The Review of Socionetwork Strategies (2019) 13:227–236
https://doi.org/10.1007/s12626-019-00045-1

ARTICLE

Application-Oriented Succinct Data Structures for Big Data

Tetsuo Shibuya[1]

Received: 21 December 2018 / Accepted: 9 July 2019 / Published online: 9 October 2019

Abstract
A data structure is called succinct if its asymptotical space requirement matches the original data size. The development of succinct data structures is an important factor to deal with the explosively increasing big data. Moreover, wider variations of big data have been produced in various fields recently and there is a substantial need for the development of more application-specific succinct data structures. In this study, we review the recently proposed application-oriented succinct data structures motivated by big data applications in three different fields: privacy-preserving computation in cryptography, genome assembly in bioinformatics, and work space reduction for compressed communications.

Keywords Sublinear paradigm · Succinct data structures · Survey · Oblivious RAM · de Bruijn graph · Compressed communication · Work space reduction

1 Introduction

The amount of data being generated in various fields is increasing rapidly. For example, the amount of DNA data obtained by next-generation sequencers (NGSs) doubles almost every year [19], which is much faster than the pace of Moore's law [27, 53]. Hence, more sophisticated algorithms and data structures are highly desired for big data.

A data structure is said to be succinct if its additional space requirement is sublinear, i.e., $o(n)$, where n is the size of target data. In the last two decades, significant work has been done toward the development of succinct versions of various data structures for manipulating big data. In this paper, we introduce recent work on succinct data structures driven by actual big data applications.

This work was supported by JST CREST Grant number JPMJCR1402, JSPS KAKENHI Grant numbers 17H01693, and 17K20023JST.

✉ Tetsuo Shibuya
tshbiuya@hgc.jp

[1] Tokyo, Japan

One of the most fundamental tasks for big data is to search for a substring in a text database. Traditionally, we can search a substring in $O(n)$ time using the Knuth–Morris–Pratt (KMP) algorithm with $O(m)$ working space, where n is the text size and m is the query size [35]. However, when there are multiple online queries, the KMP algorithm requires $O(k \cdot n)$ time, where k is the number of queries, which may be inefficient for large n. Aho–Corasick algorithm [1] can find all occ occurrences in $O(n + k \cdot m + occ)$ time with $O(k \cdot m)$ working space, but it cannot process queries online. For this purpose, the suffix tree was proposed [22, 51, 73, 75]. It enables $O(m \log \sigma + occ)$-time query after building a $\Theta(n)$-size suffix tree in $O(n)$ time, where σ is the alphabet size. However, the suffix tree requires a large space. Later, the suffix array was proposed, which is much smaller than the suffix tree [34, 36, 46, 55]. FM index based on Burrows–Wheeler transform [13] was then proposed [24], It requires $o(n)$ space in addition to the original text size, which is much smaller than the suffix array. Now, the FM index is known as one of the most famous succinct data structures used in various fields. For example, the FM index is now a necessary component in various large genome projects in molecular biology [40, 45].

There are various indexing problems for various targets other than just texts and many succinct data structures have been proposed for them. For example, the suffix tree of a tree is a data structure for indexing paths on a tree [37, 65], for which XBW was proposed as a succinct counterpart [23]. Another type of succinct data structure was proposed for parameterized indexing [26], where we need to index strings of parameterized characters [3, 66].

Succinct data structures are not restricted to indexing. The topology of a tree can be represented in a very small space if we use balanced parentheses (BP) representation or level-order unary degree sequence (LOUDS) representation [33]. Various succinct data structures have been proposed for them [6, 62] and they can be applied to various data in tree form, e.g., XML data.

Nowadays, there is an increasing demand for succinct data structures in various applications. In this paper, we introduce three examples of application-driven succinct data structure research. The first example is research motivated from privacy-preserving computation in cryptography. Privacy is an important keyword in big data systems, but many current technologies for privacy cannot be easily implemented for big data applications owing to too a large computation time and/or working space. In Sect. 2, we introduce the research on succinct data structures for oblivious RAMs (ORAMs), which are used for hiding access patterns. Another example is on a topic related to genome assembly problem, which is an important bioinformatics topic in current molecular biology. In Sect. 3, we introduce the research on de Bruijn graphs, which are used for genome assembly. The third example is on compressed communication, where the data are compressed and decompressed in real time. In Sect. 4, we introduce research on succinct techniques for online compression.

2 Succinct Oblivious RAM

Currently, there is more emphasis on privacy while dealing with big data. There are various encryption techniques useful for hiding the content of data, but recent studies have revealed that adversaries might learn various things from access patterns [14, 31], implying that there are cases where the encryption of data are not sufficient.

In 1987, Goldreich proposed a seminal concept called ORAM, or Oblivious RAM (Random Access Memory), which can theoretically hide the content of data and their access patterns [28, 29]. Many variations of ORAMs have been proposed since then [2, 15, 20, 38, 57, 58, 70, 71, 74].

In literature, we consider two different situations where ORAMs are used. In one situation, we store data on an insecure cloud database (i.e., accessed positions can be viewed by the adversary) and we access data from a secure client PC. Here, we need to hide our access patterns, i.e., any information related to the order of accessed positions, except for the number of accesses. In the other situation, we store data on an insecure RAM and we access data from a secure CPU. Here, the access patterns on the RAM are hidden. In the following, we use the terms database and client instead of cloud database/RAM and client PC/CPU.

An ORAM is a database simulation where we need to read/write data of n blocks of B bits without leaking the access patterns. Here, we assume that data in blocks are probabilistically encrypted, i.e., the adversary cannot know the content of the block and moreover, the adversary cannot know whether two blocks have the same content.

Most ORAMs are roughly categorized into two types: hash-based ORAMs and tree-based ORAMs. Hash-based ORAMs utilize hierarchical hash techniques based on one-way functions. They include the original Goldreich's Square-Root ORAM (SR-ORAM) [28, 29] and many succeeding ORAMs [2, 38, 58]. In contrast, tree-based ORAMs maintain data in tree data structures [57, 71, 74].

There are several metrics to evaluate the performance of ORAM (Table 1). The most important metric is bandwidth blowup, which is the complexity of the

Table 1 Theoretical performance of various ORAMs

ORAMs	Bandwidth blowup	Client storage (#blocks)	Server storage overhead (#blocks)
Goldreich [28]	$O(\sqrt{n}\log n)$	$O(1)$	$n \cdot \Theta\left(\frac{\log n}{B} + \frac{1}{\sqrt{n}}\right)$
Kushlevitz et al. [38]	$O\left(\frac{\log^2 n}{\log\log n}\right)$	$O(1)$	$n \cdot \Theta(1)$
Stefanov et al. [70]	$O(\log n)$	$O(n)$	$n \cdot \Theta(1)$
Stefanov et al. [71]	$O(\log^2 n)$	$\omega(\log n)$	$n \cdot \Theta(1)$
Onodera et al. [57]	$O(\log^2 n)$	$\omega(\log n)$	$n \cdot \left(\Theta\left(\frac{\log n}{B}\right) + o\left(\frac{\log\log n}{\log n}\right)\right)$
Patel et al. [58]	$O(\log n \cdot \text{poly}(\log\log n))$	$O(1)$	$n \cdot \Theta(1)$
Asharov et al. [2]	$O(\log n)$	$O(1)$	$n \cdot \Theta(1)$

number of actual accesses per one simulated access on the ORAM. It should be noted that a lower bound $\Omega(\log n)$ for the bandwidth blowup is known [11, 28, 41].

Until recently, the best known ORAM was Balanced ORAM (B-ORAM) proposed by Kushlevits et al. [38], whose bandwidth blowup is $O(\log^2 n/\log\log n)$. It should be noted that the B-ORAM has a large constant before its bandwidth blowup complexity [15]. Recently, PanORAMa proposed by Patel et al. [58] achieved $O(\log n \cdot \text{poly}(\log\log n))$ bandwidth blowup. More recently, OptORAMa with tight bandwidth blowup was proposed by Asharov et al. [2]. It should be noted that PanORAMa and OptORAMa are complicated algorithms and rather difficult to implement. In contrast, Path ORAM proposed by Stefanov et al. [71] is said to be one of the most practical ORAMs, with reasonable bandwidth blowup complexity of $O(\log^2 n)$ [15]. In addition, it is easy to implement.

There are two other important metrics, i.e., client storage size and database storage overhead. Client storage size is the data size that a client is allowed to have. Most ORAMs assume the client storage sizes at most $O(\text{polylog } n)$, though there are practical ORAMs that allow larger theoretical complexities [70].

Because we need to store n simulated blocks of data, we require server storage of at least n blocks (of size B) for any type of ORAMs. Database storage overhead is the additional storage size (in number of blocks) required by ORAM in addition to the unavoidable n blocks.

In any known ORAM in literature, each block on the database has additional metadata containing its original address of $O(\log n)$ bits for verifying its correctness. This implies that all known ORAMs have a database storage overhead of $\Omega(n \cdot \log n/B)$ blocks. In other words, we cannot design succinct ORAMs under the assumption that $B = O(\log n)$, while using the current metadata strategy. So far, all the ORAMs that have been proposed assume $\Omega(\log n)$ size blocks, and many ORAMs assume blocks of size $\omega(\log n)$ [57, 71, 74].

An ORAM is said to be succinct if its database storage overhead size is $o(n)$. Unfortunately, most known ORAMs require $\Theta(n)$ database storage overhead. Some ORAMs have even larger storage overhead [29, 64]. As of today, only two succinct ORAMs are known under the assumption that $B = \omega(\log n)$. Until very recently, the SR-ORAM, the first ORAM proposed by Goldreich [28] was the only succinct ORAM. SR-ORAM achieves a database storage overhead of $\Theta((n \cdot \log n/B) + \sqrt{n})$ blocks. However, its bandwidth blowup is almost impractical $(O(\sqrt{n}\log n))$. Recently, succinct ORAM, which is a variation of the Path ORAM [71], is proposed by Onodera et al. [57]. It is the first and the only known succinct ORAM with reasonable $O(\text{polylog } n)$ bandwidth blowup. Practically, the Succinct ORAM requires only two or three times larger storage overhead, while ordinary tree-based ORAMs like Path ORAM require almost ten times the storage overhead [57].

There are several open problems. One question is whether we can design a succinct ORAM with tighter bandwidth blowup. Existing ORAMs with $o(\log^2 n)$ bandwidth blowup are hash-based, and ordinary hashes require $\Theta(n)$ empty slots in nature to keep ORAMs secure and efficient. A succinct hash data structure for a very restricted class of keywords is known [61], but it does not seem to be applicable to ORAMs.

Another question is about the existence of succinct ORAMs with better client storage size, keeping the $O(\text{polylog } n)$ bandwidth blowup. It should be noted that the SR-ORAM requires optimal $O(1)$ client storage, but its bandwidth blowup is large.

Another question that arises is whether a succinct ORAM for $B = O(\log n)$ can be designed. As already stated above, all known ORAMs, including the above two succinct ORAMs (the SR-ORAM and the Succinct ORAM), maintain $O(\log n)$-bit metadata for each block. We need a totally different approach for manipulating metadata to achieve it.

3 Succinct de Bruijn Graph

A tremendous amount of DNA data are obtained in genome science these days with the advent of next-generation sequencers (NGSs). Although NGSs can sequence DNA rapidly with low cost, the current NGSs can read only short sequences and cannot read the entire genome. Therefore, we need to estimate the entire genome sequence with some computational algorithms, considering information from its short substrings called reads. This computational task is called genome assembly and many algorithms have been proposed for it [21, 56, 67].

The de Bruijn graph is a graph data structure used in many recent genome assembly algorithms. The original concept was proposed by de Bruijn for graph theory [12], and Pevzner et al. used a variation of the graph for genome assembly [60]. It should be noted that Pevzner et al.'s definition of a de Bruijn graph is different from the original one by de Bruijn. We follow Pevzner et al.'s definition below.

In the following, $S[i]$ denotes the ith character of string S, $S[i \dots j]$ denotes the region of string S $(= S[1 \dots |S|])$ that starts at the ith position and ends at the jth position, and $S + a$ denotes a string obtained by adding character a after string S.

A string of length k is called a k-mer in bioinformatics. Let $\mathcal{M} = \{M_1, M_2, \dots, M_n\}$ be a set of n different k-mers. Let $\mathcal{M}^- = \left\{M_1^-, M_2^-, \dots, M_{|M^-|}^-\right\}$ denote the set of all the $(k-1)$-mers that appear as substrings of strings in $\mathcal{M}$. The de Bruijn graph of $\mathcal{M}$ is defined as the directed graph $G_{\mathcal{M}} = \left\{\mathcal{M}^-, E_{\mathcal{M}}\right\}$, where $E_{\mathcal{M}}$ is the set of all the ordered pairs of strings (M_i^-, M_j^-), such that $M_i^-[2 \dots k-1] = M_j^-[1 \dots k-2]$ and $M_i^-[1 \dots k-1] + M_j^-[k-1] \in \mathcal{M}$.

Ideally, we can enumerate all k-mers that occur in the target genome with an NGS if we assume an ideal case where the sequenced reads ideally cover the entire genome with no errors (i.e., any k-mer substring of the entire genome appears as a substring of some read, and each read has no errors). Then, a simplified genome assembly problem can be considered as the following.

Ideal k-mer Assembly Problem Given a set $\mathcal{M}$ of k-mers, find the shortest string which contains k-mers in $\mathcal{M}$ but no other k-mers as its substrings.

We can solve this problem by computing an Eulerian path (i.e., one of the shortest paths that uses all edges on the graph) on the de Bruijn graph of $\mathcal{M}$; in case each k-mer in $\mathcal{M}$ occurs exactly once in the shortest string [60]. It should be noted that we can compute an Eulerian path on a general graph in time linear to the graph size.

Most recent genome assemblers use the de Bruijn graph for a set of all k-mers on all reads (or those on all screened confident reads) obtained by NGSs. To store a de Bruijn graph naively, we require space of $\Omega(n \cdot (k + \log n))$ bits, where n is the number of edges in the de Bruijn graph, as each edge requires $\Theta(k)$ bits for storing its label and $\Theta(\log n)$ bits for storing a pointer to its next node. It becomes prohibitively large for very large genomes. For example, they required > 4.3 TB memory for storing distributed de Bruijn graphs to assemble the 20 Gbp genome of white spruce [7]. In cases of metagenome analyses, we often need to deal with even larger data [30].

Several approaches have been proposed to reduce the memory requirement of de Bruijn graphs. The first compact representation was proposed by Conway et al. [18], where they represented the graph with some compressed bit encoding. They succeeded in storing a de Bruijn graph ($k = 27$) with 12, 292, 819, 311 edges in 40.8 GB space, i.e., 28.5 bits per edge, which is significant improvement over naive implementation.

Ye et al. proposed another approach to use a sparse sampled graph instead of the entire de Bruijn graph [77]. Pell et al. [59] proposed another heuristic approach based on Bloom filter [8]. They utilized the Bloom filter to heuristically represent de Bruijn graphs, where the represented graph is not exactly same as the original de Bruijn graph. Chikhi et al. improved this Bloom filter-based approach to represent de Bruijn graphs without any errors [16, 17]. They succeeded in storing a de Bruijn graph in $O(n \log k)$ bits, which is a theoretical improvement over the naive implementation. This strategy is used in the assembler ABySS [32].

A succinct data structure for de Bruijn graphs, called the succinct de Bruijn graph, is proposed by Bowe et al. [10]. They extended the XBW data structure [23], so that it can represent de Bruijn graphs. This data structure can store a de Bruijn graph in just $4n + o(n)$ bits, which is usually lesser than 5 bits per edge, and is independent from k. The succinct de Bruijn graph is used in the assembler MegaHIT [43, 44]. There are several extensions of the succinct de Bruijn graph, e.g., succinct data structures for variable-order de Bruijn graphs [5, 9], dynamic de Bruijn graphs [4], colored de Bruijn graphs [54], and genome graphs [69].

However, this implementation raises some questions. One question is about the existence of distributed succinct representations of de Bruijn graphs. As related work, there is an attempt to store non-succinct de Bruijn graphs in distributed space [68]. Note also that there is an attempt to build FM index in parallel in shared memory [39].

A class of graphs (called Wheeler graphs) that can be succinctly indexed by BWT-related techniques is discussed in [25]. Another question is whether we can expand the class of succinctly representable graphs.

4 Succinct Schemes for Compressed Communications

There is a need to transmit a tremendous amount of data between different sites. We can reduce communications by just compressing data, but we need to pay costs for compressing/decompressing data before/after communication. It becomes a problem in IoT communications where we only have small hardware with very small working space. Hence, we need to design efficient online algorithms for compressing/decompressing data with small restricted working space.

There are various kinds of compression algorithms [63]. Among them, there is a group of compression algorithms that utilize the inference of context-free grammars (CFGs). Famous examples are LZ78 [78] and Re-pair [52]. They are called grammar compression algorithms. It should be noted that most of them infer CFGs heuristically, because the inference of the smallest CFG is known to be NP hard [42]. Many grammar compression algorithms are known to achieve very high compression ratio, but unfortunately, many of them require a large working space. In addition, some of these algorithms have large latency and cannot be applied to compressed communication. Thus, we need to design online grammar compression algorithms with less working space to achieve compressed communication for IoT applications.

Re-pair [52] is a fast grammar compression algorithm, but it requires $O(N)$ working space, where N is the input size. Masaki et al. [50] reduced the working space to $O(n)$, where n is the grammar size.

FOLCA [49] is the first $O(\log n \log^* n)$-approximable online grammar compression algorithm with succinct working space based on edit-sensitive parsing (ESP). It uses only $n \lg(n+\sigma) + 2n + o(n)$ bits of working space, while a naive implementation requires $2n \log(n+\sigma)$ bits, where n is the grammar size and σ is the alphabet size. They achieved it by improving the compression algorithm called LCA [48]. More recently, Takabatake et al. improved it by proposing SOLCA [72], where the same approximability was achieved with optimal $O(N \log \log n)$ computation time for an input of size N, using $n \log(n+\sigma) + o(n \log(n+\sigma))$ working space.

Research on implementation of online grammar compression algorithms on small-sized hardware is ongoing. Yamagiwa et al. implemented FPGA-based lossless compression hardware [47, 76]. They utilized a compact-memory grammar compression algorithm called LCA-DLT, which is another variation of LCA [48].

There are also open problems. One problem is on the existence of online succinct self-indices, which could help to abuse detection on compressed communications. Another problem is on the development of compressed broadcasting/uploading algorithms on various network structure settings.

References

1. Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, *18*(6), 333–340.
2. Asharov, G., Komargodski, I., Lin, W. K., Nayak, K., & Shi, E. (2018). Optorama: Optimal oblivious ram. In *Cryptology ePrint Archive, Report 2018/892*. https://eprint.iacr.org/2018/892.
3. Baker, B.S. (1995). On finding duplication and near-duplication in large software systems. In *Reverse engineering, proceedings of 2nd working conference on* (pp. 86–95). IEEE.
4. Belazzougui, D., Gagie, T., Mäkinen, V., & Previtali, M. (2016). Fully dynamic de Bruijn graphs. In *International symposium on string processing and information retrieval* (pp. 145–152). Springer.
5. Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., & Puglisi, S.J. (2016). Bidirectional variable-order de Bruijn graphs. In *Latin American Symposium on Theoretical Informatics* (pp. 164–178). Springer.

6. Benoit, D., Demaine, E. D., Munro, J. I., Raman, R., Raman, V., & Rao, S. S. (2005). Representing trees of higher degree. *Algorithmica*, *43*(4), 275–292.
7. Birol, I., Raymond, A., Jackman, S. D., Pleasance, S., Coope, R., Taylor, G. A., et al. (2013). Assembling the 20 gb white spruce (picea glauca) genome from whole-genome shotgun sequencing data. *Bioinformatics*, *29*(12), 1492–1497.
8. Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422–426.
9. Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., & Sadakane, K. (2015). Variable-order de Bruijn graphs. In *2015 data compression conference* (pp. 383–392). IEEE.
10. Bowe, A., Onodera, T., Sadakane, K., & Shibuya, T. (2012). Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics* (pp. 225–235). Springer.
11. Boyle, E., & Naor, M. (2016). Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM conference on innovations in theoretical computer science, ITCS '16* (pp. 357–368).
12. de Bruijn, N. G. (1946). A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, *49*(49), 758–764.
13. Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto.
14. Cash, D., Grubbs, P., Perry, J., & Ristenpart, T. (2015). Leakage-abuse attacks against searchable encryption. In *Proceedings of SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 668–669).
15. Chang, Z., Xie, D., & Li, F. (2016). Oblivious ram: A dissection and experimental evaluation. *Proceedings of the VLDB Endowment*, *9*, 1113–1124.
16. Chikhi, R., Limasset, A., Jackman, S., Simpson, J.T., & Medvedev, P. (2014). On the representation of de Bruijn graphs. In *International conference on research in computational molecular biology* (pp. 35–55). Springer.
17. Chikhi, R., & Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, *8*(1), 22.
18. Conway, T. C., & Bromage, A. J. (2011). Succinct data structures for assembling large genomes. *Bioinformatics*, *27*(4), 479–486.
19. Davis-Dusenbery, B. (2017). Precision medicine research in the million-genome era: Gaining the most from research with multi-omic data in the million-genome era. *Genetic Engineering & Biotechnology News*, *37*(2), 26–27.
20. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., & Wichs, D. (2016). Onion oram: A constant bandwidth blowup oblivious ram. In *Proceedings of the 13th international conference on theory of cryptography conference* (pp. 145–174). Springer.
21. El-Metwally, S., Hamza, T., Zakaria, M., & Helmy, M. (2013). Next-generation sequence assembly: Four stages of data processing and computational challenges. *PLoS Computational biology*, *9*(12), e1003345.
22. Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *Foundations of computer science. Proceedings, 38th annual symposium on* (pp. 137–143). IEEE.
23. Ferragina, P., Luccio, F., Manzini, G., & Muthukrishnan, S. (2009). Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, *57*(1), 4.
24. Ferragina, P., & Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of computer science. Proceedings, 41st annual symposium on* (pp. 390–398). IEEE.
25. Gagie, T., Manzini, G., & Sirén, J. (2017). Wheeler graphs: A framework for bwt-based data structures. *Theoretical Computer Science*, *698*, 67–78.
26. Ganguly, A., Hon, W.K., Sadakane, K., Shah, R., Thankachan, S.V., & Yang, Y. (2016). Space-efficient dictionaries for parameterized and order-preserving pattern matching. In *LIPIcs-Leibniz International Proceedings in Informatics* (vol. 54). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
27. Gargini, P. A. (2017). How to successfully overcome inflection points, or long live Moore's law. *Computing in Science & Engineering*, *19*(2), 51–62.
28. Goldreich, O. (1987). Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of symposium on theory of computing (STOC)* (pp. 182–194).
29. Goldreich, O., & Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, *43*(3), 431–473.
30. Howe, A.C., Jansson, J.K., Malfatti, S.A., Tringe, S.G., Tiedje, J.M., & Brown, C.T. (2014). Tackling soil diversity with the assembly of large, complex metagenomes. In *Proceedings of the National Academy of Sciences* (p. 201402564).

31. Islam, M., Kuzu, M., & Kantarcioglu, M. (2012). Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of network and distributed system security symposium (NDSS)*.
32. Jackman, S. D., Vandervalk, B. P., Mohamadi, H., Chu, J., Yeo, S., Hammond, S. A., et al. (2017). Abyss 2.0: Resource-efficient assembly of large genomes using a bloom filter. *Genome Research*, *27*, gr-214346.
33. Jacobson, G. (1989). Space-efficient static trees and graphs. In *Foundations of Computer Science, 30th Annual Symposium on* (pp. 549–554). IEEE.
34. Kärkkäinen, J., & Sanders, P. (2003). Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming* (pp. 943–955). Springer.
35. Knuth, D. E., Morris, J. H, Jr., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, *6*(2), 323–350.
36. Ko, P., & Aluru, S. (2003). Space efficient linear time construction of suffix arrays. In *Annual Symposium on Combinatorial Pattern Matching* (pp. 200–210). Springer.
37. Kosaraju, S.R. (1989). Efficient tree pattern matching. In *Foundations of Computer Science, 30th Annual Symposium on* (pp. 178–183). IEEE.
38. Kushilevitz, E., Lu, S., & Ostrovsky, R. (2012). On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the 23rd Annual ACM-SIAM symposium on Discrete Algorithms (SODA)* (pp. 143–156). Society for Industrial and Applied Mathematics.
39. Labeit, J., Shun, J., & Blelloch, G. E. (2017). Parallel lightweight wavelet tree, suffix array and fm-index construction. *Journal of Discrete Algorithms*, *43*, 2–17.
40. Langmead, B., Trapnell, C., Pop, M., & Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, *10*(3), R25.
41. Larsen, K., & Nielsen, J. (2018). Yes, there is an oblivious ram lower bound!. *Advances in Cryptology CRYPTO*, *10992*, 523–542.
42. Lehman, E., & Shelat, A. (2002). Approximation algorithms for grammar-based compression. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* (pp. 205–212). Society for Industrial and Applied Mathematics.
43. Li, D., Liu, C. M., Luo, R., Sadakane, K., & Lam, T. W. (2015). Megahit: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, *31*(10), 1674–1676.
44. Li, D., Luo, R., Liu, C. M., Leung, C. M., Ting, H. F., Sadakane, K., et al. (2016). Megahit v1. 0: A fast and scalable metagenome assembler driven by advanced methodologies and community practices. *Methods*, *102*, 3–11.
45. Li, H., & Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, *25*(14), 1754–1760.
46. Manber, U., & Myers, G. (1993). Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, *22*(5), 935–948.
47. Marumo, K., Yamagiwa, S., Morita, R., & Sakamoto, H. (2016). Lazy management for frequency table on hardware-based stream lossless data compression. *Information*, *7*(4), 63.
48. Maruyama, S., Sakamoto, H., & Takeda, M. (2012). An online algorithm for lightweight grammar-based compression. *Algorithms*, *5*(2), 214–235.
49. Maruyama, S., & Tabei, Y. (2014). Fully online grammar compression in constant space. In *Data Compression Conference (DCC)* (pp. 173–182). IEEE.
50. Masaki, T., & Kida, T. (2016). Online grammar transformation based on re-pair algorithm. In *2016 Data Compression Conference (DCC)* (pp. 349–358). IEEE.
51. McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, *23*(2), 262–272.
52. Moffat, N.L.A., & Larsson, J. (2000). Offline dictionary-based compression. In *Data Compression Conference* (pp. 296–305).
53. Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8), 114–117.
54. Muggli, M. D., Bowe, A., Noyes, N. R., Morley, P. S., Belk, K. E., Raymond, R., et al. (2017). Succinct colored de Bruijn graphs. *Bioinformatics*, *33*(20), 3181–3187.
55. Nong, G., Zhang, S., & Chan, W.H. (2009). Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09* (pp. 193–202). IEEE.
56. Olson, N. D., Treangen, T. J., Hill, C. M., Cepeda-Espinoza, V., Ghurye, J., Koren, S., & Pop, M. (2017). Metagenomic assembly through the lens of validation: Recent advances in assessing and

improving the quality of genomes assembled from metagenomes. *Briefings in Bioinformatics*. https://doi.org/10.1093/bib/bbx098
57. Onodera, T., & Shibuya, T. (2018). Succinct Oblivious RAM. In 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018), *Leibniz International Proceedings in Informatics (LIPIcs), 96*, 52.1–52.16.
58. Patel, S., Persiano, G., Raykova, M., & Yeo, K. (2018). Panorama: Oblivious ram with logarithmic overhead. In *Proceedings of 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)* (pp. 871–882).
59. Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J. M., & Brown, C. T. (2012). Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, *109*(33), 13272–13277.
60. Pevzner, P. A., Tang, H., & Waterman, M. S. (2001). An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, *98*(17), 9748–9753.
61. Policriti, A., & Prezza, N. (2014). Hashing and indexing: Succinct datastructures and smoothed analysis. In *International Symposium on Algorithms and Computation* (pp. 157–168). Springer.
62. Sadakane, K., & Navarro, G. (2010). Fully-functional succinct trees. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms* (pp. 134–149). Society for Industrial and Applied Mathematics
63. Salomon, D., & Motta, G. (2010). *Handbook of data compression*. New York: Springer.
64. Shi, E., Chan, T. H. H., Stefanov, E., & Li, M. (2011). Oblivious ram with $o((\log n)^3)$ worst-case cost (pp. 197–214). Berlin: Springer.
65. Shibuya, T. (2003). Constructing the suffix tree of a tree with a large alphabet. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, *86*(5), 1061–1066.
66. Shibuya, T. (2004). Generalization of a suffix tree for rna structural pattern matching. *Algorithmica*, *39*(1), 1–19.
67. Simpson, J. T., & Pop, M. (2015). The theory and practice of genome sequence assembly. *Annual review of genomics and human genetics*, *16*, 153–172.
68. Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., & Birol, I. (2009). Abyss: A parallel assembler for short read sequence data. *Genome Research*, *19*(6), 1117–1123.
69. Sirén, J. (2017). Indexing variation graphs. In *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)* (pp. 13–27). SIAM.
70. Stefanov, E., Shi, E., & Song, D. (2012). Towards practical oblivious ram. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*.
71. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., & Devadas, S. (2013). Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 299–310). ACM.
72. Takabatake, Y., Sakamoto, H., et al. (2017). A space-optimal grammar compression. In *LIPIcs-Leibniz International Proceedings in Informatics* (vol. 87). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
73. Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, *14*(3), 249–260.
74. Wang, X., Chan, H., & Shi, E. (2015). Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 850–861). ACM.
75. Weiner, P. (1973). Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on* (pp. 1–11). IEEE.
76. Yamagiwa, S., Marumo, K., & Sakamoto, H. (2015). Stream-based lossless data compression hardware using adaptive frequency table management. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware* (pp. 133–146). Springer.
77. Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., & Douglas, W.Y. (2012). Exploiting sparseness in de novo genome assembly. In *BMC bioinformatics* (vol. 13, p. S1). BioMed Central.
78. Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, *24*(5), 530–536.