Holley and Melsted *Genome Biology* (2020) 21:249
https://doi.org/10.1186/s13059-020-02135-8

Genome Biology

SOFTWARE Open Access

Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs

Guillaume Holley* and Páll Melsted

*Correspondence:
guillaumeholley@gmail.com
Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, Reykjavík, Iceland

Abstract

Memory consumption of de Bruijn graphs is often prohibitive. Most de Bruijn graph-based assemblers reduce the complexity by compacting paths into single vertices, but this is challenging as it requires the uncompacted de Bruijn graph to be available in memory. We present a parallel and memory-efficient algorithm enabling the direct construction of the compacted de Bruijn graph without producing the intermediate uncompacted graph. Bifrost features a broad range of functions, such as indexing, editing, and querying the graph, and includes a graph coloring method that maps each *k*-mer of the graph to the genomes it occurs in.

Availability: https://github.com/pmelsted/bifrost

Introduction

The de Bruijn graph is an abstract data structure with a rich history in computational biology as a tool for genome assembly [1, 2]. With the advent of high throughput sequencing, the Overlap Layout Consensus (OLC) framework frequently used to assemble Sanger sequencing data [3] was progressively replaced in favor of de Bruijn graph-based methods. Since 2008, a wide range of genome assemblers based on the de Bruijn graph have been released [4–10]. Although single molecule sequencing technologies [11, 12] have reintroduced the OLC framework as the method of choice to assemble long and erroneous reads [13–16], de Bruijn graph-based methods are nonetheless used to assemble and correct long reads [17, 18]. Overall, de Bruijn graphs have found widespread use for a variety of problems such as de novo transcriptome assembly [19], variant calling [20], short read compression [21], short read correction [22], long read correction [17], and short read mapping [23] to name a few. The colored de Bruijn graph is a variant of the de Bruijn graph which keeps track of the source of each vertex in the graph [24]. The initial application was for assembly and genotyping, but it has also found use in pan-genomics [25], variant calling [26], and transcript quantification methods [27].

Despite serving as a building block for many methods in computational biology, the de Bruijn graph adoption is hindered by two factors. First, the memory usage and computational requirements for building de Bruijn graphs from raw sequencing reads are considerable compared to alignment to a reference genome, while only a handful of tools have focused on de Bruijn graph compaction [28–33]. Second, de Bruijn graph construction usually requires tight integration with the code. In the best case, software libraries for building and manipulating de Bruijn graphs are used [34, 35], but in most cases, data structures to index the de Bruijn graph are re-implemented. Those downsides are intensified in the colored de Bruijn graph for which the memory consumption of colors rapidly overtakes the vertices and edges memory usage [36]. For this reason, a lot of attention has been given to succinct data structures for building the colored de Bruijn graph [30, 31, 36–41] and data structures for multi-set k-mer indexing [42–47]. In the following, we focus on tools for constructing compacted de Bruijn graphs (cdBGs) with or without colors. We refer the reader to the survey of [48] for more details about k-mer-based data structures as well as the reviews of [25] and [49] for data structures to index collections of k-mer sets.

TwoPaCo [28] is a highly parallel construction tool for the cdBG. It builds progressively the cdBG from assembled genomes by identifying *junction* k-mers which are either branching or located at the extremities of unitigs. A Bloom filter is first used to approximate the graph and a hash table is subsequently employed to remove false positives. The approach taken by BCALM2 [29] is orthogonal to the one of TwoPaCo: rather than identifying junction k-mers, BCALM2 incrementally assembles k-mers into unitigs until junction k-mers are reached. K-mers are partitioned according to their minimizers, and partitions are compacted independently in parallel. A final step glues the compaction of different partitions together. Note that BCALM2 can process assembled genomes as well as short read data. deGSM [50] performs an external sorting of the k-mers from the input sequences and then constructs a Burrows-Wheeler transform (BWT) [51] of the unitigs from which the final graph is extracted. SplitMEM [30] uses the suffix tree [52] to construct a ccdBG. Unitigs of the graph are derived from the set of Maximum Exact Matches in the input genomes, while colors are implicitly encoded in the suffix tree. SplitMEM is not adapted to short read data input and splits the unitigs to ensure all k-mers of each unitig share the same set of colors. Baier et al. [31] provided two algorithms improving SplitMEM with a lower time complexity using a Compressed Suffix Tree and the BWT. PanTools [33] creates first an uncompacted k-mer index from which are derived unitigs. By iterating over the input assembled genomes, k-mers that have not been visited yet are extended to form unitigs, possibly leading to the merging and splitting of previously created unitigs. The graph index is maintained in a database providing edit operations such as updating the graph with additional data. PanTools was specifically designed for pangenomic applications with assembled genomes in input and allows gene annotations in the graph.

In this paper, we present Bifrost, a software for efficiently constructing, indexing, and querying the colored and compacted de Bruijn graph (ccdBGs), both in terms of runtime and memory usage. The data structures and algorithms implemented in Bifrost are specifically tailored for fast and lightweight construction, querying, and dynamic manipulation of compacted de Bruijn graphs, both regular and colored. The software is designed to take advantage of multiple cores and modern processors instruction sets (SIMD operations).

Bifrost is also available as a C++11 software library with minimal external dependencies and allows developers to build on top of an efficient de Bruijn graph engine by using the Bifrost API. Bifrost has been successfully employed for alignment- and reference-free phylogenomics [53] as well as bacterial genomes querying of genes linked to pathogenicity islands and *fluoroquinolone* resistance [54].

Results

We benchmarked Bifrost against state-of-the-art software on publicly available dataset. We focus on three representative use cases: cdBG construction, cdBG querying, and cdBG coloring. All experiments were run of a server with an 16-core Intel Xeon E5-2650 processor and 256G of RAM. Running time was measured as wall clock time using the `time` command, and peak memory was measured by `ps`.

cdBG construction

We constructed the cdBG of the NA12878 human genome short read dataset from the Genome In A Bottle consortium [55]. The dataset is downsampled from 300-fold to 30-fold coverage to reflect normal sequencing depth, resulting in about 696 million 150-bp paired-end sequences.

We compared Bifrost to BCALM2 because of its low computational requirements and versatility as it can build a cdBG from short read data or assembled genomes. BCALM2 can be configured for different memory usage where a lower memory usage results in a longer running time. In our experiments, it was configured with the maximum memory usage of Bifrost for each *k*-mer size tested. Additionally, BCALM2 uses by default up to 5 GB of disk space while Bifrost does not use any disk except for the final output. Results are shown in Table 1, and summaries of the unitig N50, *k*-mer cardinality, and unitig cardinality in each graph built are reported in Table 2.

Table 1 Time and memory comparison of Bifrost and BCALM2 for different *k*-mer sizes and number of threads during graph construction

	Tool	*k*-mer size	Number of threads			
			1	4	8	16
Time (h)	Bifrost	31	**20.81**	**8.53**	**6.10**	**5.55**
		63	**14.38**	**4.20**	**2.40**	**2.00**
		95	**12.51**	**3.88**	**2.25**	**1.58**
		127	**9.56**	**2.96**	**1.81**	**1.41**
	BCALM2	31	44.25	14.11	8.48	6.33
		63	N/A	25.6	13.96	8.71
		95	N/A	39.91	21.45	12.56
		127	N/A	N/A	27.73	16.15
Memory (GB)	Bifrost	31	39.59	39.58	39.59	39.60
		63	37.77	37.77	37.77	37.78
		95	44.33	44.30	44.30	44.32
		127	55.88	55.86	55.86	55.86
	BCALM2	31	**36.00**	**35.66**	**35.61**	**35.58**
		63	N/A	**29.83**	**29.73**	**29.64**
		95	N/A	**33.47**	**33.51**	**33.66**
		127	N/A	N/A	**43.42**	**53.77**

Best results are highlighted. N/A indicates the result is unavailable because the computation took more than 48 h

Table 2 Unitig N50, *k*-mer, and unitig cardinalities in cdBGs built from NA12878 for different *k*-mer sizes

k-mer size	*k*-mer cardinality	Unitig cardinality	Unitig N50
31	2,675,559,250	80,478,269	421
63	2,991,703,769	28,262,463	950
95	3,058,681,425	16,691,669	1299
127	2,702,556,396	44,221,433	297

Bifrost was consistently faster than BCALM2, up to a factor 15.32, on all *k*-mer sizes and number of threads tested. For increasing *k*-mer sizes, Bifrost construction time kept decreasing while BCALM2 construction time increased. However, BCALM2 used up to 24.3% less memory than Bifrost. Memory usage for a fixed *k*-mer size was fairly constant for both tools across different number of threads, except for BCALM2 using $k = 127$.

cdBG querying

We compared Bifrost to two tools for querying dBGs based on the *k*-mer composition of the queries, namely Blight [56] and Mantis [45]. The dataset used for the graph index was the NA12878 dataset from the Genome In A Bottle consortium described in the "cdBG construction" section. For querying, Bifrost takes as input the graph it constructed and builds an index for querying *k*-mers. Mantis requires processing the unitigs of the graph with Squeakr [57] to produce a compressed table of all *k*-mers present. Mantis then builds an index directly from the output of Squeakr for querying. Blight takes as input a graph created by BCALM2. All indexes were created using $k = 31$ and 16 threads.

To query the graph, we used 30 million single-end reads from the NA12878 short read dataset that was used to construct the reference graph.

Note that both Bifrost and Mantis return query hits for every query while Blight only returns the total number of *k*-mers found in the graph from all input queries. Furthermore, Mantis and Blight cannot be configured to return the presence or absence of a query based on different *k*-mer inclusion rates. Hence, Bifrost was queried initially with parameter $e = 1.0$ to indicate that an input query is returned present in the graph only if all of its composing *k*-mers are present. This is done to ensure that all methods query the graph for all *k*-mers in the read. Results are shown in Table 3. Finally, Bifrost enables graph querying based on *k*-mers with up to one substitution or indel. Table 4 shows the performance of Bifrost with different *k*-mer inclusion rates, where $e = \theta$ requires at least the presence of θ fraction of the *k*-mers in the graph, both using exact or inexact *k*-mers. Querying for inexact *k*-mers, where an edit distance of 1 is allowed, increases the number of hits but requires more running time. However, even in the case where all *k*-mers are queried, the inexact version is still competitive with Blight and Mantis which only perform exact *k*-mer queries. Overall, the results show that Bifrost is the fastest at querying, while using 26.8 GB of memory, whereas Blight uses less memory at the expense of speed. The low memory usage of Blight is partially explained by the fact that Blight maintains its index in main memory but stores subsequences of the graph on disk.

cdBG coloring

We constructed ccdBGs with $k = 31$ for a maximum of 117,913 assembled genomes of *Salmonella*. The input represents all publicly available *Salmonella* assemblies from the

Table 3 Running time and memory usage for indexing and querying a de Bruijn graph for 30 million short reads

Tool	Process	Time (m)	Memory (GB)
Bifrost	Build	**333**	39.6
	Index	**11.1**	26.8
	Query	**4.7**	26.8
	Query-total	**16.4**	26.8
BCALM2	Build	380	**35.58**
Blight	Index	80	**8.3**
	Query	13.6	**8.3**
	Query-total	93.6	**8.3**
Squeakr	Build	1147	80
Mantis	Index	54	17
	Query	38.8	168
	Query-total	96.9	168

The total time of Bifrost and Blight is split into index and query as reported by the software, whereas query-total is the wall time measurement. For Mantis, the index is a separate process and needs only to be run once

database Enterobase [58] as of August 2018. This is a 7.3× increase in the number of colors compared to the work of [41] who reported the ccdBG construction for 16,000 *Salmonella* strains. We compared Bifrost to VARI-merge [41] as both tools can construct the colored de Bruijn graph and update it without reconstructing the graph entirely. The main differences between the two tools is that VARI-merge is mainly a disk-based method that produces a non-compacted colored de Bruijn graph. We only benchmarked VARI-merge as it is currently the state-of-the-art for colored de Bruijn graph construction. A comparison of VARI-merge to other colored de Bruijn graph construction tools is given in [41]. Results are given in Table 5 for a variable number of strains. Note that the reported VARI-merge time includes the time spent by KMC2 [59] to compute the k-mers required in input of VARI-merge.

In [41], the authors process 16,000 strains in batches of 4000, merging the batches to produce a colored de Bruijn graph of all strains. This required 254 GB of memory and 2.34 TB of external disk, with a total running time of 69 h. In comparison, Bifrost processed 117,913 strains using about 103 GB of memory, no external disk usage and a total running time of 93.35 h. While the running time is not directly comparable across different machines due to different processors, this is in line with Bifrost being about

Table 4 Running time and fraction of queries found for different k-mer inclusion rates (θ) using exact and inexact k-mers

Query type	θ	Time (m)	Queries found (%)
Exact k-mers	0.50	2.8	99.0
	0.75	3.8	96.0
	0.90	4.4	93.9
	1.00	4.7	92.2
Inexact k-mers	0.50	7.2	99.6
	0.75	14.8	99.0
	0.90	17.7	98.1
	1.00	21.2	97.3

Inexact k-mers allow for one substitution or indel in the k-mer search

Table 5 Running time, memory usage, and external disk usage for constructing the colored de Bruijn graphs of an increasing number of *Salmonella* strains

Number of strains	Tool	Time (h)	Memory (GB)	Disk (GB)
100	Bifrost	**0.016**	**0.16**	**0**
	VARI-merge + KMC2	0.33	5.1	17
400	Bifrost	**0.05**	**0.29**	**0**
	VARI-merge + KMC2	1.016	15.4	51
1600	Bifrost	**0.38**	**2.4**	**0**
	VARI-merge + KMC2	4.86	56.9	228
4000	Bifrost	**1.66**	**3.7**	**0**
	VARI-merge + KMC2	12.35	138	449
117,913	Bifrost	**93.35**	**102.74**	**0**
	VARI-merge + KMC2	N/A	N/A	N/A

N/A indicates the result is unavailable

eight times faster than VARI-merge. The graph built from the 117,913 strains contains 413,658,482 *k*-mers: 39.19% of the *k*-mers have only one color (*singleton*), less than 0.01% of the *k*-mers have all the colors (*core*), and 60.80% of the *k*-mers have more than one but not all colors (*dispensable*). Among the 26,324,369 unitigs, 98.72 % have a single set of colors shared by all their *k*-mers.

Discussion

The de Bruijn graph has been widely used as a fundamental data structure in assemblers, but the memory requirements and focus on speed mean that the implementation has been tightly integrated into the project. Bifrost allows for the integration of the de Bruijn graph as a data structure into projects that work with short read sequencing datasets or assemblies of several genomes. Reusing assemblers can often lead to suboptimal results, e.g., genome assemblers often have coverage assumptions that are not valid for transcriptome assembly. By making minimal assumptions about the input, Bifrost enables researchers to extend our work rather than having to reimplement it.

Conclusion

We present Bifrost, a method for constructing, indexing, and querying compacted de Bruijn graphs, both regular and colored, with minimal computational requirements. Bifrost is competitive with the state-of-the-art de Bruijn graph construction method BCALM2 and the unitig indexing tool Blight with the advantage that Bifrost is dynamic. For colored de Bruijn graphs, Bifrost is about eight times faster than VARI-merge and uses about 20 times less memory with no external disk. The query capabilities of Bifrost are for both identifying colors for a given *k*-mer and navigating the de Bruijn graph. The software was developed with the intention of being usable as a tool or a library wherever large de Bruijn graphs are needed with minimal external dependencies.

Methods

"Definitions" section details the concepts and data structures that will be used throughout this paper. "Approximating the de Bruijn graph" section describes how an approximation of the uncompacted de Bruijn graph is built from a set of sequencing reads.

"Constructing the compacted de Bruijn graph" section shows how the approximate compacted de Bruijn graph is built from its uncompacted counterpart and subsequently converted to an exact compacted de Bruijn graph. "Coloring" section presents how the graph coloring is built efficiently on top of the compacted de Bruijn graph.

Definitions

A string s is a sequence of symbols drawn from an alphabet $\mathcal{A}$. The length of s is denoted by $|s|$. A substring of s is a string occurring in s: it has a starting position i and a length l and is denoted by $s(i,l)$. A substring of length l is also denoted an l-mer. In the following, we assume $\mathcal{A}$ is the DNA alphabet $\mathcal{A} = \{A, C, G, T\}$ for which symbols have complements: (A, T) and (C, G) are the complementing pairs. The reverse-complemented string $\bar{s}$ is the reverse sequence of complemented symbols in s. The canonical string $\hat{s}$ is the lexicographically smallest of s and its reverse-complement $\bar{s}$. The minimizer [60, 61] of an l-mer x is a g-mer y occurring in x such that $g < l$ and y is the lexicographically smallest of all the g-mers in x. The lexicographical order can be cumbersome to use since poly-A g-mers naturally occur in sequencing data and is often replaced by a random order. The simplest way to obtain a random order is to compute a hash-value for each g-mer in x and select the g-mer with the smallest hash-value as the minimizer. In this work, we will only consider minimizers generated by random orderings.

A de Bruijn graph (dBG) is a directed graph $G = (V, E)$ in which each vertex $v \in V$ represents a k-mer. A directed edge $e \in E$ from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(2, k-1) = x'(1, k-1)$. Each k-mer x has $|\mathcal{A}|$ possible successors $x(2, k-1) \odot a$ and $|\mathcal{A}|$ possible predecessors $a \odot x(1, k-1)$ in G with $a \in \mathcal{A}$ and $\odot$ as the concatenation operator. Note that in the original combinatorial definition of the dBG, all possible k-mers for an alphabet $\mathcal{A}$ are present in the graph, whereas in computational biology, the definition is restricted to a subset of the de Bruijn graph representing the k-mers in the input. A path in the graph is a sequence of distinct and connected vertices $p = (v_1, ..., v_m)$. We say that the path p is *non-branching* if all its vertices have an in- and out-degree of one with exception of the head vertex v_1 which can have more than one incoming edge and the tail vertex v_m which can have more than one outgoing edge. A non-branching path is maximal if it cannot be extended in the graph without being branching. A compacted de Bruijn graph (cdBG) merges all maximal non-branching paths of η vertices from the dBG into single vertices, called unitigs, representing words of length $k + \eta - 1$. Minimal examples of dBG and cdBG are provided in Fig. 1a and b respectively. A colored de Bruijn graph is a graph $G = (V, E, C)$ in which (V, E) is a dBG and C is a set of colors such that each vertex $v \in V$ maps to a subset of C; we extend the definition of a cdBG to a colored compacted de Bruijn Graph (ccdBG) to be a graph $G = (V, E, C)$, where (V, E) is a cdBG, so the vertices represent unitigs, and each k-mer of a unitig maps to a subset of C.

Introduced by [62], the Bloom filter (BF) is a space- and time-efficient data structure that records the approximate membership of elements in a set. The BF is represented as a bitmap B of m bits initialized with 0s, coupled with a set of f hash functions $h_1, ..., h_f$. Inserting and querying an element e into B is performed with the functions

$$\mathsf{Insert}(e, B) : B[h_i(e)] \leftarrow 1 \text{ for all} i = 1, ..., f$$

and

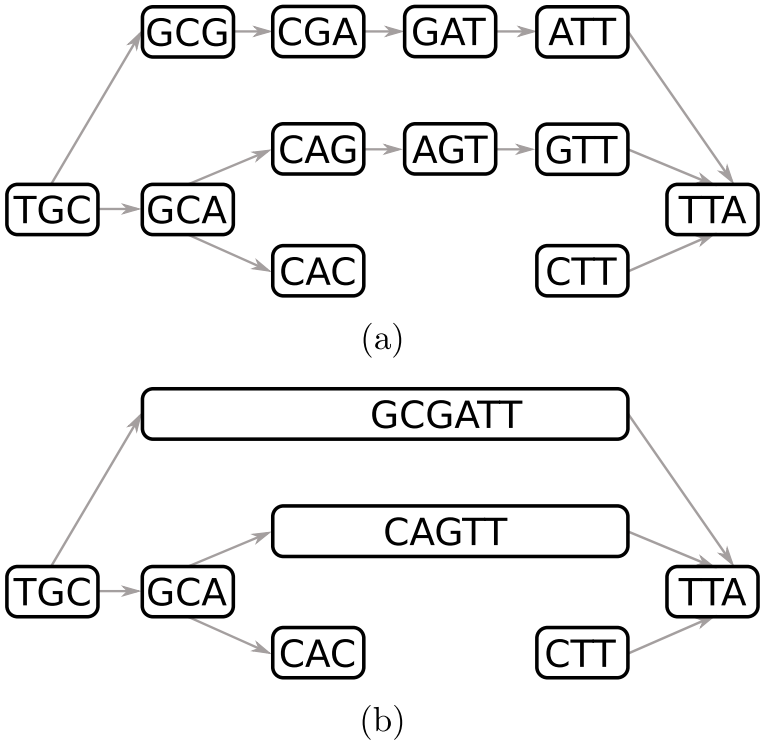

Fig. 1 A de Bruijn graph in **a** and its compacted counterpart in **b** using 3-mers. For simplicity, reverse-complements are not considered

$$\mathsf{MayContain}(e, B) : \bigwedge_{i=1}^{f} B[\, h_i(e)]\,,$$

respectively, in which $\bigwedge$ is the logical conjunction operator. Those functions require $\mathcal{O}(1)$ time. The function MayContain may report false positives when querying for elements which were never inserted but are present in B as a result of independent insertions. Given n elements to insert, the optimal number of hash functions to use [63] is $f = \frac{m}{n} \ln(2)$, for an approximate false positive rate of

$$\varphi \approx \left(1 - e^{\frac{-fn}{m}}\right)^f \approx 0.7^{\frac{m}{n}}$$

Hence, the BF trades off memory usage and time complexity with a decreased false positive rate.

In order to accelerate BFs, [63] demonstrated that two hash functions combined in a double hashing technique can be applied in order to simulate more than two hash functions and obtain similar hashing performance. One main drawback of BFs is their poor data locality as bits corresponding to one element are scattered over B, resulting in several CPU cache misses when inserting and querying. This issue was addressed in [64], which presented the Blocked Bloom Filter (BBF), an array of smaller BFs individually fitting into one or multiple cache lines. To insert or look-up an element, a supplementary hash function is used to determine which BF to load. While BBFs are fast, their false positive ratios are usually higher than regular BFs due to the unbalanced load of each BF in the array.

As minimizers are used extensively throughout Bifrost, we use an efficient rolling hash function based on the work of [65] to select a g-mer as the minimizer within a single k-mer. Since overlapping k-mers are likely to share minimizers, we use an ascending minima approach [66] to recompute minimizers with amortized $O(1)$ costs, so that iterating over minimizers of adjacent k-mers in a sequence is linear in the length of the sequence. Another optimization is to restrict the computation of minimizers to a subset of g-mers of a k-mer, namely, we exclude the first and last g-mer as a candidate for being a minimizer. This ensures that for a given k-mer, all of its forward, respectively backward, adjacent k-mers necessarily share the same minimizer. While it is likely that a k-mer x and its neighbor x' share a minimizer, this neighbor hashing trick [38] guarantees that when searching all forward, respectively backward, neighbors of x, they will all have the same minimizer and will be stored within the same block of a BBF, thus minimizing cache misses.

Approximating the de Bruijn graph

The k-mers extracted from the reads will be inserted into two BBFs: BBF_1 will contain all k-mers occurring at least once in the input read sets while BBF_2 will contain all k-mers occurring twice or more often. This separation allows us to filter out unique k-mers which are likely to be sequencing errors [67]. Algorithm 1 starts by iterating over the reads and extracts all the canonical k-mers. BBF_1 is queried for the presence of each such k-mer, and k-mers already present in BBF_1 are inserted into BBF_2. Finally, BBF_1 is discarded as the cdBG will be built from the k-mers of BBF_2.

Algorithm 1 Construct Blocked Bloom Filters

Input: Read set F

```
1: function FILTER(F)
2:     BBF_1, BBF_2 ← empty Blocked Bloom filters
3:     for each read r ∈ F do
4:         for each canonical k-mer x ∈ r do
5:             b ← MayContain(x, BBF_1)
6:             if b is true then Insert(x, BBF_2)
7:             else Insert(x, BBF_1)
8:     return BBF_2
```

In order to accelerate the insertions into the BBFs, the minimizer hash-value of each k-mer is used to determine the BBF block in which the k-mer is inserted. This guarantees that overlapping k-mers sharing the same minimizer position within a read are inserted into the same BBF block, thus improving the cache efficiency of BBFs. Furthermore, the neighbor hashing of the minimizers guarantees that all predecessors and successors of a k-mer are hashing to the same block, thus improving graph traversal for the exact cdBG construction step. Finally, the BBFs in Bifrost use 2-choice hashing [68] to balance the number of insertions per block and reduce the number of false positives. Instead of selecting a single BBF block when inserting a k-mer, two blocks are selected. If none of the two blocks already contains the k-mer, it is inserted into the block which has the fewest number of bits set. To enable parallel insertions, each BBF block is equipped with

a spinlock to avoid multiple threads inserting at the same time within the same block. Algorithm 2 refines the insertion function introduced in the "Definitions" section to enable 2-choice hashing and spinlocks usage with BBFs. Bifrost can make use of modern processors instruction sets to query simultaneously up to 16 bits within a block using AVX instructions.

Algorithm 2 Insert a k-mer into a BBF

Input: k-mer x, Blocked Bloom Filter BBF

```
1:  function INSERT(x, BBF)
2:      y ← x.getMinimizer()                          ▷ Minimizer of x
3:      b ← |BBF|                                     ▷ Number of blocks
4:      b_1 ← BBF.h_1(y) mod b                        ▷ First block ID
5:      b_2 ← BBF.h_2(y) mod b                        ▷ Second block ID
6:      if b_2 < b_1 then swap(b_1, b_2)              ▷ Avoid deadlock
7:      BBF[b_1].lock()                               ▷ Lock the first block
8:      if MayContain(x, BBF[b_1]) is false then
9:          BBF[b_2].lock()                           ▷ Lock the second block
10:         if MayContain(x, BBF[b_2]) is false then
11:             w_1 ← HammingWeight(BBF[b_1])
12:             w_2 ← HammingWeight(BBF[b_2])
13:             if w_1 < w_2 then Insert(x, BBF[b_1])
14:             else Insert(x, BBF[b_2])
15:         BBF[b_2].unlock()                         ▷ Unlock the second block
16:     BBF[b_1].unlock()                             ▷ Unlock the first block
```

Constructing the compacted de Bruijn graph

The following section describes the data structure indexing the unitigs. The "Unitig extraction" section details the unitig extraction procedure from the BBF and the insertion of unitigs into the cdBG data structure.

Data structure

The cdBG data structure $D = (U, M)$ is composed of a unitig array U and a hash table of minimizers M. A unitig u is first inserted into U and gets a unique identifier id_u. Unitig u is then decomposed into its set of constituent k-mers from which minimizers are extracted. Each minimizer is identified by a position p_m in u. While there can be as many minimizer positions as there are k-mers in the unitig, it is likely that multiple overlapping k-mers share the same minimizer position. The canonical g-mers corresponding to the minimizers are inserted into M and associated with their position p_m in u and the identifier id_u. Note that a minimizer might have multiple occurrences, either within a unitig or in different unitigs of the graph. The cdBG data structure D is illustrated in Fig. 2. Algorithm 3 details the insertion of a unitig u in the cdBG data structure. Note that removing a unitig from the graph can be done in a reversed-fashion to Algorithm 3: The tuples associated with unitig u are removed from M and unitig u is removed from U.

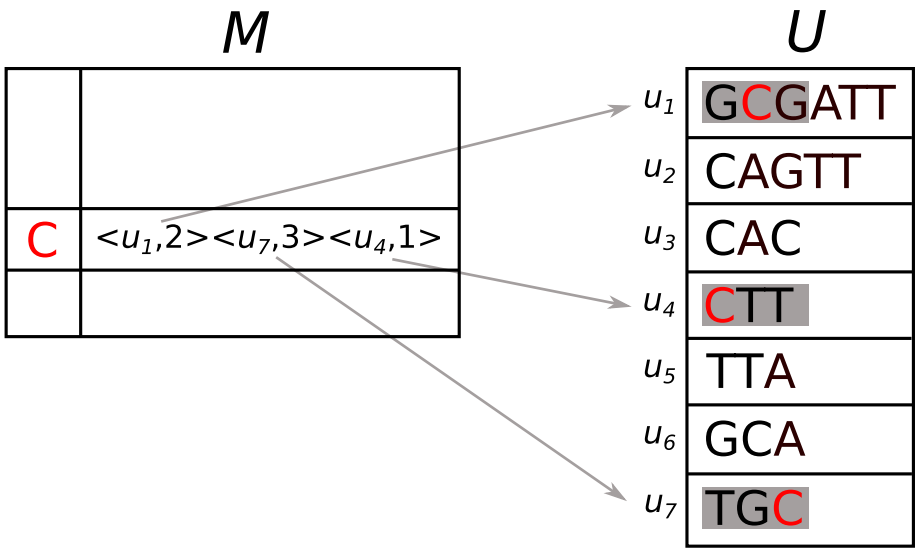

Fig. 2 Data structure of a cdBG composed of a hash table M and a unitig array U. Unitigs are composed of 3-mers and are indexed using minimizers of length 1. For simplicity, a lexicographic ordering of minimizers is here used and only one minimizer is shown

Algorithm 3 Insert a Unitig into a cdBG

Input: Unitig u, cdBG data structure D

1: **function** INSERT($u, D = (U, M)$)
2: $id_u \leftarrow Insert(u, U)$ ▷ Unitig u is inserted and gets an identifier
3: $p'_m \leftarrow -1$ ▷ Initialize position of previous minimizer
4: $i \leftarrow 1$
5: **while** $i \leq |u| - k + 1$ **do** ▷ Iterate over k-mer positions of u
6: $x \leftarrow u(i, k)$ ▷ k-mer at position i in u
7: **for each** minimizer positions $p_m \in x$ **do**
8: **if** $p_m + i > p'_m$ **then** ▷ New minimizer
9: $y \leftarrow x(p_m, g)$ ▷ Minimizer of x
10: $\mathcal{T} \leftarrow \mathsf{Find}(\hat{y}, M)$ ▷ Find occurrences of $\hat{y}$ in U
11: **if** $\mathcal{T} = \varnothing$ **then** $\mathsf{Insert}(\{\hat{y}, \langle id_u, p_m + i \rangle\}, M)$
12: **else** $\mathcal{T} \leftarrow \mathcal{T} \cup \langle id_u, p_m + i \rangle$
13: $p'_m = p_m + i$
14: $i \leftarrow i + 1$

Looking-up a k-mer x in the cdBG data structure is similar to inserting a unitig. The canonical g-mer corresponding to the minimizer of x is extracted and used to query M. If the g-mer is not in M, x does not occur in a unitig of the cdBG. However, if the g-mer is present, the identifiers of the unitigs containing the g-mer and the g-mer positions within those unitigs are returned. K-mer x and its reverse-complement $\overline{x}$ are then anchored in those unitigs at the given minimizer positions and compared. If the comparison is positive, a tuple with the unitig identifier and the k-mer position in the unitig is returned. Algorithm 4 shows how to look-up D for a k-mer.

Algorithm 4 Find a k-mer in a Unitig in a cdBG

Input: k-mer x, cdBG data structure D

1: **function** FIND($x, D = (U, M)$)
2: **for each** minimizer positions $p_m \in x$ **do**
3: $y \leftarrow x(p_m, g)$
4: $\mathcal{T} \leftarrow \mathsf{Find}(\hat{y}, M)$
5: **if** $\mathcal{T} \neq \varnothing$ **then**
6: **for each** tuple $\langle p'_m, id \rangle \in \mathcal{T}$ **do**
7: $u \leftarrow U[\,id]$
8: $p_k \leftarrow p'_m - p_m + 1$ ▷ Possible position of x on u
9: **if** $1 \leq p_k \leq |u| - k + 1$**and**$u(p_k, k) = x$ **then**
10: **return** $\langle p_k, id \rangle$
11: $p_k \leftarrow p'_m - k + g + p_m$ ▷ Possible position of $\overline{x}$ on u
12: **if** $1 \leq p_k \leq |u| - k + 1$**and**$u(p_k, k) = \overline{x}$ **then**
13: **return** $\langle p_k, id \rangle$
14: **return** $\langle -1, -1 \rangle$ ▷ No matches found

Unitig extraction

The BBF returned by Algorithm 1 represents an approximation of the dBG: It contains the true positive k-mers, namely all the k-mers present in the unitigs of the cdBG, but also false positive k-mers, which do not belong to the cdBG. The false positive k-mers are either artifacts of BBF_2 or single occurrence k-mers that should have been filtered out by Algorithm 1 but were inserted into BBF_2 as a result of their false occurrences in BBF_1. Although BBFs are efficient data structures, they do not allow to iterate over the contents. To get around this limitation, we iterate over the original set of reads and query BBF_2 to identify k-mers that are present.

Given a k-mer x, Algorithm 5 extracts from the BBF the unitig from which x is a substring, conditioned upon the presence of x in the BBF. K-mer x is extended forward, respectively backward, by reconstructing iteratively the prefix, respectively suffix, of the unitig using function Extend. Note that a backward extension is performed by extending forward from the reverse-complement of x and the extracted suffix is reverse-complemented to obtain the unitig prefix. Forward extensions are made with function ExtendForward which iteratively concatenate the last character from the next k-mer in the extension until no more k-mer is found or the extracted k-mer creates a cycle. Finally, k-mer x is extended with x' using function ExtendKmer if the two k-mers belong to the same maximal non-branching path, i.e, if x' is the only successor of x in the BBF and x is the only predecessor of x' in the BBF,

Given the read set, the BBF containing the filtered k-mers, and an empty cdBG data structure, Algorithm 6 extracts the unitigs from the BBF and inserts them into the cdBG data structure. The algorithm iterates over the k-mers of the reads and queries the BBF for their presence. A missing k-mer in the BBF indicates the k-mer was filtered out by Algorithm 1 and will not be part of a unitig, in which case the next k-mer in the read is queried. However, in case of the k-mer presence in the BBF, the cdBG is searched for the unitig containing this k-mer using Algorithm 4. If the k-mer is missing from the unitigs present in the cdBG data structure, it means its unitig has not been extracted yet from

Algorithm 5 Unitig extraction from a BBF

Input: k-mer x, Blocked Bloom Filter BBF

function EXTEND(x, BBF)
  $s_f \leftarrow$ ExtendForward(x, BBF) ▷ Forward extension
  $s_b \leftarrow$ ExtendForward($\overline{x}, BBF$) ▷ Backward extension
  $s \leftarrow \overline{s_b} \odot x \odot s_f$ ▷ Unitig
  return s

Input: k-mer x, Blocked Bloom Filter BBF

function EXTENDFORWARD(x, BBF)
  $s \leftarrow \varepsilon$ ▷ String for extension
  $x_e \leftarrow x$ ▷ Previous extended k-mer
  $x'_e \leftarrow$ ExtendKmer(x_e, BBF) ▷ New extended k-mer
  while $x'_e \neq \varepsilon$ **do** ▷ While extending is possible
    if $x'_e = x$ **then** $x'_e \leftarrow \varepsilon$ ▷ Cycle
    else if $x'_e = x_e$ **then** $x'_e \leftarrow \varepsilon$ ▷ Self-loop k-mer
    else if $x'_e = \overline{x_e}$ **then** $x'_e \leftarrow \varepsilon$ ▷ Self-loop k-mer
    else
      $s \leftarrow s \odot x'_e(k, 1)$ ▷ Extend string
      $x_e \leftarrow x'_e$
      $x'_e \leftarrow$ ExtendKmer(x_e, BBF) ▷ Extend previous k-mer
  return s_f

Input: k-mer x, Blocked Bloom Filter BBF

function EXTENDKMER(x, BBF)
  $i \leftarrow 0$
  $x' \leftarrow x(2, k-1)$ ▷ Prefix of all successors of x
  $x_e \leftarrow \varepsilon$ ▷ Extended k-mer
  for each $a \in \mathcal{A}$ **do**
    if MayContain($x' \odot a$) is true **then** ▷ BBF has successor of x
      $i \leftarrow i + 1$ ▷ Increment count of successors
      $x_e \leftarrow x' \odot a$ ▷ Save last successor found
  if $i = 1$ **then** ▷ If x has only one successor
    $i \leftarrow 0$
    $x' \leftarrow x_e(1, k-1)$ ▷ Suffix of all predecessors of x_e
    for each $a \in \mathcal{A}$ **do**
      if MayContain($a \odot x'$) is true **then** $i \leftarrow i + 1$
  if $i \neq 1$ **then** $x_e \leftarrow \varepsilon$
  return x_e

the BBF. The extraction using Algorithm 5 takes place, and the extracted unitig is inserted into the cdBG data structure with Algorithm 3.

Eliminating the false positive k-mers

The cdBG constructed by Algorithm 6 is not exact as it contains false positive k-mers of BBF_2. Those false positive k-mers create two types of errors in the graph:

Algorithm 6 Initial cdBG Construction

Input: Read set F, Blocked Bloom Filter BBF, cdBG data structure D

```
1:  function INSERT(F, BBF, D = (U, M))
2:      for each read r ∈ F do
3:          i ← 1
4:          while i ≤ |r| − k + 1 do                    ▷ For all k-mer positions in r
5:              x ← r(i, k)                              ▷ Get k-mer
6:              if MayContain(x, BBF) is true then       ▷ If x in BBF
7:                  t ← Find(x, D)                       ▷ Search unitig associated to x
8:                  if t = ⟨−1, −1⟩ then                 ▷ If unitig not found
9:                      u ← Extend(x, BBF)               ▷ Extract unitig from BBF
10:                     Insert(u, D)                     ▷ Insert unitig in data structure
11:                     t ← Find(x, D)
12:                 p, id ← t
13:                 r_s ← r(i + k, |r| − i − k)           ▷ Suffix of r starting at x
14:                 u_s ← u(p + k, |u| − p − k)           ▷ Suffix of u starting at x
15:                 l ← Lcp(r_s, u_s)                    ▷ Longest Common Prefix of u_s and r_s
16:                 i ← i + |l|                          ▷ Move iterator forward of |l| positions
17:             else i ← i + 1
```

- False connection: A false positive k-mer connects a unitig with no successors to a unitig with no predecessors. Hence, one unitig is extracted from the BBF instead of two.
- False branching: A false positive k-mer connects as a successor, respectively predecessor, to a true positive k-mer which already has a successor, respectively predecessor. Hence, three unitigs are extracted from the BBF instead of one.

An example of a cdBG containing the two types of errors is illustrated in Fig. 3: K-mer "CCG" creates a false branching and "ACT" creates a false connection.

In order to distinguish false positive from true positive k-mers, a counter is maintained on each k-mer of the unitigs and Algorithm 6 is modified to increment the counters of the k-mers occurring in the reads. Hence, false positive k-mers with no or one single occurrence are deleted from the graph. In the case of a false connection k-mer, deleting

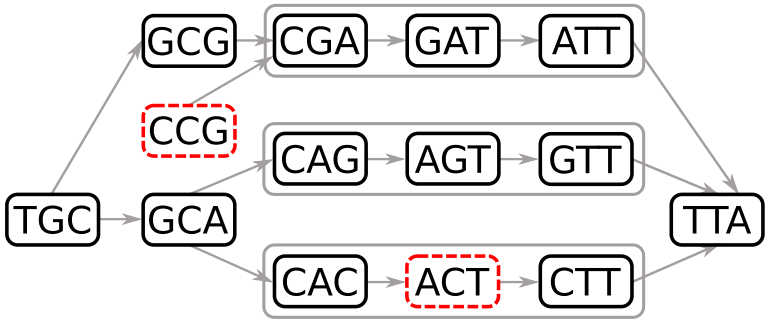

Fig. 3 A compacted de Bruijn graph containing false positive 3-mers. Errors are represented in red dashed line vertices: *K*-mer "CCG" creates a false branching and "ACT" creates a false connection. *K*-mers that are compacted in a unitig are grouped in a gray line box

the k-mer splits a unitig. In case of a false branching, deleting the k-mer joins one or multiple unitigs.

Algorithm 7 Removal of False Positives

Input: cdBG data structure D

function REMOVEFP($D = (U, M)$)
  $\mathcal{X} \leftarrow \varnothing$
  for each unitig $u \in U$ **do**
    $i \leftarrow 1$
    while $i \leq |u| - k + 1$ **do** ▷ For all k-mer positions in u
      if getCounter$(i, u) < 2$ **then** ▷ k-mer is a false positive
        $\mathcal{X} \leftarrow \mathcal{X} \cup u(i, k)$
      $i \leftarrow i + 1$
  for each k-mer $x \in \mathcal{X}$ **do** ▷ For all false positive k-mers
    $p, id \leftarrow$ Find(x, D) ▷ Find unitig containing FP
    $u \leftarrow U[\, id]$ ▷ Unitig containing FP
    Remove(u, D) ▷ Remove u from graph
    if $p = 1$ **then** Join$(\overline{u(1, k)}, D)$ ▷ Fix false connection
    else Insert$(u(1, p + k - 2), D)$ ▷ Insert prefix of u
    if $p = |u| - k + 1$ **then** Join$(u(|u| - k + 1, k), D)$
    else Insert$(u(p + 1, |u| - p), D)$ ▷ Insert suffix of u

Input: k-mer x, cdBG data structure D

function JOIN($x, D = (U, G)$)
  for each $a \in \mathcal{A}$ **do**
    $x_s \leftarrow x(2, k - 1) \odot a$ ▷ Possible successor of x
    $p_s, id_s \leftarrow$ Find(x_s, D) ▷ Find possible successor
    if $p_s \neq -1$ **then** ▷ Successor x_s is found
      $x_p \leftarrow$ ExtendKmer$(\overline{x_s}, D)$
      if $x_p \neq \varepsilon$ **then** ▷ Unitigs of x_s and x_p can be joined
        $p_p, id_p \leftarrow$ Find(x_p, D)
        $u_p \leftarrow U[\, id_p]$ ▷ Unitigs of x_p
        $u_s \leftarrow U[\, id_s]$ ▷ Unitigs of x_s
        if $x_p \neq u(|u| - k + 1, k)$ **then** $u_p \leftarrow \overline{u_p}$
        if $x_s \neq u_s(1, k)$ **then** $u_s \leftarrow \overline{u_s}$
        $u \leftarrow u_p \odot u_s(k, |u_s| - k + 1)$
        Remove(u_p, D) ▷ Remove u_p
        Remove(u_s, D) ▷ Remove u_s
        Insert(u, D) ▷ Insert joined unitig u

Ghost k-mers

The false positive rate of the BBF will affect the length of the unitigs extracted by Algorithm 5. Consider a unitig of length $k + \eta - 1$ in the true cdBG, consisting of η k-mers. For each internal k-mer, the algorithm makes 8 queries to the BBF, two of which

will return true and 6 of which should return false. If the BBF has a false positive rate of p, the algorithm will advance to the next k-mer with probability $(1-p)^6 \approx 1-6p$ and stop prematurely with probability $\approx 6p$. The number of k-mers in the extracted unitig will then be limited by η on one hand and a geometric distribution with probability $6p$, whose expected value is $\frac{1}{6p}$. When $p = 10^{-3}$, this would lead to an average unitig length of 167. While these errors are fixed with Algorithm 7, this leads to an increased memory usage. One way to increase the length would be to use more memory in the BBF which would reduce the false positive rate. However, we observe that the most likely configuration is that a single false positive k-mer x', adjacent to a real k-mer x in the unitig, causes a premature halt to the extraction of the true unitig. When x' has no other neighbor in the BBF except for x, we call it a ghost k-mer, insert it into a hash table to keep track of it in case we observe it later but do not stop the extraction of the unitig. In the rare case that x' turns out to belong to the true cdBG, we identify the unitig containing x' and fix the mistake. The probability that we halt can now be approximated as $42p^2$, since this would require two adjacent false positive k-mers to occur in the BBF. The use of ghost k-mers greatly reduces fragmentation which improves memory usage and running time.

Recurrent minimizers

Even in the case of a minimizer random ordering as described in the "Definitions" section, some minimizers are expected to occur more often in unitigs than others, due to indels occurring in homopolymer and tandem repeat sequences. Those minimizers are likely to increase the running time as their lists of tuples in the minimizer hash table M will be much longer than for the other minimizers. We define a minimizer as *recurrent* if it occurs t times or more in the unitigs of the cdBG. In order to limit the impact of recurrent minimizers on the graph construction, lists of tuples in M have a maximum length t. When a k-mer x and its corresponding minimizer y must be inserted into the cdBG data structure, the length of the list associated with y in M is verified first. If the length is greater or equals to t, y is a recurrent minimizer. In such case, a non-recurrent minimizer $y' > y$ is extracted from x and inserted into M. If x does not contain a non-recurrent minimizer y', the recurrent minimizer y is inserted into M instead. Whenever k-mer x is searched, the list of tuples associated with its minimizer y is traversed and x is anchored on the instances of y in the unitigs of the graph until a match is found, as described in Algorithm 4. However, if no match is found for x and the list of tuples associated with y contains t or more tuples, the non-recurrent minimizer y' is extracted from x and the search continues using minimizer y'.

Coloring

We denote as D' the data structure of a ccdBG: It is composed of a unitig array U, a minimizer hash table M, an array O of color containers, an array H of hash functions, and a hash table K of k-mers.

Container representation

In Bifrost, a color is represented by an integer from 1 to $|C|$. A unitig u composed of $\eta = |u| - k + 1$ k-mers is associated with a binary matrix of size $\eta \times |C|$: rows represent the different k-mer positions in u and columns represent the colors from C. A bit set at row $1 \leq i \leq \eta$ and column $1 \leq j \leq |C|$ indicates that k-mer $u(i,k)$ occurs in dataset j. In

order to limit the memory usage of colors, multiple compressed index is used to represent these binary matrices depending on their sparsity:

- A 64-bit word that can be either a tuple $\langle \text{position}i, \text{color}j \rangle$ or a binary matrix of size $\eta \times |C| \leq 62$ (2 bits are reserved for the meta-data)
- A compressed bitmap adapted from a Roaring bitmap container [69]. This compressed bitmap stores up to 65488 tuples $\langle \text{position}i, \text{color}j \rangle$ and uses a maximum of 8 KB of memory. This container has 3 representations of the tuples it indexes: bit vector, sorted list of tuples, and run-length encoded list of sorted tuples. Compared to a Roaring bitmap, this compressed bitmap uses less memory for its meta-data and incurs fewer cache misses to access the tuples.
- A Roaring bitmap [69] to store more than 65488 tuples. Roaring bitmaps are SIMD accelerated and propose numerous functions to manipulate bitmaps such as set intersection and union.

Those representations have a logarithmic worst-case time look-up and insertion.

Algorithm 8 ccdBG Construction

Input: ccdBG data structure D', set of hash functions $\mathcal{H}$

function AssociateColors($D' = (U, M, O, H, K), \mathcal{H}$)
  $B \leftarrow$ binary array of length$|U|$ initialized with 0s
  $O \leftarrow$ array of empty color containers of length$|U|$
  $h_e \leftarrow$ empty hash function$f : A \rightarrow \varnothing$
  $H \leftarrow$ array of hash functions of length$|U|$ initialized with h_e
  $id_u \leftarrow 1$
  $i_B \leftarrow 1$
  while $id_u \leq |U|$ **do** ▷ For each unitig
    $x \leftarrow U[\, id_u] \, (1, k)$ ▷ First k-mer of unitig
    for each hash function $h \in \mathcal{H}$ **do**
      $v \leftarrow h(x) \mod |U|$
      if $B[\, v] = 0$ **then** ▷ Color container in $C[\, v]$ is free
        $B[\, v] \leftarrow 1$ ▷ Color container in $C[\, v]$ is linked to id_u
        $H[\, id_u] \leftarrow h$ ▷ Unitig id_u must be hashed with h
        break
    if $H[\, id_u] = h_e$ **then** ▷ No color container was free for id_u
      while $i_B \leq |U|$ **do** ▷ Search next free color container
        if $B[\, i_B] = 0$ **then**
          break ▷ Color container at pos. i_B is free
        $i_B \leftarrow i_B + 1$
      $B[\, i_B] = 1$ ▷ Color container at pos. i_B is reserved
      Insert$(\{x, i_B\}, K)$
    $id_u \leftarrow id_u + 1$

Container indexing

Color containers can become substantially large, and in order to avoid costly data transfer operations when the ccdBG data structure D' is modified, color containers are not associ-

ated directly to unitigs in D'. Instead, a solution derived from the MPHF (Minimal Perfect Hash Function) library BBHash [70] is used to link unitigs of array U to color containers of array O. The benefit of such a method is that operations which affect only the structure of the graph do not move the color containers in memory. Algorithm 8 describes how color containers are associated to their respective unitigs.

Supplementary information

Supplementary information accompanies this paper at https://doi.org/10.1186/s13059-020-02135-8.

Additional file 1: Supplementary file.

Additional file 2: Review history.

Acknowledgements
The authors would like to thank Nina Luhmann, Birte Kehr, and Thomas Krannich for their helpful feedback during the development of the software and Trausti Sæmundsson for his work on an early draft of the software.

Review history
The review history is available as Additional file 2.

Peer review information
Andrew Cosgrove was the primary editor of this article and managed its editorial process and peer review in collaboration with the rest of the editorial team.

Funding
This work was supported by the Icelandic Research Fund Project grant number 152399-053.

Authors' contributions
All authors implemented the Bifrost software and designed the algorithm and the experiments. All authors wrote the manuscript. All authors reviewed and approved the final version of the manuscript.

Availability of data and materials
We have made the source code of Bifrost available as open source software at https://github.com/pmelsted/bifrost [71]. The source code is released under a BSD-2 license. The website contains details on installation, setup, and usage. The exact version used in this paper is archived at Zenodo under https://zenodo.org/record/3973373 [72].

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 13 August 2019 Accepted: 6 August 2020
Published online: 17 September 2020

References

1. Pevzner PA, Tang H, Waterman MS. An Eulerian path approach to DNA fragment assembly. Proc Natl Acad Sci USA. 2001;98(17):9748–53.
2. Idury RM, Waterman MS. A new algorithm for DNA sequence assembly. J Comput Biol. 1995;2(2):291–306.
3. Yang B, Liu B, Mu D, Zhang H, Yuan J, Gan J, Li N, Fan W, Hu X, Chen Y, Shi Y, Li Z. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. Brief Funct Genomics. 2011;11(1):25–37.
4. Chaisson MJ, Pevzner PA. Short read fragment assembly of bacterial genomes. Genome Res. 2008;18(2):324–30.
5. Zerbino DR, Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 2008;18(5):821–9.
6. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol İ. ABySS: a parallel assembler for short read sequence data. Genome Res. 2009;19(6):1117–23.
7. Luo R, Liu B, Xie Y, Li Z, Huang W, Yuan J, He G, Chen Y, Pan Q, Liu Y, Tang J, Wu G, Zhang H, Shi Y, Liu Y, Yu C, Wang B, Lu Y, Han C, Cheung DW, Yiu S-M, Peng S, Xiaoqian Z, Liu G, Liao X, Li Y, Yang H, Wang J, Lam T-W, Wang J. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler,. GigaScience. 2012;1(1):1–18.
8. Chikhi R, Rizk G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms Mol Biol. 2013;8(22).

9. Bankevich A, Nurk S, Antipov D, Gurevich AA, Dvorkin M, Kulikov AS, Lesin VM, Nikolenko SI, Pham S, Prjibelski AD, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J Comput Biol. 2012;19(5):455–77.
10. MacCallum I, Przybylski D, Gnerre S, Burton J, Shlyakhter I, Gnirke A, Malek J, McKernan K, Ranade S, Shea TP, et al. ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. Genome Biol. 2009;10:103.
11. Rang FJ, Kloosterman WP, de Ridder J. From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. Genome Biol. 2018;19(1):90.
12. Rhoads A, Au KF. PacBio sequencing and its applications. Genomics Proteome Bioinforma. 2015;13(5):278–89.
13. Koren S, Walenz BP, Berlin K, Miller JR, Bergman NH, Phillippy AM. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Res. 2017;27(5):722–36.
14. Li H. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. Bioinformatics. 2016;32(14):2103–10.
15. Chin C-S, Peluso P, Sedlazeck FJ, Nattestad M, Concepcion GT, Clum A, Dunn C, O'Malley R, Figueroa-Balderas R, Morales-Cruz A, et al. Phased diploid genome assembly with single-molecule real-time sequencing. Nat Methods. 2016;13(12):1050.
16. Kamath GM, Shomorony I, Xia F, Courtade TA, David NT. Hinge: long-read assembly achieves optimal repeat resolution. Genome Res. 2017;27(5):747–56.
17. Salmela L, Rivals E. LoRDEC: accurate and efficient long read error correction. Bioinformatics. 2014;30(24):3506–14.
18. Ruan J, Li H. Fast and accurate long-read assembly with wtdbg2. Nat Methods. 2020;17:155–8.
19. Robertson G, Schein J, Chiu R, Corbett R, Field M, Jackman SD, Mungall K, Lee S, Okada HM, Qian JQ, et al. De novo assembly and analysis of RNA-seq data. Nat Methods. 2010;7:909–12.
20. Uricaru R, Rizk G, Lacroix V, Quillery E, Plantard O, Chikhi R, Lemaitre C, Peterlongo P. Reference-free detection of isolated SNPs. Nucleic Acids Res. 2015;43(2):11.
21. Benoit G, Lemaitre C, Lavenier D, Drezen E, Dayris T, Uricaru R, Rizk G. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. BMC Bioinformatics. 2015;16(1):288.
22. Limasset A, Flot J-F, Peterlongo P. Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. Bioinformatics. 2020;36(5):1374–81.
23. Liu B, Guo H, Brudno M, Wang Y. deBGA: read alignment with de Bruijn graph-based seed and extension. Bioinformatics. 2016;32(21):3224–32.
24. Iqbal Z, Caccamo M, Turner I, Flicek P, McVean G. De novo assembly and genotyping of variants using colored de Bruijn graphs. Nat Genet. 2012;44:226–32.
25. Zekic T, Holley G, Stoye J. Pan-genome storage and analysis techniques. In: Comparative Genomics. Springer; 2018. p. 29–53.
26. Fang H, Bergmann EA, Arora K, Vacic V, Zody MC, Iossifov I, O'Rawe JA, Wu Y, Barron LTJ, Rosenbaum J, et al. Indel variant analysis of short-read sequencing data with Scalpel. Nat Protoc. 2016;11:2529–48.
27. Bray NL, Pimentel H, Melsted P, Pachter L. Near-optimal probabilistic RNA-seq quantification. Nat Biotechnol. 2016;34:525–7.
28. Minkin I, Pham S, Medvedev P. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. Bioinformatics. 2017;33(24):4024–32.
29. Chikhi R, Limasset A, Medvedev P. Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics. 2016;32(12):201–8.
30. Marcus S, Lee H, Schatz MC. SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics. 2014;30(24):3476–83.
31. Baier U, Beller T, Ohlebusch E. Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics. 2016;32(4):497–504.
32. Minkin I, Patel A, Kolmogorov M, Vyahhi N, Pham S. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In: Proc. of the 13th Workshop on Algorithms in Bioinformatics (WABI'13); 2013. p. 215–29.
33. Sheikhizadeh S, Schranz ME, Akdel M, de Ridder D, Smit S. PanTools: representation, storage and exploration of pan-genomic data. Bioinformatics. 2016;32(17):487–93.
34. Drezen E, Rizk G, Chikhi R, Deltel C, Lemaitre C, Peterlongo P, Lavenier D. GATB: genome assembly & analysis tool box. Bioinformatics. 2014;30(20):2959–61.
35. Crusoe MR, Alameldin HF, Awad S, Boucher E, Caldwell A, Cartwright R, Charbonneau A, Constantinides B, Edvenson G, Fay S, et al. The khmer software package: enabling efficient nucleotide sequence analysis. F1000Research. 2015;4:900.
36. Almodaresi F, Pandey P, Patro R. Rainbowfish: a succinct colored de Bruijn graph representation. In: Proc. of the 17th Workshop on Algorithms in Bioinformatics (WABI'17). Schloss Dagstuhl-Leibniz-Zentrum für Informatik; 2017.
37. Holt J, McMillan L. Merging of multi-string BWTs with applications. Bioinformatics. 2014;30(24):3524–31.
38. Holley G, Wittler R, Stoye J. Bloom filter trie–a data structure for pan-genome storage. In: Proc. of the 15th Workshop on Algorithms in Bioinformatics (WABI'15), vol. 9289. Springer; 2015. p. 217–30.
39. Muggli MD, Bowe A, Noyes NR, Morley PS, Belk KE, Raymond R, Gagie T, Puglisi SJ, Boucher C. Succinct colored de Bruijn graphs. Bioinformatics. 2017;33(20):3181–7.
40. Almodaresi F, Pandey P, Ferdman M, Johnson R, Patro R. An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. In: Proc. of the 23rd International Conference on Research in Computational Molecular Biology (RECOMB'19). Springer; 2019.
41. Muggli MD, Alipanahi B, Boucher C. Building large updatable colored de Bruijn graphs via merging. bioRxiv. 2019. https://doi.org/10.1101/229641.
42. Solomon B, Kingsford C. Fast search of thousands of short-read sequencing experiments. Nat Biotechnol. 2016;34:300–2.
43. Sun C, Harris RS, Chikhi R, Medvedev P. Allsome sequence bloom trees. J Comput Biol. 2018;25(5)467–79.

44. Solomon B, Kingsford C. Improved search of large transcriptomic sequencing databases using split sequence Bloom trees. J Comput Biol. 2018;25(7)755–65.
45. Pandey P, Almodaresi F, Bender MA, Ferdman M, Johnson R, Patro R. Mantis: a fast, small, and exact large-scale sequence-search index. Cell Syst. 2018;7(2):201–7.
46. Yu Y, Liu J, Liu X, Zhang Y, Magner E, Lehnert E, Qian C, Liu J. Seqothello: querying RNA-seq experiments at scale. Genome Biol. 2018;19:167.
47. Bradley P, den Bakker HC, Rocha EP, McVean G, Iqbal Z. Ultrafast search of all deposited bacterial and viral genomic data. Nat Biotechnol. 2019;37:152–9.
48. Chikhi R, Holub J, Medvedev P. Data structures to represent sets of k-long DNA sequences. arXiv: 1903.12312. 2019.
49. Marchet C, Boucher C, Puglisi SJ, Medvedev P, Salson M, Chikhi R. Data structures based on k-mers for querying large collections of sequencing datasets. bioRxiv. 2019. https://doi.org/10.1101/866756.
50. Guo H, Fu Y, Gao Y, Li J, Wang Y, Liu B. deGSM: memory scalable construction of large scale de Bruijn Graph. IEEE/ACM Trans Comput Biol Bioinform. 2019. https://doi.org/10.1109/TCBB.2019.2913932.
51. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital SRC Research Report. 1994.
52. Weiner P. Linear pattern matching algorithms. In: Proc. of the 14th Annual Symposium on Switching and Automata Theory (SWAT'73).). IEEE; 1973.
53. Wittler R. Alignment- and reference-free phylogenomics with colored de-Bruijn graphs. In: Proc. of the 19th Workshop on Algorithms in Bioinformatics (WABI'19). Springer; 2019.
54. Luhmann N, Holley G, Achtman M. BlastFrost: fast querying of 100,000s of bacterial genomes in Bifrost graphs. bioRxiv. 2020. https://doi.org/10.1101/2020.01.21.914168.
55. Zook JM, Catoe D, McDaniel J, Vang L, Spies N, Sidow A, Weng Z, Liu Y, Mason CE, Alexander N, et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. Sci Data. 2016;3:160025.
56. Marchet C, Kerbiriou M, Limasset A. Indexing de Bruijn graphs with minimizers. In: Proc. of the 23rd International Conference on Research in Computational Molecular Biology (RECOMB'19); 2019. https://doi.org/10.1101/546309.
57. Pandey P, Bender MA, Johnson R, Patro R. Squeakr: an exact and approximate k-mer counting system. Bioinformatics. 2017;34(4):568–75. https://doi.org/10.1093/bioinformatics/btx636.
58. Zhou Z, Alikhan N-F, Mohamed K, Achtman M. The user's guide to comparative genomics with EnteroBase. Three case studies: micro-clades within Salmonella enterica serovar Agama, ancient and modern populations of Yersinia pestis, and core genomic diversity of all Escherichia. bioRxiv. 2019. https://doi.org/10.1101/613554.
59. Deorowicz S, Kokot M, Grabowski S, Debudaj-Grabysz A. KMC 2: fast and resource-frugal k-mer counting. Bioinformatics. 2015;31(10):1569–76.
60. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. Reducing storage requirements for biological sequence comparison. Bioinformatics. 2004;20(18):3363–9.
61. Grabowski S, Deorowicz S, Roguski L. Disk-based compression of data from genome sequencing. Bioinformatics. 2015;31(9):1389–95.
62. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Comm ACM. 1970;13(7):422–6.
63. Kirsch A, Mitzenmacher M. Less hashing, same performance: building a better Bloom filter. In: Proc. of the European Symposium on Algorithms (ESA'06), vol. 4168; 2006. p. 456–67.
64. Putze F, Sanders P, Singler J. Cache-, hash- and space-efficient bloom filters. ACM J Exp Algorithmic. 2009;14:9.
65. Lemire D, Kaser O. Recursive n-gram hashing is pairwise independent, at best. Comput Speech Lang. 2010;24(4): 698–710.
66. Harter R. The minimum on a sliding window algorithm. 2009. http://richardhartersworld.com/cri/2001/slidingmin.html. Accessed 25 Mar 2019.
67. Melsted P, Pritchard JK. Efficient counting of k-mers in DNA sequences using a bloom filter. BMC Bioinformatics. 2011;12(1):333.
68. Azar Y, Broder AZ, Karlin AR, Upfal E. Balanced allocations. SIAM J Comput. 1999;29(1):180–200.
69. Chambi S, Lemire D, Kaser O, Godin R. Better bitmap performance with Roaring bitmaps. Softw Pract Exp. 2016;46(5):709–19.
70. Limasset A, Rizk G, Chikhi R, Peterlongo P. Fast and scalable minimal perfect hashing for massive key sets. arXiv. 2017.
71. Holley G, Melsted P. Bifrost Github repository. 2020. https://github.com/pmelsted/bifrost.
72. Holley G, Melsted P. Zenodo repository for Bifrost. https://doi.org/10.5281/zenodo.3973373.

Publisher's Note