A tale of optimizing the space taken by de Bruijn graphs

Rayan Chikhi

Institut Pasteur, CNRS, Paris, France
rayan.chikhi@pasteur.fr

Abstract. In the last decade in bioinformatics, many computational works have studied a graph data structure used to represent genomic data, the de Bruijn graph. It is closely tied to the problem of genome assembly, i.e. the reconstruction of an organism's chromosomes using a large collection of overlapping short fragments. We start by highlighting this connection, noting that assembling genomes is a computationnally intensive task, and then focus our attention on the reduction of the space taken by de Bruijn graph data structures. This extended abstract is a retrospective centered around my own previous work in this area. It complements a recent review [10] by providing a less technical and more introductory exposition of a selection of concepts.

Keywords: Bioinformatics · data structures · de Bruijn graphs.

1 Context

Let us travel back in time in 2008, when the problem of reconstructing genomes using DNA sequencing, termed *de novo* genome assembly, had a rebirth as an active area of research within many computational research groups. It was a somewhat "fresh" problem at the time: many genomes were already assembled, e.g. the Human Genome Project completed at the beginning of the 2000s, yet performing the assembly of any organism was just starting to be within reach for most biological labs. The vast majority of organisms did not have their genomes assembled (and as of today: they still do not). So the challenge was to create software that any individual lab could use, not just large organizations. The main type of data at the time were *short reads*, i.e. fragments of around 100 nucleotides, meaning that only a tiny fraction of a genome could be read contiguously at a time. (Genomes of viruses are in the order of thousands of nucleotides, but for most other organisms they range from millions to billions.) By repeatedly sequencing fragments from random locations, reads would significantly overlap which makes genome reconstruction possible. Short reads were produced mainly from the company Illumina, still a market leader on DNA sequencing today; some of the previous technologies (e.g. 454) were on their way out.

The EULER-SR assembler was one of the first specialized genome assembly software for short reads, and it came out in 2008. It achieved an assembly of

a bacterium (*E. coli*) in 199 pieces [9]. This means that the genome was near-completely reconstructed, yet in a fragmented way to due ambiguities. This may seem unremarkable by current standards, as nowadays we can reconstruct nearly all bacteria in a single piece per chromosome. Yet the task was fundamentally hindered by the length of the short reads. Still, at the time it was clear that the next frontier would be to assemble larger genomes, e.g. animals or plants, even if the final assembly would still be largely fragmented.

The widely-used Velvet [43], ABySS [41] and SOAPdenovo [24] assemblers appared in the following two years. And indeed, the last two were able to assemble a human genome using a cluster or a single large-memory machine. These assemblers were all based on a certain representation of the input data, the de Bruijn graph, that we will explain in the next section. These graphs come from mathematics and had not yet been widely used outside of some networking applications. This was before the era of more advanced assemblers (IDBA/SPAdes [37, 3]); early assemblers only constructed a single graph, as opposed to iterating over multiple graphs with different parameters.

Even so, the construction of a large de Bruijn graph was the most computation-intensive step of genome assembly at the time. This should come as no surprise, as 1) this was the first period in history when one had to construct large de Bruijn graphs in any domain; there existed no previous litterature describing how to do it efficiently, and no software library. 2) It did not matter so much if construction was slow or memory-intensive, as long as some large-memory machine managed to run it. 3) The volume of input data was really large by historical standards: in the order of a hundred of gigabytes in compressed form. Yet, as genome assembly later became a routine task, along with the advent of huge instances such as metagenomics (the analysis of multiple genomes at once), the efficient construction de Bruijn graphs naturally became a critical aspect of genome assembly. It also turns out that de Bruijn graphs would be useful for other biological sequence analysis tasks, such as the sequencing of RNA [35], the compression of genomic data [20], and the detection or representation of variations across a single or multiple genomes [16].

The goal of this extended abstract is to retrace some of the steps that the community and I took towards achieving space-efficient representations of de Bruijn graphs, starting from the initial attempts in the first assemblers, making a detour through theoretical lower bounds, and finishing with current advances and some perspectives.

2 Problem formulation

Let us introduce some of the concepts. A DNA sequence is seen as a string over four possible characters (A,C,T,G). A **k-mer** is a portion of DNA sequence of length k, e.g. ACT is a 3-mer. The **de Bruijn graph** is a directed graph where nodes are k-mers, and edges are the exact suffix-prefix overlaps of length $k-1$ between two nodes; e.g. ACT→CTA or AAA→AAT, but ACT and AAA are not connected by an edge. See Figure 1 for another example. Note that in practice, k

is typically greater than 20. A de Bruijn graph is constructed by inserting all the possible k-mers present in an input dataset. If the same k-mer is seen multiple times, all of its occurrences are associated to the same node.

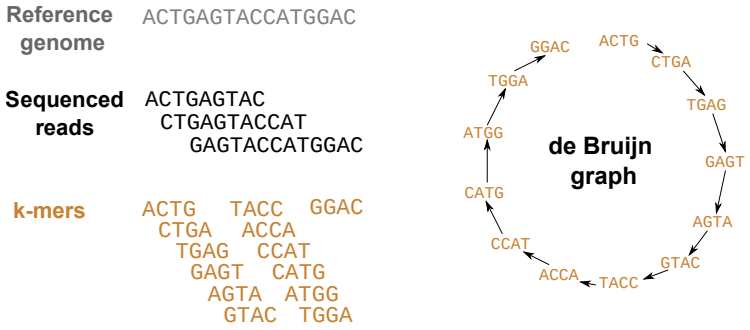

Fig. 1. Left panel: example of a toy reference genome sequence, a set of 3 sequenced reads, and the corresponding 4-mers extracted from the reads. Right panel: the de Bruijn graph constructed these reads with $k = 4$ and drawn using a circular layout.

The scientific question we will be interested in can be informally stated as follows: given a set of nodes of the de Bruijn graph, stored on disk, construct an in-memory representation[1] that supports a reasonable subset of standard graph operations, e.g. determine all the neighbors of a node, determine if some putative node is present or absent, etc. The representation should take as little memory as possible, and answer queries reasonably fast, although as we will see next, the main limiting factor here is typically not the query time but the representation size.

Note that prior to circa 2012, the problem as stated above was not recognized as its own area of investigation within bioinformatics nor computer science. Arguably it became one when several data structures were published as stand-alone articles [14, 13, 5].

3 Caveats

We will focus here on only a selection of major milestones, where space usage was reduced, ignoring other features such as query times. The presentation will also sacrifice some technical accuracy in favor of accessibility. For a more complete and technical exposition, please refer to this review [10].

Note that genome assembly cannot be reduced to the representation of the de Bruijn graph. In fact, many older tools even used different paradigms [32].

[1] Such a representation is also often called a **data structure**, and the abstract model that encompasses all the data structures supporting the same operations is called an *abstract data type*.

Among those which do use a de Bruijn graph, they implement many steps before (e.g. error correction) and after (e.g. graph cleaning) the construction of the graph that crucially affect results quality. However, for the sake of keeping the story coherent, we will set aside this broader environment to focus solely on the efficiency of graph representation.

4 The early days

The early assembly programs from the 2008–2010 era did not particularly aim to optimize the space usage of de Bruijn graphs. Therefore, their memory usage may be seen as wasteful by current standards, yet they laid the bases for future progress.

The EULER-SR assembler reported building the graph using what they describe as "an efficient hashing structure" which was then transformed into a sorted list of vertices, queried using binary search. Notably, k-mers were represented explicitly as strings.[2]

Similarly the Velvet assembler, published the same year, used a hash table to record for each k-mer "the ID of the first read encountered containing that k-mer and the position of its occurrence within that read". It is natural to want to keep track of where each k-mer is coming from, however as we will see next, storing this information in the graph is prohibitively expensive. The authors note: "The main bottleneck, in terms of time and memory, is the graph construction. The initial graph of the *Streptococcus* reads needs 2.0 [gigabytes] of RAM." Given that the *Streptococcus* genome is 2 million nucleotides in length, and under the assumption that there were roughly 10x more erroneous k-mers than correct ones, we infer that the de Bruijn graph representation of Velvet required in the order of 100 bytes per k-mer.

The SOAPdenovo assembler followed the Velvet assembler strategy, except that its authors realized that one could achieve nearly identical (or even better) results despite discarding a lot of space-intensive information in the hash table (i.e. read locations and paired-end information). Its graph representation required 120 GB of memory for storing 5 billion nodes of a human genome [32], i.e. around 24 bytes per k-mer. This prowess demonstrated that the quality of genome assembly was not sacrified when trimming down the de Bruijn graph data structure. There existed some minimal set of supported operations that would make a de Bruijn graph fit for purpose, although this set was not described at the time. As long as a data structure would support all these features, then computer scientists would be free to optimize it as much as they could.

The Meraculous assembler, published in 2011, took a radically different approach by storing the de Bruijn graph using collision-free hashing. Its representation only supports the lookup of the next nucleotide following a k-mer (i.e. the out-neighbor of a node), where k-mers having multiple out-neighbors were

[2] The total space usage of the graph was reported to be $O(L) * (v + k)$ bytes, where L is the genome size, k is the k-mer length and v is the memory allocated per vertex, reported to be 40 bytes.

previously discarded during a pre-processing step. As in other assemblers, there are further steps taken to attempt to "fill the gaps" between the discarded k-mers and to orient the assembled fragments, yet these are outside our current scope. The Meraculous de Bruijn graph structure does not support enumeration of vertices. Despite the apparent minimality in terms of supported operations, it appeared to be sufficient for enabling genome assembly. While this technique was not further re-used by other assembly tools, we revisited it 6 years later to develop the general-purpose *minimal perfect hashing* library BBHash [25].

5 The birth of a line of research

The years 2011–2012 saw a remarkable amount of independent contributions proposing new ways to represent the de Bruijn graph in a space-efficient manner. In retrospect, the field was ripe for such contributions as there was an important problem to be solved (genome assembly of human genomes was taking a prohibitive amount of memory), which was well-defined computationally[3], and there were no previous "clever" solutions apart from using off-the-shelf data structures.

To my knowledge, the first article on this topic was published in 2011 by T. Conway and A. Bromage [14]. They describe an encoding of the de Bruijn graph using an existing state-of-the-art efficient encoding of bit arrays. Furthermore, they also show that their representation is 'optimal', in the following sense: information theory dictates that any other exact de Bruijn graph representation will have to use as many bits per k-mer in the worst case. The key words of the previouse sentence are "exact" and "worst case", and we will revisit this statement later in this document. But for now, it is sufficient to note that to this date, the Conway-Bromage data structure is provably optimal. Then, does this mark the end of the line of research on the representation of de Bruijn graphs?

Not quite, despite the lower bound argument being convincing. We will briefly expose it here. Observe that to represent a de Bruijn graph, one only needs to represent its vertices[4]. The edges are indeed implicit in the representation, as one could determine all the neighbors of a certain k-mer by querying for the presence of all the potential k-mers shifted to the left or to the right.

Then, a bijection is established between the set of all possible sets of n vertices, and the set of all possible binary vectors having n ones and $4^k - n$ zeros. The bijection is actually rather straightforward: each k-mer is directly encoded as an integer in base 4 (see Figure 2 middle panel), and a bit vector has a **1** at position i if and only if i is the encoding of a k-mer that belongs to the set

[3] At least implicitly, as to my knowledge, it has not been explicitly formulated as an open question in an article.

[4] We omit a technicality here that will only be of interest to specialists. We only consider node-centric de Bruijn graphs. For edge-centric de Bruijn graphs, the argument stated in this paragraph does not apply. Yet, edge-centric graphs are tightly related to node-centric ones and in practice, using one definition or the other does not matter.

of graph vertices. Since the number of possible bit vectors is classically known, one deduces that to represent a de Bruijn graph for a certain parameter k having n vertices, one must uses in the worst case as many bits as the logarithm of the number of possible bit vectors of size 4^k that have n ones.

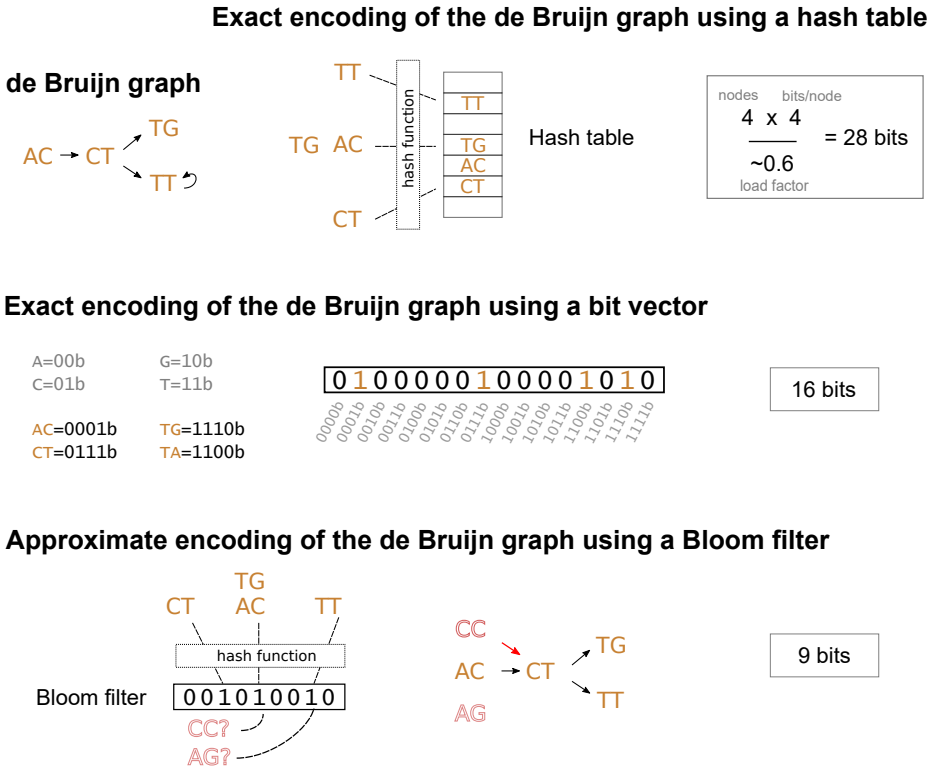

Fig. 2. Example of a de Bruijn graph (top left panel) and three possible encodings. Top right panel: hash table, each node is inserted at a position given by a random hash function. The collision between TG and AC is resolved using linear probing, i.e. by inserting AC at the next free slot in the table. The load factor is number of occupied cells over total cells. Middle panel: bit vector, storing each node converted into an integer using the classical binary encoding of characters `A,C,G,T=00b,01b,10b,11b`, where `b` indicates that the number is written in binary. Bottom panel: Bloom filter, where each node is inserted at a position given by a random hash function. Two false positives nodes (CC, AG) are shown in red. They arise because the hash function causes collisions between any possibly existing node and true nodes.

6 Beating the lower bound (by inexactness)

As it turns out, this lower bound did not discourage researchers from proposing data structures that exhibited even lower space usages in practice than those dictated by the bound. One such data structure is the encoding of a de Bruijn graph using a Bloom filter by Pell *et al* [36] (Fig 2). By inserting all the vertices of the graph inside a probabilitic membership data structure (here, the Bloom filter), it is possible to represent a set of k-mers approximatively. The trade-off is then that the graph is not exactly represented, yet the space usage is an order of magnitude lower than the one dictated by the Conway-Bromage lower bound: around 4 bits per k-mer. Pell *et al* showed that despite having many false positive nodes resulting from the approximate representation, it was still possible to perform useful analysis on the graph – not quite genome assembly, but another related task (read partitioning).

Following this, G. Rizk and I proposed to extend this representation to make it exact within a certain setting, and perform genome assembly [13]. Due to the lower bound, any attempt at removing all the false positives of the Bloom filter would result in a data structure that would necessarily be at least as large as the one from Conway-Bromage. The key insight was to realize that only a fraction of the false positives of the Bloom filter mattered: those which were neighbors of a true positive vertex. By explicitly storing them in a "blacklist" hash table, our data structure managed to represent a graph in typically ≈ 13 bits per k-mer[5], and the neighbor query operation would be exact. This "beats" the Conway-Bromage bound by a factor of roughly 2x. The caveat is that the representation is not exact everywhere, but as long as a user traverses the graph from a true positive vertex, then the representation would act identically to the exact one. We implemented this data structure inside a genome assembler, Minia [13].

Another instance of an inexact de Bruijn graph representation is the *sparse de Bruijn graph* [42], which is a de Bruijn graph that skips g intermediate k-mers, providing roughly a $1/g$ space saving, where g was set to 16. Finally, along the same line of thought the A-Bruijn graph formalism [26] selects an arbitrary set of strings, and creates an edge when two strings appear consecutively in at least one read. This concepts generalizes de Bruijn graphs; yet A-Bruijn graphs may contain one or several orders of magnitude less nodes than de Bruijn graphs. There are some potentially interesting parallels between A-Bruijn graphs and sparse de Bruijn graphs, yet to the best of my knowledge they have not been explored.

7 Beating the lower bound (by instance specificity)

Independently of Minia, and presented at the same session of the WABI conference in 2012, the BOSS data structure proposes a completely different yet exact de Bruijn graph representation [5]. It uses a variant of the Burrows-Wheeler

[5] Which would later be improved by Sahlikov & Kucherov to ≈ 8.5 bits per k-mer, using cascading Bloom filters [40].

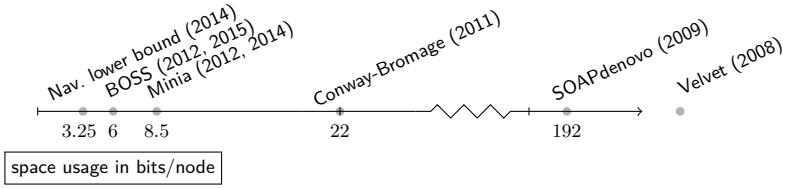

Fig. 3. Space taken by various representations of de Bruijn graphs, in bits per node. "Conway-Bromage" is both the Conway-Bromage exact lower bound and its matching upper bound. "Nav. lower bound" is the navigational lower bound for general de Bruijn graphs from [11]. BOSS is the flavor of [23].

transform specifically tailored for k-mers. A complete description of the BOSS structure, or even the Burrows-Wheeler transform, would be beyond the technical level of this document, and can be found in [1]. Intuitively, the Burrows-Wheeler transform [7] is a permutation of the characters of a string that facilitates substring search and compression. BOSS extends this concept by storing a permutation of the last characters of each k-mer[6] together with a bit array. The result is a data structure that supports efficient membership queries and neighborhood traversal of the graph, all in around 6 bits per k-mer in practice. While in the first few years the construction of this structure was relatively impractical, recent improvements lifted those limitations, allowing to process even terabases of input data [22].

Taking a step back, BOSS is an exact representation that appears to somehow beat the Conway-Bromage lower bound. How is this even possible? While this aspect was not discussed in nearly all of the publications related to BOSS, it turns out that BOSS has been mainly applied to k-mer sets that have a so-called *spectrum-like property* [10], i.e. where all the k-mers originate from some underlying long strings. Should BOSS be applied to an arbitrary set of k-mers, its space usage would mechanically be raised to match or exceed the Conway-Bromage lower bound; yet, this fact has to my knowledge never been properly tested in practice.

Regardless, the spectrum-like property and the effectiveness of BOSS are important insights: a data structure may do better than the worst-case lower bound while still remaining exact, when it is restricted to a certain class of inputs that matter in practice. Then, a natural next question arises: what would be a more realistic lower bound for representing 'practical' de Bruijn graphs, i.e. those having spectrum-like property?

Several collaborators and I addressed this question in [11], where we formulated several concepts. First, we defined a *navigational* data structure as one that enables navigation in the graph but does not necessarily support membership queries. We showed that navigational data structures for general de Bruijn

[6] along with some additional artificial k-mers to "pad" those which do not have a large enough neighborhood.

graphs require at least 3.24 bits per k-mer in the worst case. When restricted to the family of linear de Bruijn graphs (i.e. graphs where all nodes have a single in-neighbor and/or single out-neighbor), then a lower bound for navigational data structures is 2 bits per k-mer. This last lower bound is tight, as representing the linear de Bruijn graph using the Burrows-Wheeler transform (or its optimized flavor, FM-index [17]) yields also a data structure that is asymptotically close to 2 bits per k-mer. In [11], we also proposed a new data structure for de Bruijn graphs having the spectrum-like property, using the Burrows-Wheeler transform, and showed that it takes $2 + (k + 2)c/n$ bits per k-mer, where c is essentially the maximal number of k-mer-disjoint strings the k-mers could have been generated from[7].

Lastly, one may also wonder how a de Bruijn graph could be further compressed, e.g. to be stored on disk. Supposedly such a compressed representation would be even smaller than the previously mentioned data structures. The trade-off is the inability to perform fast queries. Two independent works [6, 38], one on *simplitigs* and the other on *spectrum-preserving string sets* which I was associated with, proposed to store non-overlapping paths of the compacted de Bruijn graph (defined later) as sequences, and store them in compressed form on disk. Despite the representation apparently storing an incorrect representation of the graph, due to paths being constructed by choosing edges arbitrarily, one may observe that the original graph can be reconstructed losslessly from its path representation. Such a disk representation achieved a space very close to the 2 bits per k-mer lower bound: 4.1 bits per k-mer for a whole human genome read dataset, and 2.7 bits per k-mer for a human metagenome.

8 Construction algorithms

An apparté will be made in this section, where we will briefly mention the data structure construction algorithms. One typical pre-processing step commonly done prior to creating a de Bruijn graph data structure is k-mer counting. This step takes the input sequencing data and yields the set of distinct k-mers present in the input along with their abundances. It essentially constructs the nodes of the de Bruijn graph.

During the development of Minia, we had ran into an issue. The graph representation was so succinct that other steps of the genome assembly pipeline acted as bottlenecks, including k-mer counting. At the time, the most efficient k-mer counter was Jellyfish [28], which used a custom thread-safe hash table optimized specifically to store k-mers. Yet, Jellyfish would have used much more memory than Minia. We therefore set out to design a low-memory k-mer counting tool that would use the disk to alleviate memory usage (DSK [39]). This strategy was also used by other popular k-mer counting tools, e.g. KMC [15].

The problem of k-mer counting is fascinating in its simplicity but also difficult to engineer correctly, given that large volumes of input sequences need to be

[7] For specialists, c is the number of unitigs.

processed with high CPU utilization, low memory usage, and bandwidth-limited disk accesses. A relatively current review is [27]. After k-mer counting, nearly all of the data structures presented above have their own, customized construction algorithms. As such, there does not exist an 'universal' construction algorithm for de Bruijn graph that would then be slightly adapted to derive a particular data structure.

However, several recent data structures (the one presented in [11], Pufferfish [2], BLight [30]) require as input a common object: the **compacted de Bruijn graph**. It is obtained from a classical de Bruijn graph by transforming each non-branching path into a single node, similarly to suffix tries are transformed into suffix trees by collapsing paths of vertices having one child. However, this is a circular situation: in order to construct an efficient representation of the classical de Bruijn graph, one must have already constructed a compacted de Bruijn graph, which itself is obtained from the classical de Bruijn graph. In order to break this circularity, My colleagues and I proposed an efficient construction algorithm for the compacted de Bruijn graph [11], which uses a constant amount of memory. It was further extended to make use of multiple threads efficiently [12].

9 Current state of the art

Since the influx of de Bruijn graph data structures in 2012, several more have been published in the recent years. As it turns out, many of them are based on minimal perfect hashing. It is a variation of a hash table which does not store its keys, yet still manages to resolves collisions. Minimal perfect hashing is unable to confirm if an arbitrary key is present or absent in the structure, however for any key that was inserted during its construction, it returns an exact answer. This makes the structure highly space-efficient. Along with colleagues, we proposed a fast parallel C++ library for constructing minimal perfect hashes (BBHash [25]) which has been engineered to scale significantly better than other existing implementations at the time.

Among current de Bruijn graph data structures, I will briefly highlight Pufferfish [2] and Blight [30] which are both based on the compacted de Bruijn graph, and queries are supported by an additional minimal perfect hashing structure that quickly locates positions within nodes of the compacted graph. The Bloom Filter trie [19] and Bifrost [18] structures both use Bloom filters in addition with other auxiliary data structures to keep the representation exact, yet even brushing their algorithmic details would be too technical for this survey. Counting quotient filters [34] improve upon Bloom filters by also storing the number of occurrences of each node in the input data. Belazzougui *et al.* proposed a navigational data structure based on minimal perfect hashing, with a clever addition of a tree data structure to restore membership queries. For more details on all of these data structures, see [10].

In a way, one might acknowledge that "the dust has settled" in the landscape of de Bruijn graph data structures. The bioinformatics algorithms community

has attempted for several years to come up with solutions that combine low space usage, fast query speed and a reasonable set of features. The outcome is a set of current data structures that achieve reasonable trade-offs, with space close to the known lower bounds. As a result, de Bruijn graphs are no longer a bottleneck in genome assembly, partly also due to decreasing RAM costs (Fig 4).

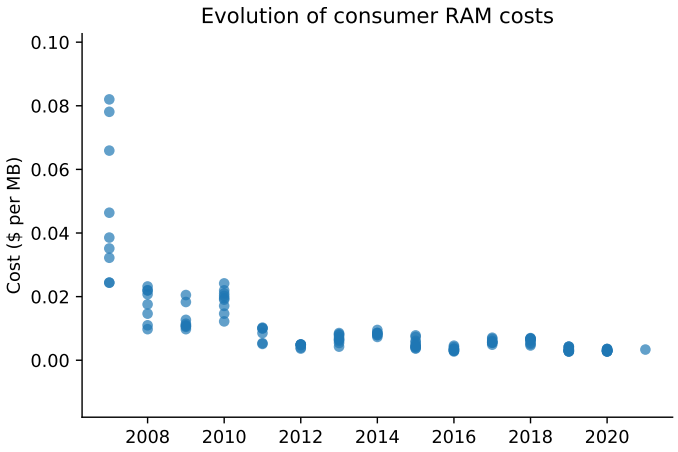

Fig. 4. Consumer RAM costs from the 2007–2021 period. Each dot is a retail DIMM product sold on the current year. Source: `https://jcmit.net/memoryprice.htm`

Then, is this the end of this line of research? Not quite, as the natural next frontier is the representation of multiple genomes within a generalization of the de Bruijn graph.

10 Colored de Bruijn graphs

As a coincidence of dates (or perhaps not), 2012 was not only the year where many seminal data structures for de Bruijn graphs were proposed, but also the year when the term *colored de Bruijn graph* was coined (in [21]), which will pave the way to the next type of contributions that we will mention here. Colored de Bruijn graphs generalize de Bruijn graphs to multiple samples. When faced with multiple samples, a classical de Bruijn graph would bundle them together and consider the union of all samples as a single "mega-sample". Colored de Bruijn graph also do that, but they add additional information associated to the nodes so that one can tell the origin of each node across samples. Naturally, speaking in terms of lower bounds, storing such a graph for multiple samples should require strictly more space than storing the graph of any subset of samples.

Several data structures have been proposed to store colored de Bruijn graph, the first of which was based on an efficient hash table [21], then later using the

Burrows-Wheeler transform [33]. More recently my colleagues and I proposed the REINDEER structure, based on compacted de Bruijn graphs and minimal perfect hashing [29], with the distinctive feature of not only storing the presence/absence of nodes, but also the approximate frequency of each node within each sample.

To the best of my knowledge, there has been no attempt made at formulating space lower bounds for colored de Bruijn graphs. One may obtain one through an immediate application of existing lower bounds to the union graph disregarding color information. However, this would be a loose bound as much of the difficulty of storing colored graphs lies in the color information.

11 Wrap-up and open questions

As we reviewed above, many data structures have been proposed to store de Bruijn graphs, achieving several order of magnitudes improvement in space usage compared to using off-the-shelf data structures for graph storage. From this perspective, the theoretical study of data structures along with their practical implementations has been successful at providing performance gains for widely-used software tools (e.g. [3, 23]). Looking back, the improvements have mainly be due to two realizations. 1) Data structure exactness can be sacrified yet still provide exact results in a certain frame of operations. 2) The theoretical worst-case analysis of data structures inadequately applies to practical instances. The latter realization is the topic of an upcoming article from Medvedev [31], critically reflecting on the analysis of bioinformatics algorithms more broadly.

Several topics were not covered in this document, to keep it simple. One is double-strandedness, which forces all the data structures mentioned above to consider that a k-mer and its reverse-complement should be the same object; this adds theoretical and especially practical complications, yet does not fundamentally change the exposition of the data structures. An additional one is the use of multiple k values. Nowadays genome assembly tools on short reads typically construct multiple de Bruijn graphs iteratively. This is a somewhat orthogonal matter as presented here, given that each individual graph is represented using one of the techniques above. We note however that some works have attempted to unify multiple graphs into one [4, 8]. Another consideration is how to store the number of times each k-mer is seen in the input. All these considerations are discussed in more details in [10].

We summarize here a few open questions:

1. Can compressed representations e.g. spectrum-preserving string sets be made efficiently queryable? This would lead to even more compressed de Bruijn graphs.
2. What would be a space lower bound for exactly representing a colored de Bruijn graph of n samples, each sample i having D_i distinct k-mers?
3. A matching upper bound of the above.

4. How to efficiently represent not only the presence/absence of a node but also its abundance in colored de Bruijn graphs (improving upon REINDEER [29]).

Acknowledgements

I thank Paul Medvedev for helpful suggestions on this article.

References

1. Bahar Alipanahi, Alan Kuhnle, Simon J Puglisi, Leena Salmela, and Christina Boucher. Succinct Dynamic de Bruijn Graphs. *Bioinformatics*, 2020.
2. Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.
3. Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.
4. C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi, and K. Sadakane. Variable-Order de Bruijn Graphs. In *2015 Data Compression Conference*, pages 383–392, 2015.
5. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.
6. Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *bioRxiv*, 2020.
7. Michael Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Report 124, Digital Systems Research Center, Palo Alto, CA, USA, May 1994.
8. Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Information Processing Letters*, 155:105862, 2020.
9. Mark J Chaisson and Pavel A Pevzner. Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–330, 2008.
10. Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k-long DNA sequences. *arXiv preprint arXiv:1903.12312*, 2019.
11. Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *International conference on Research in computational molecular biology*, pages 35–55. Springer, 2014.
12. Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.
13. Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):1–9, 2013.
14. Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

15. Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. Disk-based k-mer counting on a PC. *BMC bioinformatics*, 14(1):1–12, 2013.
16. Jordan M. Eizenga, Adam M. Novak, Jonas A. Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D. Seaman, Robin Rounthwaite, Jana Ebler, Mikko Rautiainen, Shilpa Garg, Benedict Paten, Tobias Marschall, Jouni Sirén, and Erik Garrison. Pangenome graphs. *Annual Review of Genomics and Human Genetics*, 21(1):139–162, 2020. PMID: 32453966.
17. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.
18. Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome biology*, 21(1):1–20, 2020.
19. Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11(1):1–9, 2016.
20. Guillaume Holley, Roland Wittler, Jens Stoye, and Faraz Hach. Dynamic alignment-free and reference-free read compression. In *International Conference on Research in Computational Molecular Biology*, pages 50–65. Springer, 2017.
21. Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.
22. Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Ratsch, and André Kahles. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *bioRxiv*, 2020.
23. Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015.
24. Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.
25. Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *arXiv preprint arXiv:1702.03154*, 2017.
26. Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.
27. Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018.
28. Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.
29. Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020.
30. Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Blight: Efficient exact associative structure for k-mers. *bioRxiv*, page 546309, 2020.
31. Paul Medvedev. The theoretical analysis of sequencing bioinformatic algorithms. in preparation.
32. Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

33. Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.
34. Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*, pages 775–787, 2017.
35. Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017.
36. Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.
37. Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA–a practical iterative de Bruijn graph de novo assembler. In *Annual international conference on research in computational molecular biology*, pages 426–440. Springer, 2010.
38. Amatur Rahman, Rayan Chikhi, and Paul Medvedev. Disk Compression of k-mer Sets. In *20th International Workshop on Algorithms in Bioinformatics (WABI 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
39. Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.
40. Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology*, 9(1):1–10, 2014.
41. Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.
42. Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and W Yu Douglas. Exploiting sparseness in de novo genome assembly. In *BMC bioinformatics*, volume 13, pages 1–8. BioMed Central, 2012.
43. Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.