 7, 8, 9, 10]$ — i.e., $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ from our running example — then $R = a_1a_1a_1a_2a_2a_1a_1a_1a_1a_1a_1$ and $R' = a_1a_2a_2a_1a_1a_1a_1a_1a_1a_1a_1$ and

$$\pi[4] = R'.\mathrm{select}_{R[4]}(R.\mathrm{p\_rank}(4)) = R'.\mathrm{select}_{a_2}(2) = 2 \, .$$

□

In summary, by Observation 1 and Lemma 2, we can store $\mathrm{SA}^{-1} \circ \mathrm{SSA}$ in $(2 + o(1))n \min(w, \ell - w) \lg \sigma$ bits such that we can support random access to it in $\mathcal{O}(1)$ time. Since $\mathrm{SSA} = \mathrm{SA} \circ (\mathrm{SA}^{-1} \circ \mathrm{SSA})$, this gives us the following result:

Theorem 3. *Let* $T[0..n-1]$ *be a text over an alphabet of size* σ *and let* S *be a spaced seed with length* ℓ *and weight* w*. If we have already stored the suffix array* SA *for* T *such that we can support random access to* SA *in time* t_{SA}*, then we can store a spaced suffix array* SSA *for* T *and* S *in* $(2 + o(1))n \min(w, \ell - w) \lg \sigma$ *bits such that we can support random access to* SSA *in* $t_{\mathrm{SA}} + \mathcal{O}(1)$ *time.*

4 Practice

Theorem 3 says we can store SSAs for the human Y-chromosome `chrY.fa` in FASTA format (about 60 million characters over an alphabet of size 5) and SHRiMP2's three default spaced seeds — i.e., 11110111101111, 1111011100100001111 and 1111000011001101111 — in about 560 MB, in addition to the SA, whereas storing the SSAs using four bytes per entry would take about 720 MB. Storing the SSAs packed such that each entry takes $\lceil \lg 60\,000\,000 \rceil = 26$ bits would reduce this from to about 580 MB.

To test our approach, we built the SSAs as described in Section 3; computed $\mathrm{SA}^{-1} \circ SSA$, and the strings R and R' from Lemma 2 for each SSA; and stored each copy of R or R' as a wavelet tree. We chose wavelet trees because they are simple to use and often more practical than Belazzougui and Navarro's theoretically smaller and faster data structures mentioned in Section 3. We ran all our tests described in this section on a computer with a quad-core Intel Xeon CPU with 32 GB of RAM, running Ubuntu 12.04. We used a wavelet-tree implementation from `https://github.com/fclaude/libcds` and compiled it with GNU `g++` version 4.4.3 with optimization flag `-O3`.

The uncompressed SA took 226 MB, and the six wavelet trees took a total of 215 MB and performed 10 000 random accesses each in 7.67 microseconds per access. That is, we compressed the SSAs (including the uncompressed SA) into about 60% of the space it would take to store them using four bytes per entry, or about 75% of the space it would take to store them packed. Although our accesses were much slower than direct memory accesses, they were fast compared to disk accesses. Thus, our approach seems likely to be useful when a set of SSAs is slightly larger than the memory and fits only when compressed.

Using the same test setup, we then compressed SSAs for the ten spaced seeds BFAST [16, Table S3] uses for 36-base-pair Illumina reads, which all have weight 18:

1. 111111111111111111
2. 111101001101111010101011111
3. 11111111111111001111
4. 111101110110010100111111 1
5. 111101110001010100000101011101 11
6. 101100110101111010011001001 0111
7. 11101100101000010001011001 11001111
8. 111101111111111111
9. 110111111000101101111011 01
10. 111010001110001110100011011111 .

Since the first seed consists only of 1s, the SSA we would build for it is the same as the SA. The uncompressed SA again took 226 MB and the 18 wavelet trees for the other nine seeds took a total of 649 MB — so instead of 2.26 GB, we used 875 MB (about 39%) for all ten seeds — and together performed 10 000 random accesses to each of the ten SSAs in about 7 microseconds per access. The left side of the top half of Figure 1 shows how many bits per character (bpc) of the text each SSA took, and the average time per access to each SSA.

We also compressed the SSAs for the ten spaced seeds BFAST uses for 50-base-pair Illumina reads, which all have weight 22:

1. 11111111111111111111
2. 11111011101110101001011011111
3. 10111101011010010110000110100011111111
4. 10111001101001100100111101010001011111
5. 11111011011101111011111111
6. 11111110010100100010111101110111
7. 11110101110010100010101101010111111
8. 11110110101101100110000010110100101101
9. 11110110100010001101011001011001101001 11
10. 11110100101101101011100101101110 11 .

Again, the first seed consists only of 1s. This time, the 18 wavelet trees for the other nine seeds took a total of 712 MB; each access took about 8 microseconds. The left side of the lower half of Figure 1 shows how many bits per character of the text each SSA took, and the average access time per access to each SSA.

Notice that, if we have a permutation π_1 on $\{0, \ldots, n-1\}$ stored and π_2 is any other permutation on $\{0, \ldots, n-1\}$, then we can store π_2 relative to π_1 using the ideas from Section 3. For example, we can store SSAs relative to other SSAs. Suppose we consider the size of each SSA (except the SA) when compressed relative to each other SSA (including the SA); view these sizes as edge costs in a complete graph whose nodes are the SSAs; build a minimum spanning tree rooted at the SA; and compress each SSA relative to its parent in the tree. This can reduce our space usage at the cost of increasing the random-access time, as shown for the BFAST seeds on the right side of Figure 1. We leave further exploration of such tradeoffs as future work.

5 Subsequent Research

In this paper we have shown how to compress spaced suffix arrays relative to normal suffix arrays while still supporting fast random access to them. Since publishing the conference version of this paper [14], we have investigated other relative data structures, particularly FM-indexes and related data structures.

An FM-index [13] for a text T is essentially a rank data structure on the Burrows-Wheeler Transform [9] (BWT) of T. Intuitively, if T and T' are similar texts then in practice BWT(T) and BWT(T') should be similar as well. In our second paper on relative data structures [3] we showed how, given a rank data structure for BWT(T), we can store a rank data structure for BWT(T') in small space. This means that in practice, given an FM-index for T, we can store an FM-index for T' in small space.

Bowe et al. [6] gave a space-efficient implementation of de Bruijn graphs based on a BWT-like permutation of the edge labels. In addition to rank, they used select to traverse edges backwards. In our third paper on relative data structures [5], we showed how to implement a relative select data structure and, building on that, a relative de Bruijn graph data structure. Using these ideas, we are currently implementing a space-efficient data structure for coloured de Bruijn graphs [18].

We recently implemented and tested [15] a relative compressed suffix tree data structure. We refer interested readers to that paper for an up-to-date discussion of our continuing work in this area.

Acknowledgments

Many thanks to Francisco Claude, Maxime Crochemore, Matei David, Martin Frith, Costas Iliopoulos, Juha Kärkkäinen, Gregory Kucherov, Bin Ma, Ian Munro, Taku Onodera, Gonzalo Navarro, Luis Russo, German Tischler and the anonymous reviewers. The first author was partly funded by Academy of Finland grant 258308, and the first and second authors were partly funded by Academy of Finland grant 250345 (CoECGR).

References

[1] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69:232–268, 2014.

[2] G. Battaglia, D. Cangelosi, R. Grossi, and N. Pisanti. Masking patterns in sequences: A new class of motif discovery with don't cares. *Theoretical Computer Science*, 410:4327–4340, 2009.

[3] D. Belazzougui, T. Gagie, S. Gog, G. Manzini, and J. Sirén. Relative FM-indexes. In *Proceedings of the 21st Symposium on String Processing and Information Retrieval (SPIRE)*, pages 52–64, 2014.

[4] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 11(4), 2015.

[5] C. Boucher, A. Bowe, T. Gagie, G. Manzini, and J. Sirén. Relative select. In *Proceedings of the 22nd Symposium on String Processing and Information Retrieval (SPIRE)*, 2015. To appear.

[6] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. In *Proceedings of the 12th Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012.

[7] D. G. Brown. A survey of seeding for sequence alignment. In I. Măndoiu and A. Zelikovsky, editors, *Bioinformatics Algorithms: Techniques and Applications*, pages 126–152. Wiley-Interscience, 2008.

[8] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informicae*, 56:51–70, 2003.

[9] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[10] M. Crochemore and G. Tischler. The gapped suffix array: A new index structure for fast approximate matching. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 359–364, 2010.

[11] M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno. SHRiMP2: Sensitive yet practical short read mapping. *Bioinformatics*, 27:1011–1012, 2011.

[12] L. Egidi and G. Manzini. Better spaced seeds using quadratic residues. *Journal of Compututer and System Sciences*, 79:1144–1155, 2013.

[13] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52:552–581, 2005.

[14] T. Gagie, G. Manzini, and D. Valenzuela. Compressed spaced suffix arrays. In *Proceedings of the 2nd International Conference on Algorithms for Big Data (ICABD)*, pages 37–45, 2014.

[15] T. Gagie, G. Navarro, S. J. Puglisi, and J. Sirén. Relative compressed suffix trees. Technical Report 1508.02550, arxiv.org, 2015.

[16] N. Homer, B. Merriman, and S. F. Nelson. BFAST: An alignment tool for large scale genome resequencing. *PLOS One*, 4:e7767, 2009.

[17] L. Ilie, S. Ilie, S. Khoshraftar, and A. Mansouri Bigvand. Seeds for effective oligonucleotide design. *BMC Genomics*, 12:280, 2011.

[18] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226–232, 2012.

[19] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, 21:487–493, 2011.

[20] B. Langmeand and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9:357–359, 2012.

[21] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[22] P. Peterlongo, N. Pisanti, F. Boyer, A. Pereira do Lago, and M. Sagot. Lossless filter for multiple repetitions with Hamming distance. *Journal of Discrete Algorithms*, 6(3):497–509, 2008.

[23] L. M. S. Russo and G. Tischler. Succinct gapped suffix arrays. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 290–294, 2011.

[24] Y. Sun and J. Buhler. Designing multiple simultaneous seeds for DNA similarity search. *Journal of Computational Biology*, 12:847–861, 2005.

[25] K. J. Supowit. Decomposing a set of points into chains, with applications to permutation and circle graphs. *Information Processing Letters*, 21:249–252, 1985.

seed	space (bpc)	time (μs)
1	32.00	0
2	11.29	9
3	4.41	4
4	9.75	8
5	11.54	9
6	13.77	11
7	13.14	10
8	3.85	3
9	10.10	7
10	13.91	11

seed	reference	space (bpc)	time (μs)
1	-	32.00	0
2	8	9.71	11
3	1	4.41	4
4	8	9.22	10
5	4	9.23	19
6	8	12.27	14
7	3	12.58	14
8	1	3.85	3
9	1	10.10	7
10	7	12.59	26

seed	space (bpc)	time (μs)
1	32.00	0
2	9.03	8
3	12.30	10
4	13.86	11
5	8.13	7
6	10.80	9
7	11.14	8
8	11.09	8
9	11.77	9
10	12.54	10

seed	reference	space (bpc)	time (μs)
1	-	32.00	0
2	1	9.03	6
3	1	12.30	10
4	2	12.59	18
5	1	8.13	6
6	1	10.80	8
7	1	11.14	8
8	1	11.09	9
9	8	11.34	18
10	8	8.94	17

Figure 1: The space usage of the SSAs of the spaced seeds BFAST uses for Illumina reads, in bits per character of the text, and the average time for a random access. Above, the seeds are for 36-base-pair reads; below, the seeds are for 50-base-pair reads. On the left, all the SSAs are compressed relative to the SA; on the right, some of the SSAs are compressed relative to other SSAs.