Indexing De Bruijn graphs with minimizers

Camille Marchet[1], Maël Kerbiriou[1] and Antoine Limasset[1]
[1]Univ. Lille, CNRS, Inria, UMR 9189 - CRIStAL.

Abstract

Background: The need to associate information to words is shared among a plethora of applications and methods in high throughput sequence analysis, and could be marked as fundamental. A scalability problem is promptly met when indexing billions of k-mers, as exact associative indexes can be memory expensive. To leverage this challenge, recent works take advantage of the k-mer sets properties. They exploit the overlaps shared among k-mers by using a De Bruijn graph as a compact k-mer set to provide lightweight structures
Contribution: We propose a scalable and exact index structure able to associate unique identifiers to indexed k-mers and to reject alien k-mers. The proposed index combines an extremely compact representation along with a high throughput. Moreover, it can be efficiently built from the De Bruijn graph sequences. The efficient index implementation we provide, achieved to index the k-mers from the human genome with 8GB within 30 minutes and was able to scale up to the huge axolotl genome with 63 GB within 10 hours. Furthermore, while being memory efficient, the index allows above a million queries per second on a single CPU in our experiments and its throughput can be raised using multiple cores. Finally, we also present the index ability to practically represent metagenomic and transcriptomic sequencing data to highlight its wide applicative range.
Availability: The index is implemented as a header-only library in C++, is open source under AGPL3 license and available at `https://github.com/Malfoy/Blight`. It was designed as a user-friendly library and comes along with sample code usage.

1 Introduction

Tremendous, ever growing amounts of DNA and RNA reads are made available through high-throughput sequencing methods. Single RNA, DNA or metagenome and metatranscriptome samples can contain up to billion reads. As an example, the NIH Sequence Read Archive (SRA) [1] gathers petabases of sequences. Working on such collections of samples is an important challenge for indexation scheme. Even when assembling raw data to genome contigs, indexing very large genomes (for instance *Pinus taeda* [2] with 20Gbp or *Ambystoma mexicanum* [3] with 32 Gbp) or metagenomes remain a serious difficulty. In order to deal with these magnitudes, efforts have been put into designing data structures that perform the generic task of associating pieces of information to words (k-mers) from studied sequences. This fundamental block is a corner stone for a large spectrum of methods in bioinformatics: genome assembly [4] efficient overlap detection among large sequences [5], quick quantification of transcriptomes [6], sequence search in large sequences collections [7], variant detection [8]; and can be identified as a generic need in large scale sequence analysis.
The main difficulty remains to design data structures that can handle billions of k-mers, so they can scale to large genomes or metagenomics instances. The methodology traditionally adopted

three main strategies. A first one is to index fixed-size words from sequences in sets structures that enable presence/absence queries. This strategy often relies on Bloom filters, and is for instance used to allow the search of sequences in thousands of indexed raw datasets [9], or for assembly [10, 11]. A second strategy uses full-text indexes that can localise words of arbitrary length in the sequence or set of sequences. They commonly rely on FM-indexes [12]. These methods can offer extremely memory efficient indexes but can present the inconvenient of a high construction cost and a reduced throughput compared to hash based methods. Finally, some data structures propose generic associative indexes. Based on hash tables [8] and/or filters [7], they allow to store (*key=k-mer, value*) pairs. This way, k-mers can be associated to information of any nature, for instance to their original dataset(s) [6], or counts [13]. The presented work pertains to this latter category.
Hash tables' cost can be illustrated by pioneer works that introduced associative structures such as Cortex [8]. Using a colored De Bruijn graph, it represents the k-mers associated to their datasets of origin and allows quick queries. However, it cannot scale up to more than a dozen datasets. Such difficulty motivated recent improvements [14] based on minimal perfect hash functions (MPHF) [15]. Building indexes upon such techniques has the advantage to yield powerful space and time compromises.
However, MPHFs do not represent sets of k-mers. Membership operations and stranger keys rejection must be handled using additional information. In a previous work we proposed to add an additional structure to associate to each k-mer a fingerprint [13], in order to obtain a probabilistic associative structure. The fingerprint is a hash of the k-mer, as a consequence the structure presents a false positive rate due to hash collisions, that depends of the fingerprint size.
More recently, Pufferfish's [16] authors took advantage of the possibility to assemble k-mers using compacted De Bruijn graphs [17, 18]. They store the position of each k-mer in the set of assembled sequences and can handle stranger k-mers by seeking their sequence at the position indicated by the index. De Bruijn graphs are convenient representations of k-mers collections since they collapse redundancy in their vertices that represent the k-mers set. There are numerous efficient De Bruijn graph representations (succinct data structures such as BOSS [19], efficient representations of De Bruijn graphs vertices such as DBGFM [20] or deGSM [21]), however they differ from Pufferfish and from the scope of this work since they are not associative structures.
In this contribution, we propose a novel, exact associative structure for k-mers, able to scale to extremely large nucleotide content while being memory and time efficient. Although it is based on a MPHF, this structure is deterministic and yields no false positives at query. It enabled the indexation of the 18 billions of 31-mers from the axolotl genome using 62.4GB of RAM (≈ 28 bits per k-mer) in 10 hours. The constructed index was able to perform more than a million query per second. Contrary to works dedicated to a particular application (colored De Bruijn graphs [22], quantification [6]), we propose a generic associative index that can be used for different purposes. To this extent, we implement a header-only library for user-friendly integration to various projects. We demonstrate its performances on different datasets to illustrate its potential applications on various issues.

2 Methods

2.1 Outline

2.1.1 Inputs/outputs

We construct our index from a set of sequences representing the nodes of a compacted De Bruijn graph called unitigs. We will consider a graph of unitigs, i.e. a compacted De Bruijn graph as defined in [18]. Unitigs are simple paths from a De Bruijn graph, with all but the first vertex (respectively last vertex) having a in-degree of 1 (respectively an out-degree of 1), compacted into a single vertex in the graph of unitigs.
For each k-mer present in the input graph, the index returns a unique identifier $i \in [1, N]$ with N the total number of k-mer, and -1 for any other k-mer.

2.1.2 Index construction

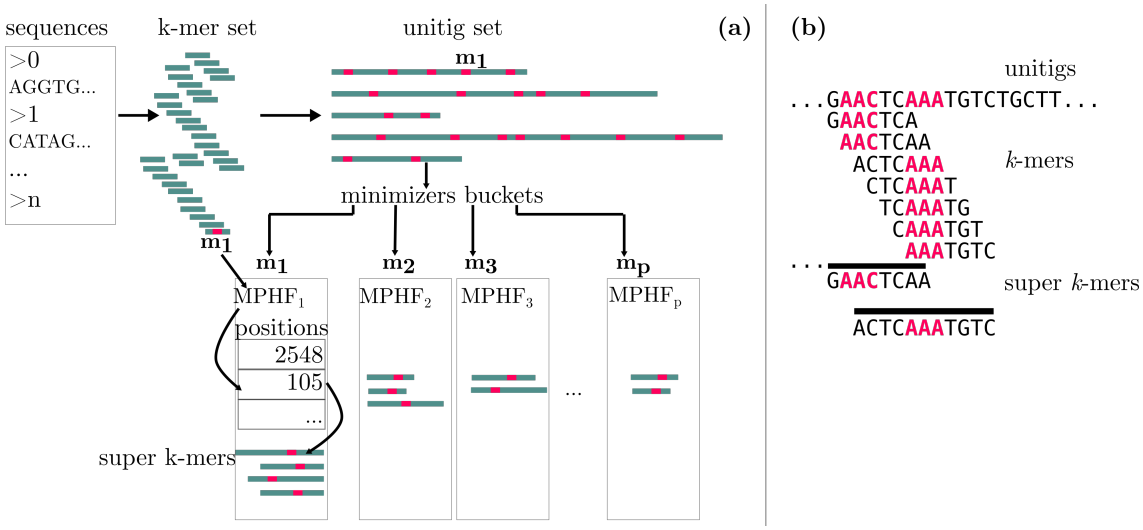

Figure 1: (a) Overview of BLight's method. A compacted De Bruijn graph is built from of the initial set of k-mers. Buckets are built upon the split of unitigs from the graph into super k-mers. Each bucket is represented by a MPHF that associates k-mers to their positions in the bucket. When querying, a k-mers is directly sent to the relevant bucket for lookup using its minimizer. (b) Example of how a unitig is split into super k-mers per minimizers (in red).

First, we split the graph into several sets of k-mers called buckets, depending on the minimizers [23] of the k-mers. For a minimizer size m, 4^m buckets will be constructed where each bucket will handle all k-mers with a given minimizer. (See also algorithm 1 in Appendix for the detailed bucket construction procedure). Once filled, for each bucket a specific MPHF is built from its k-mer set. Then, for a given bucket and a given k-mer, the MPHF returns a unique identifier. We use this identifier to associate to each k-mer its position in its corresponding bucket sequences to ensure that it can be retrieved. Those position arrays are retained as bit-vectors, where each k-mer position needs $log2(bucket\_size)$ to be encoded. Therefore each bucket handle k-mers that share a

given minimizer, contains the k-mer sequences and has a MPHF that associate to each k-mer its position in the bucket sequences.

2.1.3 Query

The query is done in two times (Figure 2, also Algorithm 3 in Appendix): first the relevant bucket is found according to the k-mer minimizer. Second, the MPHF is queried with the k-mer. MPHFs can return false positives at query, never false negatives. This means that for an existing k-mer, the MPHF returns its identifier that can be used to get its position in its bucket (Figure 2 (a)). For an alien k-mer, either no information is returned, meaning the k-mer is not present, or the MPHF falsely indicates a position (Figure 2 (b)). Then, for any k-mer for which the MPHF returned a position p, a lookup is performed in the sequences bucket at position p. This way, we check that the right k-mer sequence is read from this position. If not, we return a -1 identifier to indicate this k-mer is an alien. It is thus important to notice that, despite relying on the probabilistic MPHF response, our structure always permits deterministic queries.

2.2 Implementation details

2.2.1 Super k-mers

In order to represent k-mers from a bucket in a memory efficient way, we rely on the notion of super k-mers as described in [24]. The intuition behind the idea of super k-mers is that overlapping k-mers will often share the same minimizer. Therefore a group of x successive k-mers from a unitig sharing the same minimizer can be encoded as a word of length $k + x - 1$ (Figure 1 (b)). We use super k-mers in order to store more efficiently k-mers in each bucket. These sequences and their associated minimizers are written on the disk, which allows to reload the index while skipping the first bucket construction step. In order to easily access the sequences of the original graph, super k-mer are stored in gzipped FASTA format. Meta-data linked to each bucket and super k-mer are written in FASTA headers.

2.2.2 Minimizer size

The number of buckets increases exponentially (4^m) as m the minimizer size increases. As a consequence buckets contain less k-mers at higher m values, which means that the bit-vectors used to encode positions in each buckets are also globally reduced (as each k-mer position requires $log2(bucket\_size)$ to be encoded).
Thus, increasing the size of the minimizers seems to be interesting to encode k-mers positions in a more efficient way. However, increasing the number of buckets also means having an additional overhead as internal information, such as the beginning and the size of buckets, have to be stored for each bucket independently of the input. As this overhead is exponential, it can represent significant amount of memory when m is above 10. Moreover, another downside of a high m is that overlapping k-mer are less likely to share a large minimizer. Therefore, a larger m tends to produce more and smaller super k-mers and this fragmentation raises the total amount of nucleotide necessary to represent a set of k-mer.

2.2.3 Sparse index

One can index k-mer approximate position via sub-sampling to reduce the memory impact of the position encoding. Thus the trade-off involves the query time (supplementary time is required to retrieve some k-mers) against the index size (by storing less position information, the bit-vectors size for each bucket can be reduced). More precisely, in order to save b bits when encoding the positions, one indexes $\tilde{p} = p_i//2^b$ for each k-mer k_i at position p_i. Thus, 2^b k-mers share the same indexed position $\tilde{p}$. From this position, we have to check at most to 2^b k-mers during the query to ensure the k-mer exists. An example is given in Figure 2 (c). In the case of queried alien k-mers, we always are in this worst case. We experimentally show (Table 2) that up to $b = 6$, the additional time has no impact since the query bottleneck is to compute k-mers minimizers and query their MPHF identifiers. We observe that for $b > 6$ the query becomes increasingly slower. The value $b = 6$ has been therefore chosen as the default value for all the presented results unless said otherwise.

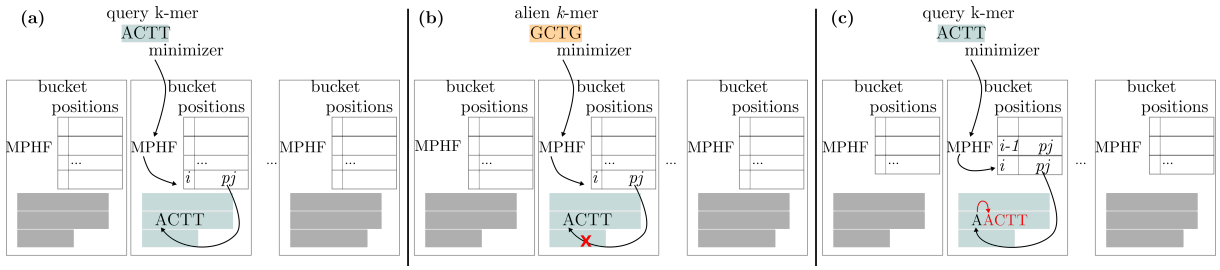

Figure 2: Queries in the index: (a) query of an indexed k-mer, (b) query of an alien k-mer, (c) query in the sparse index.

3 Results

We benchmark our structure on three different use cases. First, we select large, complex, reference genomes of increasing sizes (starting from the human that represents a moderate reference size, up to the currently larger available at NCBI, i.e. the axolotl *Ambystoma mexicanum*, with 32Gbp) in order to demonstrate how the structure scales to these objects. We use these datasets to demonstrate how the minimizer size impacts the performances of our structure, as well as to illustrate the trade-off obtained when using the sparse version of our index. On this application, we compare the current state-of-the art approach designed to index such data, Pufferfish.

Second, we demonstrate how our method can handle the indexation of raw reads datasets and show example of potential applications of such index. We use BLight to associate to each k-mer its number of occurrences across the datasets. We selected a dataset from TARA Oceans samples [25], that was previously used for the validation of a method we compare with [13]. We compare to two lightweight, recent ubiquitous k-mer abundance index: Short Read Connector (SRC) [13] and Squeakr [26]. Finally, we propose to use BLight as a proof-of-concept colored De Bruijn graph representation. Using 100 simulated RNA-seq datasets, we retain each k-mer's color (i.e. list of initial datasets) using our structure. We compare our performances to another recent in-memory colored De Bruijn graph exact representation: Mantis [7].

All experiments were performed on a cluster running with Intel(R) Xeon(R) CPU E5-2420 0 @ 1.90GHz with 192GB of RAM, except for section 3.3 that was benchmarked using a machine running with Intel(R) Core(TM) i7-8650U CPU (1.90GHz with 4 cores and 8MB L3 cache size) with 32GB of RAM and Ubuntu 16.04.

3.1 Indexing up to top-largest reference genomes

Selected genomes To assess the impact of the proposed minimizer partitioning, we built a De Bruijn graph ($k = 31$) from several reference genomes and built the BLight index with different minimizer sizes on their graphs:

- The human reference genome (GRCh38.p12) of 3.2 Gbp, counting 2.5 billions k-mers and constructed with Bcalm2 [18] using 12 CPU hours and 6.6GB of RAM.
- The latest assembly of *Pinus taeda* (GCA_000404065.3) of 22 Gbp, counting 10.5 billions k-mers and constructed with Bcalm2 using 68 CPU hours and 17.3GB of RAM.
- The latest assembly of *Ambystoma mexicanum* (GCA_002915635.2) of 32 Gbp, counting 18.3 billions k-mers and constructed with Bcalm2 using 107 CPU hours and 44.1GB of RAM.

To compare BLight to Pufferfish we also included the bacterial genomes graph from Pufferfish paper, constructed from more than 8000 bacterial genomes counting 5,4 billions k-mers.

Index construction We constructed the BLight index with several minimizer sizes and report in Figure 3 the amount of memory necessary to encode the graph super k-mer, to encode the k-mers positions and the actual maximal memory peak during construction. All value are expressed as bits per k-mers. We globally observe that a higher minimizer size leads to a slight augmentation of the memory needed to encode the graph sequences and a neat decrease of the memory allowed to the position encoding. With a uniform coverage we expect the bit required to encode a k-mer position to be $\approx log2(\frac{bucket\_size}{4^m})$ resulting of an expected gain of 2 bits per k-mer for increasing the minimizer size of one. Interestingly, the observed results are close to this trend in practice on our indexed genomes on the Figure 3, where we report the BLight results for the index construction on the chosen genomes graphs. We summarized the best result obtained with BLight with respect to Pufferfish in Table 1. We show that BLight is able to construct its index using substantially less memory than Pufferfish construction steps, but also than Pufferfish index itself. We also show improved construction time and we want to highlight the fact that unlike Pufferfish, no pre-processing is needed on the graph for the index construction, since only the unitig sequences in a FASTA file are needed. All detailed results are presented in the Appendix in Table 6.

Query time and impact of sub-sampling factor We compared our throughput according to Pufferfish in Table 2. We observe that our query is slightly slower than Pufferfish's on a single CPU. However the query throughput can be significantly increased by using multiple threads. Either way both indexes are able to propose a very high throughput of the order of the million queries per second. As mentioned before, using a sub-sampling factor enables to control the memory usage at the expense of the query performances. However, we observe that below $b = 7$ the impact on our query time is negligible. We therefore chose b=6 as the default sub-sampling value for all other benchmarks (Table 2 shows only even b values, a full report can be found in Appendix).

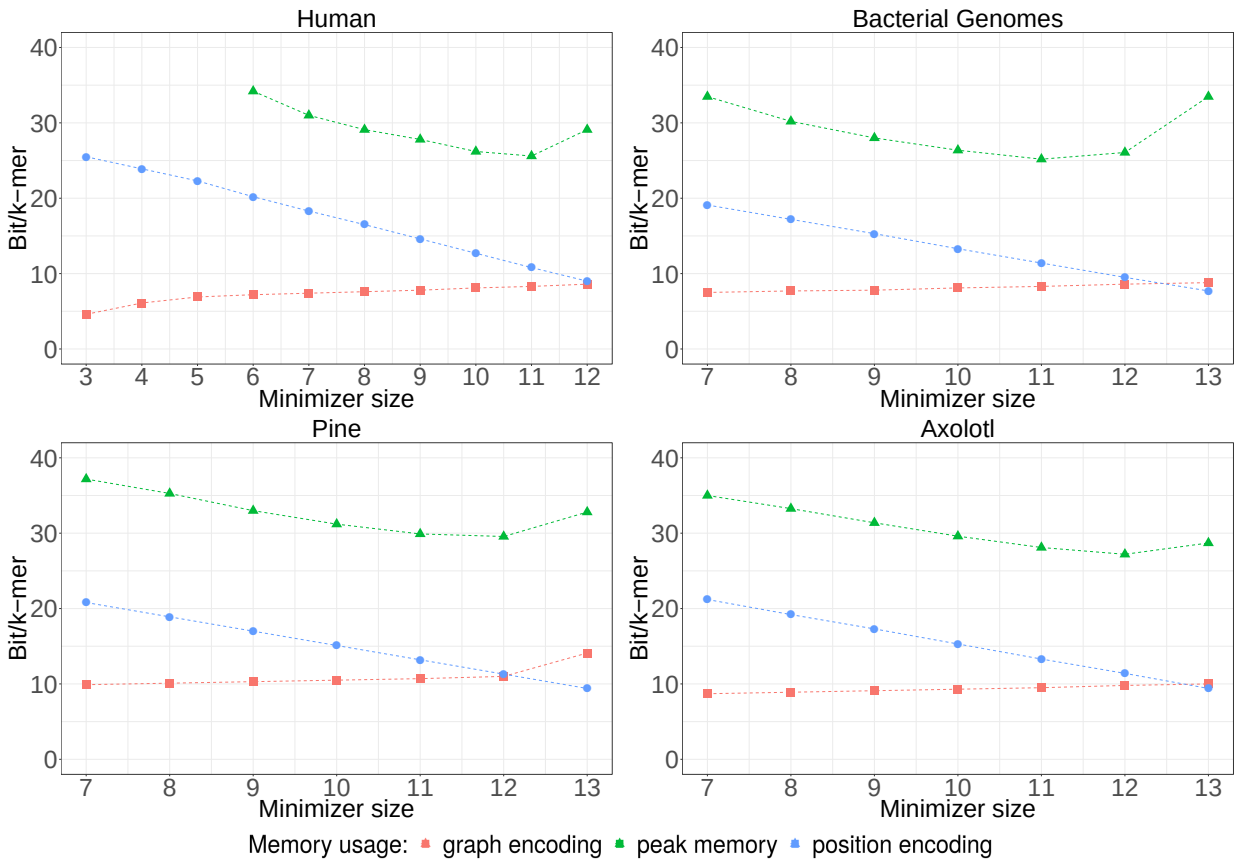

Figure 3: Detailed memory usage in bits per k-mer respectively on the human, bacterial genomes, pine and axolotl graphs during the BLight index construction.

Blight as a De Bruijn graph representation BLight can be used as a high-throughtput De Druijn graph representation, but is not optimal in comparison to space-efficient graph structures. For each species, we computed the theoretical bound indicated by Conway & Bromage (CB) [27] (for $k = 31$, human: 32.2, pine: 30.15, axolotl: 29.35 and bacterial genomes set: 31.11 GB). Contrary to our representation and Pufferfish's, CB does not take into account the possibility to assemble k-mers. This is why we use less space than CB on the human genome. Our structure and Pufferfish's use $\mathcal{O}(log(genome\_size))$ bits per k-mer, thus exceeds CB on larger genomes. Other FM-index based methods such as DBGFM maintain compressed, non-associative representations of the De Bruijn graph, thus are always more space-saving than ours. We follow a different trade-off, since we expect a faster query time in practice.

3.2 Indexing and k-mer counting in metagenomics datasets

In this section, we assess the ability of the BLight structure to index complex metagenomic datasets. We first chose to index all non unique k-mers of a TARA sample[1] containing five datasets ERR1712199, ERR1711907, ERR599280, ERR562434 and ERR1718455. BLight was able to build

[1] `https://www.ebi.ac.uk/ena/data/view/SAMEA2620556`

Genome	Pufferfish time	BLight time	Pufferfish memory	BLight memory	Bits/k-mer
Human	01:09	**00:29**	27.4	**8.0**	25.6
Bacterial genomes	03:07	**02:10**	75.0	**17.0**	25.2
Pine	NA	**06:48**	NA	**38.8**	29.6
Axolotl	NA	**10:11**	NA	**62.4**	27.2

Table 1: Comparison of the maximal memory usage in GB and the CPU time in minutes used during construction with BLight and Pufferfish from the De Bruijn graph. We were unable to construct Pufferfish's index for the pine and the axolotl genomes due to memory limitation. We want to point up that the Pufferfish index use less RAM than its construction steps. This index represent 17GB for the human genome and 40GB for the bacterial genomes.

b	Position encoding (bit/k-mer)	Query real reads (hh:mm)	Multi thread (hh:mm)	Query reference (hh:mm)	Multi thread (hh:mm)
0	18.7	00:23	**00:04**	00:29	**00:06**
2	16.7	00:27	00:05	00:32	00:07
4	14.7	00:27	00:05	00.33	00:07
6	12.7	00:28	00:05	00:36	00:07
8	10.7	00:45	00:08	01:04	00:13
10	8.7	02:28	00:22	02:44	00:31
Pufferfish	NA	**00:21**	00:21	**00:30**	00:30

Table 2: Influence of the sub-sampling parameter on the query time. We report the amount of memory used by the positions divided by the number of k-mers, the time required to query the real dataset SRR5833294 with one thread, the time required to query the real dataset with 6 thread, the time required to query all the k-mers of the reference graph and the time required to query this graph using 6 threads.

an index from the graph constructed from Bcalm2. The construction lasted less an hour and used 24.7 GB, which represents 27.1 bits per k-mer. This memory usage is higher than the bacterial genomes despite having a comparable amount of k-mers. This can be imputed to the expected graph fragmentation of a raw metagenomic sample.
In a second experiment we provide a proof of concept of an index able to associate to each k-mer its abundance among a dataset. We compare a naive usage of BLight with Squeakr and Short Read Connector counter (SRC) in exact mode in Table 3. If the memory usage of BLight is very reduced, the index construction is way slower than SRC or Squeakr. This can be explained by the fact that the proposed snippet has to perform query on each k-mer of the dataset to re-count the $k-$mer abundance. A real implementation should parse a k-mer counting result in order to initialize the

Tool	Peak memory (GB)	Time (hh:mm)
Squeakr exact	185.4	15:11
SRC counter exact	44.20	**05:12**
BLight count	**20.8**	19:49

Table 3: Performance comparison of an abundance index on the ERR599280 metagenomic dataset.

k-mer abundance as SRC. Moreover, SRC or Squeakr inexact modes are expected to be more space efficient but will yield false positives.

3.3 Colored De Bruijn graphs

Selected tools In this section, we propose to apply BLight to construct and query a colored De Bruijn graph. In colored De Bruijn graphs, vertices/edges of De Bruijn graphs receive labels (or colors) that represent their dataset(s) of origin. We implement a proof-of-concept colored De Bruijn graph using BLight, by simply adding a new step that loops other k-mers in each dataset to associate color information to k-mers using our index.
We chose to compare our approach to Mantis since it benefits from the most recent improvements and outperforms previous efficient colored De Bruijn graph methods [28, 29]. Mantis allows exact query and thus, can be fairly benchmarked with our method. Other approaches based on Sequence Bloom Trees [9, 30, 31] do not represent the colored De Bruijn graph but yet can provide the datasets that share k-mers with a query sequence. They explored the compromise of probabilistic membership query to reduce their costs. Though it is not exact, we include SSBT [31] to our benchmark for the sake of the comparison, as a recent Sequence Bloom Tree implementation.

Data generation and pre-processing We simulated a collection of 100×10M reads simulated RNA-seq human datasets (with GRCh38 reference) using the Flux Simulator [32]. We filtered and kept for further indexation the k-mers ($k = 31$) which abundance is at least 2 in each dataset (263,811,076 k-mers). For the query set, we randomly selected transcripts (of size $\geq k$) to build batches of increasing sizes (from Gencode transcripts GRCh38.p12). Before running each tool, the set of k-mers that they index must be computed. For Mantis, Squeakr exact is run each data set (peak memory 1.8GB RAM and 1:42 minutes). For BLight, we ran Bcalm2 over each datasets (21:16 minutes and 2,0GB RAM). For SSBT, we built Bloom filters for each datasets (20:06 minutes and 0.66GB RAM), setting Bloom filters sizes roughly as the number of distinct k-mers to index and using one hash function, as recommended. Then all tools were run single-threaded for indexing and query steps.

Index construction and query Index construction times and peak memory are reported in Table 4. SSBT is the fastest at construction (1min 46sec). BLight mitigates well construction time and memory, being second in terms of speed (2min 33sec) and having the more lightweight memory footprint (2.89 GB). Moreover, Mantis and BLight allow parallel index construction. Using only 4 threads, BLight becomes the fastest tool to build the index (1min 42sec for BLight versus 3min 38 sec for Mantis), and remains with the lowest memory consumption (2.89 GB versus 11.8 GB for Mantis). For Mantis and BLight, we could report detailed performances for each step in Table 8 in Appendix: both index construction and color addition steps are achieved in less time in comparison to Mantis (Table 8).
We show the query results in Table 5. Since query time can evolve with the batch size, we observe how three batches of 10, 100 and 1000 transcripts are queried to the three methods' indexes. BLight's query times are the fastest in each case, requiring up to two orders of magnitude less than Mantis and SSBT.

	index constr. total (mm:ss)	peak resident (GB)
Mantis	7:53	11.8
SSBT	**1:46**	3.34
BLight	2:33	**2.89**

Table 4: Comparison of Mantis, SSBT and BLight for index construction from 100 RNA-seq datasets with k-mers of abundance ≥ 2.

Query (nb. transcripts)	Mantis (seconds)	SSBT (seconds)	BLight (seconds)
10	1.81	64.0E-1	**1.35E-2**
100	1.78	1.82	**8.77E-2**
1000	2.25	1.88E+1	**1.58**

Table 5: Query times for batches of 10, 1000 and 10000 transcripts from Gencode, on the indexes built in previous table.

4 Conclusion and future work

In this work we propose BLight, an ubiquitous, efficient and exact associative structure for indexing k-mers, relying on De Bruijn graphs. Based on efficient hashing techniques and light memory structure, we believe that the proposed index has a very interesting time/memory compromise, being able to perform millions query per second and using less than 32 bits per k-mer on our experiments. We demonstrate that BLight is able to index the largest available genomes to date using reasonable amount of memory while outperforming state of the art methods in both construction time and memory. We also shown it could be relevant and efficient on raw transcriptomic and metagenomic data. We therefore believe that a huge number of methods could rely and benefit from this structure due to its wide application spectrum. To that extent, we implemented a user friendly library along with different snippets to allow our method to be usable in practical cases.
A vast amount of improvements could be brought to the proposed backbone structure. Specific minimizer schemes could be designed to obtain the smallest possible buckets, supplementary structures to sort and rank the super k-mers could allow faster query or reduced position encoding. Such optimizations could improve the global resources usage of the index or provide different time/memory trade-offs.
Another relevant continuation of this work would be to adapt this structure for the specific needs of applications requiring such scalable data structure. In particular, the indexing challenge of colored De Bruijn graph (or more generally to answer large sequence search problems as defined in [9]) have caught the interest of a community and we believe that the proposed work could be relevant. The field was first renewed very recently through two main categories, methods that adopted succinct graph structures and color sets to save space [28, 29, 22], and structures that explored very memory efficient structures at the cost of false positives [9, 30, 31]. We chose to face the problem by enhancing the initial k-mers sets representation, and compared our approach to a representative of each category. We demonstrated the promising results of BLight for encoding colors per k-mer, even when competing with approximate membership query methods such as SSBT. At the moment, results were performed on a rather restrained set of datasets and an interesting future work includes a better representation colored De Bruijn graph using color factorization techniques.

Acknowledgements

We thank Rob Patro, Fatemeh Almodaresi, Tatiana Rocher and Rayan Chikhi for their support and interesting discussions on this project. This work was supported by the ANR Transipedia (ANR-18-CE45-0020), and benefited from Université de Lille HPC Cloud computing resources.

References

[1] Rasko Leinonen, Hideaki Sugawara, Martin Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. *Nucleic acids research*, 39(suppl_-1):D19–D21, 2010.

[2] Aleksey V Zimin, Kristian A Stevens, Marc W Crepeau, Daniela Puiu, Jill L Wegrzyn, James A Yorke, Charles H Langley, David B Neale, and Steven L Salzberg. An improved assembly of the loblolly pine mega-genome using long-read single-molecule sequencing. *Gigascience*, 6(1):1–4, 2017.

[3] Sergej Nowoshilow, Siegfried Schloissnig, Ji-Feng Fei, Andreas Dahl, Andy WC Pang, Martin Pippel, Sylke Winkler, Alex R Hastie, George Young, Juliana G Roscito, et al. The axolotl genome and the evolution of key tissue formation regulators. *Nature*, 554(7690):50, 2018.

[4] 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.

[5] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

[6] Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature biotechnology*, 34(5):525, 2016.

[7] Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. In *Research in Computational Molecular Biology: 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812, page 271. Springer, 2018.

[8] 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, 2012.

[9] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300, 2016.

[10] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1):22, 2013.

[11] Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome research*, pages gr–214346, 2017.

[12] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[13] Camille Marchet, Lolita Lecompte, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo. A resource-frugal probabilistic dictionary and applications in bioinformatics. *Discrete Applied Mathematics*, 2018.

[14] Avi Srivastava, Fatemeh Almodaresi, Hirak Sarkar, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 06 2018.

[15] 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.

[16] Fatemeh Almodaresi, Hirak Sarkar, and Rob Patro. A space and time-efficient index for the compacted colored de bruijn graph. *bioRxiv*, page 191874, 2017.

[17] Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, 2016.

[18] 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.

[19] Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In *2015 Data Compression Conference*, pages 383–392. IEEE, 2015.

[20] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de bruijn graphs. *Journal of Computational Biology*, 22(5):336–352, 2015.

[21] Hongzhe Guo, Yilei Fu, Yan Gao, Junyi Li, Yadong Wang, and Bo Liu. degsm: memory scalable construction of large scale de bruijn graph. *bioRxiv*, page 388454, 2018.

[22] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, 2018.

[23] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[24] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[25] Eric Karsenti, Silvia G Acinas, Peer Bork, Chris Bowler, Colomban De Vargas, Jeroen Raes, Matthew Sullivan, Detlev Arendt, Francesca Benzoni, Jean-Michel Claverie, et al. A holistic approach to marine eco-systems biology. *PLoS biology*, 9(10):e1001177, 2011.

[26] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2017.

[27] Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[28] 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.

[29] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: A succinct colored de bruijn graph representation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 88. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[30] Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. AllSome Sequence Bloom Trees. In *RECOMB 2017 - 21st Annual International Conference on Research in Computational Molecular Biology*, Hong Kong, China, May 2017.

[31] Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *Journal of Computational Biology*, 25(7):755–765, 2018.

[32] Thasso Griebel, Benedikt Zacher, Paolo Ribeca, Emanuele Raineri, Vincent Lacroix, Roderic Guigó, and Michael Sammeth. Modelling and simulating generic rna-seq experiments with the flux simulator. *Nucleic acids research*, 40(20):10073–10083, 2012.

Appendix

Result details

In Table 8 we show details about Mantis and BLight's runtimes on the index construction for the 100×10 million reads RNA-seq datasets.

Algorithms

Genome	minimizer size	graph encoding	position encoding	Total RAM	construction time
Human	3	**4.6**	25.5	26.4	00:50
Human	4	6.1	23.9	18.0	00:54
Human	5	6.9	22.3	13.3	00:47
Human	6	7.2	20.2	10.7	00:37
Human	7	7.4	18.3	9.7	00:33
Human	8	7.6	16.5	9.1	00:29
Human	9	7.8	14.6	8.7	**00:27**
Human	10	8.1	12.7	8.2	00:28
Human	11	8.3	10.8	**8.0**	00:29
Human	12	8.6	**9.0**	9.1	00:30
Human	Pufferfish	NA	NA	27.4	01:09
Metagenomic sample	6	**8.4**	21.6	35.0	01:06
Metagenomic sample	7	8.6	19.7	31.9	01:06
Metagenomic sample	8	8.8	17.8	29.2	00:53
Metagenomic sample	9	9.0	15.8	27.6	01:02
Metagenomic sample	10	9.2	13.8	25.9	**00:52**
Metagenomic sample	11	9.5	11.9	**24.7**	00:56
Metagenomic sample	12	9.7	9.9	25.0	01:03
Metagenomic sample	13	10.0	**8.0**	29.8	01:11
Bacterial genomes	7	**7.5**	19.1	22.6	02:29
Bacterial genomes	8	7.7	17.2	20.4	02:19
Bacterial genomes	9	7.8	15.3	18.9	02:18
Bacterial genomes	10	8.1	13.3	17.8	**02:09**
Bacterial genomes	11	8.3	11.4	**17.0**	02:10
Bacterial genomes	12	8.6	9.5	17.6	02:15
Bacterial genomes	13	8.8	**7.7**	22.6	02:29
Bacterial genomes	Pufferfish	NA	NA	75.0	03:07
Pine	7	**9.9**	20.8	48.8	07:44
Pine	8	10.1	18.9	46.3	07:13
Pine	9	10.3	17.0	43.3	07:01
Pine	10	10.5	15.1	41.0	**06:37**
Pine	11	10.7	13.2	39.3	06:40
Pine	12	11.0	11.3	**38.8**	06:48
Pine	13	14.1	**9.4**	43.1	07:00
Axolotl	7	**8.7**	21.2	80.1	12:04
Axolotl	8	8.9	19.2	76.1	11:18
Axolotl	9	9.1	17.3	71.8	10:23
Axolotl	10	9.3	15.3	67.8	09:54
Axolotl	11	9.5	13.3	64.3	**09:48**
Axolotl	12	9.8	11.4	**62.4**	10:11
Axolotl	13	10.0	**9.4**	65.7	10:18

Table 6: Influence of the minimizer size on different genome. The amount of memory used to encode the positions and the graph divided by the number of k-mer is indicated. We also report to the maximum memory and the CPU time used during the index construction. We were not able to construct the Pufferfish index on the Pine and Axolotl genomes.

b	Position encoding (bit/k-mer)	Query real reads (hh:mm)	Multi thread (hh:mm)	Query reference (hh:mm)	Multi thread (hh:mm)
0	18.7	00:23	**00:04**	00:29	**00:06**
1	17.7	00:26	00:05	00:32	00:07
2	16.7	00:27	00:05	00:32	00:07
3	15.7	00:27	00:05	00:32	00:07
4	14.7	00:27	00:05	00.33	00:07
5	13.7	00:28	00:05	00:34	00:07
6	12.7	00:28	00:05	00:36	00:07
7	11.7	00:31	00:06	00:43	00:09
8	10.7	00:45	00:08	01:04	00:13
9	9.7	01:24	00:13	01:38	00:18
10	8.7	02:28	00:22	02:44	00:31
Pufferfish	NA	**00:21**	00:21	**00:30**	00:30

Table 7: Influence of the sub-sampling parameter on the query time. We report the amount of memory used by the positions divided by the number of k-mers, the time required to query the real dataset SRR5833294 with one thread, the time required to query the real dataset with 6 thread, the time required to query all the k-mers of the reference graph and the time required to query this graph using 6 threads.

	index constr. (mm:ss)	build colors (mm:ss)	peak resident (GB)
Mantis	1:56	5:57	11.8
BLight	**01:43**	**00:49**	**2.89**

Table 8: Details of performances of Mantis and BLight at indexing k-mers of abundance ≥ 2. Several steps are separated: index construction (`mantis build` command for Mantis) and colors are added to the index (`mantis mst` command for Mantis).

Algorithm 1: Outline of the bucket creation

Data: A set of unitigs $\mathcal{U} = u_1, u_2, \dots u_n$
Data: m the size of minimizers
Result: A list of buckets $\mathcal{B}$, i.e. a list of lists of k-mers

```
create_bucketMap =<<>>;
forall unitig u_i ∈ U do
    forall k-mer k_ij from u_i do
        min_ij = k_ij.compute_minimizer(m);
        if min_ij ∉ bucketMap then
            new_bucket =<>;
            bucketMap[min_ij] = new_bucket;
        bucketMap[min_ij].write(k_ij);
```

Algorithm 2: Bucket index

Data: A list of buckets $\mathcal{B}$
Result: An associative index $\mathcal{T}$

```
forall bucket b_i ∈ B do
    S = k_ij, k_ip, ... k-mers ∈ b_i;
    create_MPHF_{b_i}(S) ;
    forall k-mer k_ij from b_i do
        p_i = position of k_ij in b_i;
        index = MPHF[k_ij];
        T[index] = p_i;
```

Algorithm 3: Query a k-mer

Data: The associative index $\mathcal{T}$
Data: A k-mer k_i
Data: m the minimizer size used to build $\mathcal{T}$
Result: An integer: either position p_i if the k-mer is present; -1 if the k-mer is absent.

```
min_i = k_i.compute_minimizer(m) → bucket b_i;
p_i = MPHF_{b_i}[k_i]
```