Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2

Jamshed Khan[1,2], **Marek Kokot**[3], **Sebastian Deorowicz**[3], **and Rob Patro**[1,2,✉]

[1]Department of Computer Science, University of Maryland, College Park, Maryland, USA
[2]Center for Bioinformatics and Computational Biology, University of Maryland, College Park, Maryland, USA
[3]Faculty of Automatic Control, Electronics and Computer Science, Silesian University of Technology, Gliwice, Poland

The de Bruijn graph has become a key data structure in modern computational genomics, and of keen interest is its compacted variant. The compacted de Bruijn graph provides a lossless representation of the graph, and it is often considerably more efficient to store and process than its non-compacted counterpart. Construction of the compacted de Bruijn graph resides upstream of many genomic analyses. As the quantity of sequencing data and the number of reference genomes on which to perform these analyses grow rapidly, efficient construction of the compacted graph becomes a computational bottleneck for these tasks.

We present CUTTLEFISH 2**, significantly advancing the existing state-of-the-art methods for construction of this graph. On a typical shared-memory machine, it reduces the construction of the compacted de Bruijn graph for 661K bacterial genomes (2.58 Tbp of input reference genomes) from about 4.5 days to 17–23 hours. Similarly on sequencing data, it constructs the graph for a 1.52 Tbp white spruce read set in about 10 hours, while the closest competitor, which also uses considerably more memory, requires 54–58 hours.**

Correspondence: *rob@cs.umd.edu*

1. Background

Rapid developments in the throughput and affordability of modern sequencing technologies have made the generation of billions of short-read sequences from a panoply of biological samples highly time- and cost-efficient. The National Center for Biotechnology Information (NCBI) has now moved the Sequence Read Archive (SRA) to the cloud, and this repository stores more than 14 petabytes worth of sequencing data (NCBI Insights). Yet, this is only a fraction of the total sequencing data that has been produced, which is expected to reach exabyte-scale within the current decade (2). In addition to the continued sequencing of an ever-expanding catalog of various types and states of tissues from reference organisms, metagenomic sequencing of environmental (3) and microbiome (4) samples is also expected to enjoy a similar immense growth.

Given the expansive repository of existing sequencing data and the rate of acquisition, Muir et al. (5) argue that the ability of comptuational approaches to keep pace with data acquisition has become one of the main bottlenecks in contemporary genomics. These needs have spurred methods developers to produce ever more efficient and scalable computational methods for a variety of genomics analysis tasks, from genome and transcriptome assembly to pan-genome analysis. Against this backdrop, the de Bruijn graph, along with its variants, has become a compact and efficient data representation of increasing importance and utility across computational genomics.

The de Bruijn graph originated in combinatorics as a mathematical construct devised to prove a conjecture about binary strings posed by Ir. K. Posthumus (6, 7). In bioinformatics, de Bruijn graphs were introduced in the context of genome assembly algorithms for short-reads (8, 9), although the graph introduced in this context adopts a slightly different definition than in combinatorics. Subsequently, the de Bruijn graph has gradually been used in an increasing variety of different contexts within computational biology, including but not limited to: read correction (10, 11), genomic data compression (12), genotyping (13), structural variant detection (14), read mapping (15, 16), sequence-similarity search (17), metagenomic sequence analysis (18–20), transcriptome assembly (21, 22), transcript quantification (23), and long-read assembly (24–26).

In the context of fragment assembly—whether in forming contigs for whole-genome assembly pipelines (27, 28), or in encapsulating the read set into a summary representative structure for a host of downstream analyses (29–32)—de Bruijn graphs continue to be used extensively. The non-branching paths in de Bruijn graphs are uniquely-assemblable contiguous sequences (known as *unitigs*) from the sequencing reads. Thus, they are certain to be present in any faithful genomic reconstruction from these reads, have no ambiguities regarding repeats in the data, and are fully consistent with the input. As such, maximal unitigs are excellent candidates to summarize the raw reads, capturing their essential substance, and are usually the output of the initial phase of modern *de novo* short-read assembly tools. Collapsing a set of reads into this compact set of fragments that preserve their effective information can directly contribute to the efficiency of many downstream analyses over the read set.

When constructed from reference genome sequences, the unitigs in the de Bruijn graphs correspond to substrings in the references that are shared identically across subsets of the genomes. Decomposing the reference collection into these fragments retains much of its effective information, while typically requiring much less space and memory to store, index, and analyze, than processing the collection of linear genomes directly. The ability to compactly and efficiently represent shared sequences has led many modern sequence analysis tools to adopt the de Bruijn graph as a central representation, including sequence indexers (33), read aligners (15, 16), homology mappers (34, 35), and RNA-seq analysis tools (23, 36, 37). Likewise, pan-genome analysis tools (38–43) frequently make use of the maximal unitigs of the input references as the primary units upon which their core data structures and algorithms are built.

The vast majority of the examples described above make use of the *compacted* de Bruijn graph. A de Bruijn graph is compacted by collapsing each of its maximal, non-branching paths (unitigs) into a single vertex. Many computational genomics workflows employing the (compacted) de Bruijn graph are multi-phased, and typically, their most resource-intensive step is the initial one: construction of the regular and/or the compacted de Bruijn graph. The computational requirements for constructing the graph are often considerably higher than the downstream steps—posing major bottlenecks in many applications (13, 30). As such, there has been a concerted effort over the past several years to develop resource-frugal methods capable of constructing the compacted graph (44–51). Critically, solving this problem efficiently and in a context independent from any specific downstream application yields a modular tool (45, 47) that can be used to enable a wide variety of subsequent computational pipelines.

To address the scalability challenges of constructing the compacted de Bruijn graph, we recently proposed a novel algorithm, CUTTLEFISH (44), that exhibited faster performance than pre-existing state-of-the-art tools, using (often multiple times) less memory. However, the presented algorithm is only applicable when constructing the graph from existing reference sequences. It cannot be applied in a number of contexts, such as fragment assembly or contig extraction from raw sequencing data. In this paper, we present a fast and memory-frugal algorithm for constructing compacted de Bruijn graphs, CUTTLEFISH 2, applicable *both* on raw sequencing short-reads and assembled references, that can scale to very large datasets. It builds upon the novel idea of modeling de Bruijn graph vertices as Deterministic Finite Automata (DFA) (52) from Khan and Patro (44). However, the DFA model itself has been modified, and the algorithm has been generalized, so as to accommodate all valid forms of input. At the same time, in the case of constructing the graph from reference sequences, it is considerably faster than the previous approach, while retaining its frugal memory profile. We evaluated CUTTLEFISH 2 on a collection of datasets with diverse characteristics, and assess its performance compared to other leading compacted de Bruijn graph construction methods. We observed that CUTTLEFISH 2 demonstrates superior performance in all the experiments we consider.

Additionaly, we demonstrate the flexibility of our approach by presenting another application of the algorithm. The compacted de Bruijn graph forms a vertex-decomposition of the graph, while preserving the graph topology (47). However, for some applications, only the vertex-decomposition is sufficient, and preservation of the topology is redundant. For example, for applications such as performing presence-absence queries for k-mers or associating information to the constituent k-mers of the input (53, 54), any set of strings that preserves the exact set of k-mers from the input sequences can be sufficient. Relaxing the defining requirement of unitigs, that the paths be non-branching in the underlying graph, and seeking instead a set of maximal non-overlapping paths covering the de Bruijn graph, results in a more compact representation of the input data. This idea has recently been explored in the literature, with the representation being referred to as a spectrum-preserving string set (55), and the paths themselves as simplitigs (56). We demonstrate that CUTTLEFISH 2 can seamlessly extract such maximal path covers by simply constraining the algorithm to operate on some specific subgraph(s) of the original graph. We compared it to the existing tools available in the literature (57) for constructing this representation, and observed that it outperforms those in terms of resource requirements.

2. Results

2.1. CUTTLEFISH 2 overview. We present a high-level overview of the CUTTLEFISH 2 algorithm here. A complete treatment is provided in Section 4.3.

CUTTLEFISH 2 takes as input a set $\mathcal{R}$ of strings, that are either short-reads or whole-genome references, a k-mer length k, and a frequency threshold $f_0 \geq 1$. As output, it produces the maximal unitigs of the de Bruijn graph $G(\mathcal{R}, k)$. Fig. 1 highlights the major steps in the algorithm.

CUTTLEFISH 2 first enumerates the set $\mathcal{E}$ of edges of $G(\mathcal{R}, k)$, the $(k+1)$-mers present at least f_0 times in $\mathcal{R}$. This way the potential sequencing errors, present in case in which read sets are given as input, are discarded. Then the set $\mathcal{V}$ of vertices of $G(\mathcal{R}, k)$, which are the k-mers present in these $(k+1)$-mers, are extracted from $\mathcal{E}$. Next, a Minimal Perfect Hash Function (MPHF) $\mathfrak{f}$ over these vertices is constructed, that maps them bijectively to $[1, |\mathcal{V}|]$. This provides a space-efficient way to associate information to the vertices through hashing. Modeling each vertex $v \in \mathcal{V}$ as a Deterministic Finite Automaton (DFA), a piecewise traversal on $G(\mathcal{R}, k)$ is made using $\mathcal{E}$, computing the state S_v of the automaton of each $v \in \mathcal{V}$—associated to v through $\mathfrak{f}(v)$. The DFA modeling scheme ensures the retention of just enough information per vertex, such that the maximal unitigs are constructible afterwards from the automata states. Then, with another piecewise traversal on $G(\mathcal{R}, k)$ using $\mathcal{V}$ and the states collection S, CUTTLEFISH 2 retrieves all the non-branching edges of $G(\mathcal{R}, k)$—retained by the earlier traversal—and stitches them together in chains, constructing the maximal unitigs.

2.2. Experiments. We performed a number of experiments to characterize the various facets of the CUTTLEFISH 2 algorithm, its implementation, and some potential applications. We evaluated its execution performance compared to other available implementations of leading algorithms on de Bruijn graphs solving—(1) the compacted graph construction and (2) the maximal path cover problems, applicable on shared-memory multi-core machines. Although potentially feasible, CUTTLEFISH 2 is not designed as a method to leverage the capability of being distributed on a cluster of compute-nodes. Therefore, we did not consider relevant tools operating in that paradigm. We assessed its ability to construct compacted graphs and path covers for both sequencing reads and large

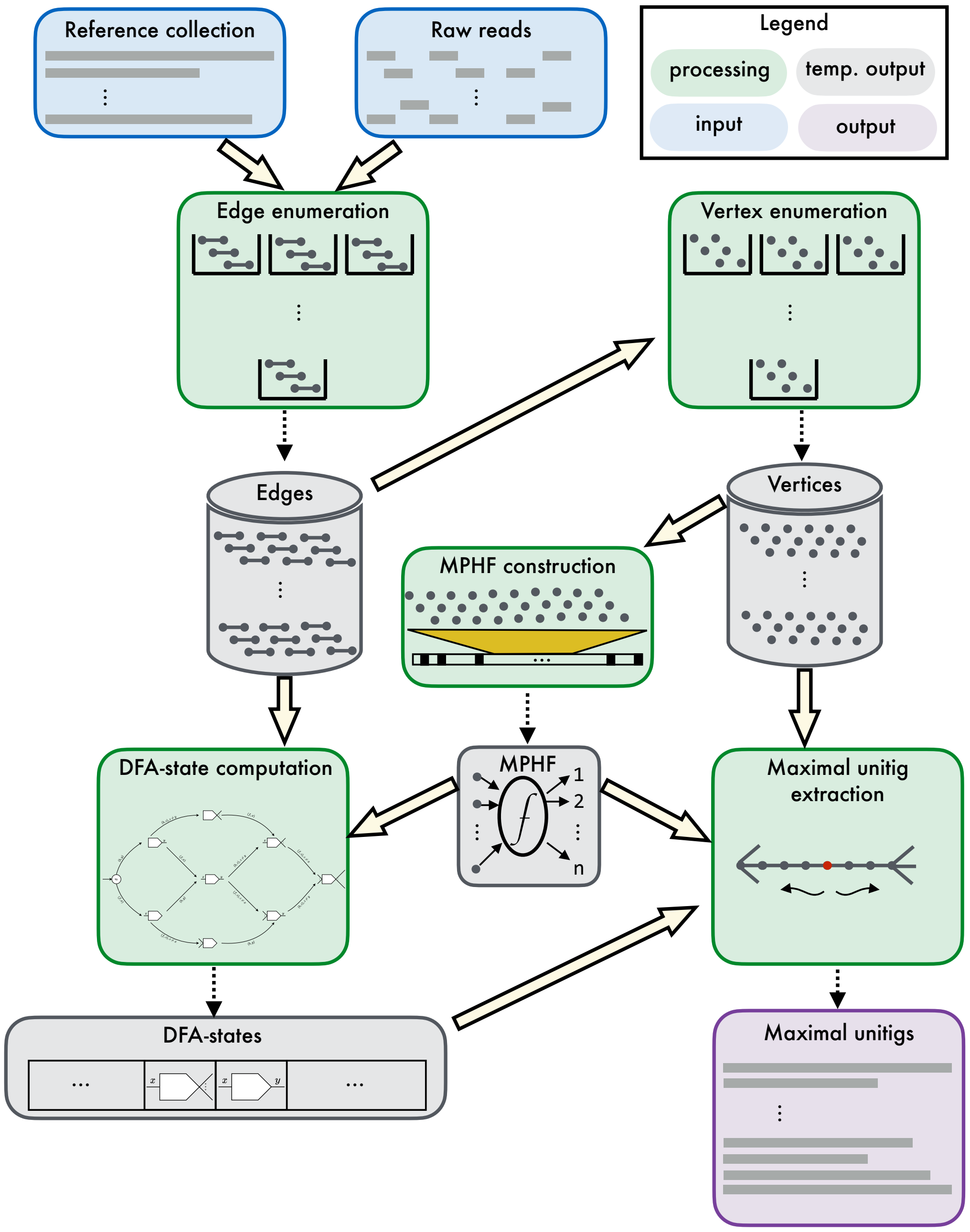


Figure 1: An overview of the CUTTLEFISH 2 algorithm. It is capable of constructing the compacted de Bruijn graph from a collection of either reference sequences or raw sequencing reads. The edges ($(k+1)$-mers) of the underlying de Bruijn graph are enumerated from the input, and optionally filtered based on the user-defined threshold. The edges are then used to enumerate the vertices (k-mers) they contain. An MPHF is constructed over the set of vertices, to associate the DFA-state of each vertex to it. Then the edge set is iterated over to determine the state of the DFA of each vertex in the graph, by transitioning the DFA through appropriate states, based on the edges in which the vertex is observed. Then an iteration over the original vertices to stitch together appropriate edges allows the extraction of the maximal unitigs.

pan-genome collections. By working on the $(k+1)$-mer spectrum, the new method performs a substantial amount of data reduction on the input sequences, yielding considerable speedups over the CUTTLEFISH algorithm (44) that, instead, requires multiple passes over the input sequences.

Next, we assess some structural characteristics of the algorithm and its implementation. Given an input dataset and a fixed internal parameter γ, the time- and the space-complexity of CUTTLEFISH 2 depend on k (see Section 4.4). We evaluated the impact of k on its execution performance, and also assessed some structural properties of the compacted graph that change with the parameter k. Moreover, we appraised the parallel scalability of the different steps of the algorithm, characterizing the ones that scale particularly well with increasing processor-thread count, as well as those that saturate more quickly.

A diverse collection of datasets has been used to conduct the experiments. We delineate the pertinent datasets for the experiments in their corresponding sections. The commands used for executing the different tools are available in Suppl. Sec. 1.8.

Computation system for evaluation. All experiments were performed on a single server with two Intel Xeon E5-2699 v4 2.20 GHz CPUs having 44 cores in total and enabling up-to 88 threads, 512 GB of 2.40 GHz DDR4 RAM, and a number of 3.6 TB Toshiba MG03ACA4 ATA HDDs. The system is run with Ubuntu 16.10 GNU/Linux 4.8.0-59-generic. The running times and the maximum memory usages were measured with the GNU `time` command, and the intermediate disk-usages were measured using the Linux commands `inotifywait` and `du`.

2.3. Compacted graph construction for sequencing reads. We evaluated the performance of CUTTLEFISH 2 in constructing compacted de Bruijn graphs from short-read sequencing data compared to available implementations of other leading compaction algorithms: (1) ABYSS-BLOOM-DBG, the maximal unitigs assembler of the ABYSS 2.0 assembly-pipeline (27), (2) BIFROST (45), (3) DEGSM (46), and (4) BCALM 2 (47).

The performances were tested on a number of short-read datasets with varied characteristics: (1) Mammalian dataset: a human read set (NIST HG004) from an Ashkenazi white female *Homo Sapiens* (paired-end 250 bp Illumina reads with 70x coverage, SRA3440461–95, 148 GB compressed FASTQ), from Zook et al. (58); (2) Metagenomic datasets: (i) a gut microbiome read set (ENA PRJEB33098) from nine individuals (paired-end 150 bp Illumina reads with high coverage, ERP115863, 45 GB compressed FASTQ), from Mas-Lloret et al. (59); and (ii) a soil metagenome read set (Iowa Corn) from 100-years-cultivated Iowa agricultural corn soil (paired-end 76 bp and 114 bp Illumina reads with low coverage, SRX100357 and SRX099904–06, 152 GB compressed FASTQ), used by Howe et al. (60); and (3) Large organism dataset: a white spruce read set (NCBI PRJNA83435) from a Canadian *Picea glauca* tree (paired-end 150 bp and 100 bp Illumina reads with high coverage, SRA056234, 1.14 TB compressed FASTQ), from Birol et al. (61). Table 1 contains the summary results of the benchmarking.

The frequency threshold f_0 of k-mers ($(k+1)$-mers in case of CUTTLEFISH 2 [1]) for the algorithms was approximated using k-mer frequency distributions so as to roughly minimize the misclassification rates of *weak* and *solid* k-mers [2] in these experiments (See Suppl. Sec. 1.1). In many practical scenarios, it might be preferable to skip computing an (approximate) frequency distribution, setting f_0 through some informed choice based on the properties of the input data (e.g. the sequencing depth and protocol). This can incorporate more weak k-mers into the graph. We present the results for such a scenario in Suppl. Table 2 on the human read set, setting f_0 to just 2.

Across the different datasets and algorithms evaluated, several trends emerge, notable from Table 1. First, we observe that for every dataset considered, CUTTLEFISH 2 is the fastest tool to process the data, while *simultaneously* using the least amount of memory. If we allow CUTTLEFISH 2 to match the memory used by the second most memory-frugal method (which is always BCALM 2 here), then it often completes even more quickly. We note that CUTTLEFISH 2 retains its performance lead over the alternative approaches across a wide range of different data input characteristics.

Among all the methods tested, CUTTLEFISH 2 and BCALM 2 were the only tools able to process all the datasets to completion under the different configurations tested, within the memory and disk-space constraints of the testing system. The rest of the methods generally required substantially more memory, sometimes over an order of magnitude more, depending on the dataset.

Of particular interest is CUTTLEFISH 2's performance compared against BCALM 2. Relative to BCALM 2, CUTTLEFISH 2 is 1.7–5.3x faster on the human read set, while using 2.1–2.8x less memory. For the metagenomic datasets, it is 4.1–5.9x faster and uses 2.2–3.1x less memory on the gut microbiome data, and is 2.8–8.5x faster using 2.5–2.7x less memory on the soil data. On the largest sequencing dataset here, the white spruce read set, CUTTLEFISH 2 is 5.4–5.7x faster and is 1.3–2.6x memory-frugal—taking about 10 hours, compared to at least 54 hours for BCALM 2.

The timing-profile of BCALM 2 and CUTTLEFISH 2 excluding their similar initial stage: k-mer and $(k+1)$-mer enumeration, respectively, are shown in Suppl. Table 4. We also note some statistics of the de Bruijn graphs and their compacted forms for these datasets in Suppl. Table 5.

2.4. Compacted graph construction for reference collections. We assessed the execution performance of CUTTLEFISH 2 in constructing compacted de Bruijn graphs from whole-genome sequence collections in comparison to the available implementations of the following leading algorithms: (1) BIFROST (45), (2) DEGSM (46), and (3) BCALM 2 (47). TWOPACO (48) is another notable algorithm applicable in this scenario, but we did not include it in the benchmarking as its output step lacks a parallelized implementation, and we consider very large sequence collections in this experiment.

We evaluated the performances on a number of datasets with varying attributes: (1) Metagenomic collection: 30,691 representative sequences from the most prevalent human gut prokaryotic genomes, gathered by Hiseni et al. (65) (≈ 61B bp, 18 GB compressed FASTA); (2) Mammalian collection: 100 human genomes—7 real sequences from Baier et al. (49) and 93 sequences simulated by Minkin et al. (48) (≈ 294B bp, 305 GB uncompressed FASTA); and (3) Bacterial archive: 661,405 bacterial genomes, collected by Blackwell et al. (66) from the European Nucleotide Archive (≈ 2.58T bp, 752 GB compressed FASTA). Table 2 conveys the summary results of the benchmarking.

[1] From our observations, the distributions of k-mer frequencies and of $(k+1)$-mer frequencies on real data tend to agree closely, resulting in the same f_0 for these experiments for both CUTTLEFISH 2 and the rest of the algorithms, as per the setting-policy used.

[2] k-mers occurring frequently enough in input NGS reads are said to be solid k-mers, and the other ones are said to be weak (64).

Table 1: Time- and memory-performance results for constructing compacted de Bruijn graphs from short-read sets.

			ABySS-Bloom-dBG		Bifrost	deGSM	BCALM 2	Cuttlefish 2		
Dataset	k	Thread-count	Small-memory	Large-memory				Default memory	Match second-best memory	Un-restricted memory
Human	27	8	22h 18m (39.3)	20h 23m (71.3)	2d 09h 32m (340.7)	10h 36m (235.8)	04h 23m (6.7)	**01h 13m (3.2)**	01h 10m (6.2)	01h (11.3)
		16	11h 38m (39.3)	11h 02m (71.3)	2d 06h 29m (343.8)	07h 08m (235.8)	04h 58m (8.9)	**56m (3.3)**	56m (7.6)	51m (11.3)
	55	8	16h 32m (34.0)	15h 58m (66.0)	10h 03m (70.8)	16h 50m (293.2)	04h 01m (7.4)	**02h 20m (3.5)**	01h 08m (7.1)	01h 03m (11.3)
		16	09h 28m (34.1)	08h 37m (66.1)	06h 31m (70.8)	15h 54m (293.3)	04h 26m (10.5)	**02h 02m (3.7)**	01h 11m (9.5)	51m (11.3)
Gut microbiome	27	16	18h 47m (42.0)	20h 12m (74.0)	18h 05m (189.9)	02h 28m (157.2)	02h 34m (7.7)	**26m (3.5)**	23m (6.7)	20m (26.8)
	55		1d 17h 43m (35.9)	1d 08h 09m (67.8)	03h 33m (53.0)	06h 53m (293.3)	03h 02m (12.5)	**44m (4.0)**	25m (11.3)	20m (69.9)
Soil	27	16	1d 18h 35m (150.4)	14h 24m (275.0)	*	1d 14h 29m (235.8)	19h 39m (52.0)	**02h 01m (19.2))**	02h 18m (40.9)	01h 35m (40.9)
	55		07h 57m (128.9)	06h 36m (256.8)	08h 12m (155.1)	1d 11h 05m (293.3)	08h 30m (27.5)	**03h 02m (11.1)**	02h 43m (23.3)	01h 38m (23.3)
White spruce	27	16	*	X	X	†	2d 06h 12m (36.8)	**10h 05m (14.0)**	07h 47m (35.2)	07h 13m (204.2)
	55		*	X	X	†	2d 09h 59m (31.6)	**10h 12m (23.8)**	10h 08m (31.1)	07h 24m (279.3)

Each cell contains the running time in wall clock format, and the maximum memory usage in gigabytes, in parentheses. The frequency thresholds f_0 used are as follows: (i) human: 14 (k = 27) and 9 (k = 55); (ii) gut microbiome and soil: 2; and (iii) white spruce: 11 (k = 27) and 7 (k = 55). Some details on executing the different tool implementations are as follows: (1) ABySS-Bloom-dBG has two tunable parameters significantly affecting its performance: a Bloom filter (62) memory budget and the number of hash functions for the filters. We executed it with two configurations: small-memory (with 4 hashes) and large-memory (with 3 hashes). The memory budgets used in these configurations are as follows: (i) human and gut microbiome: 32 GB and 64 GB; (ii) soil: 64 GB and 128 GB; and (iii) white spruce: 400 GB, and no large-memory execution due to hardware limitations. (2) Bifrost does not support the usage of arbitrary f_0, and uses a default $f_0 = 2$. For a uniform comparison across the tools with $f_0 = 2$ on the human dataset, see Supplementary Table 2. We did not execute Bifrost on the white spruce dataset due to this limitation—while on the human dataset the increases in the vertex-count for Bifrost are approximately 26% ($k = 27$) and 19% ($k = 55$), these are 91% and 45% respectively on the white spruce dataset. (3) deGSM has a maximum-memory parameter, with an upper-limit of 128 GB. We observed that its internal k-mer enumeration steps using Jellyfish (63) use more memory than this limit in all the experiments, and therefore we used 128 GB for deGSM in all its executions. (4) BCALM 2 also has a maximum-memory option, which we set to the best memory usage obtained from the rest of the algorithms. It also has a maximum disk usage option, which we set to the entire usable space (3.4 TB) of the disk used for its working directory, for maximum efficiency. (5) The Cuttlefish 2 implementation also supports tunable memory up-to a certain extent, and we executed it with three settings: (i) default memory: using the default minimum memory of ≈ 9.7 bits/vertex (see Section 4.4.2); (ii) match second-best memory: using up-to the memory amount found best in executions other than Cuttlefish 2 strict-memory mode; and (iii) unrestricted memory: using no strict upper-limit for memory.
The best performance with respect to each metric in each row is highlighted, where only the default-memory mode is considered for Cuttlefish 2. The $*$'s and the †'s denote that the corresponding executions could not complete due to hardware shortage of memory and disk-space, respectively. The X's denote that the corresponding executions were not run for reasons noted earlier. Supplementary Table 1 also includes the intermediate disk-usages incurred by the tools, besides time and memory.

Evaluating the performance of the different tools over these pan-genomic datasets, we observe similar trends to what was observed in Table 1, but with even more extreme differences than before. For a majority of the experiment configurations here, only BCALM 2 and Cuttlefish 2 were able to finish processing within time- and machine-constraints. Again, Cuttlefish 2 exhibits the fastest runtime on all datasets, and the lowest memory usage on all datasets except the human gut genomes (where it consumes 1–2 GB more memory than BCALM 2, though taking 6 to 7 fewer hours to complete).

Cuttlefish 2 is 2.4–8.9x faster on the 30K human gut genomes compared to the closest competitors, using similar memory. On the 100 human reference sequences, Cuttlefish 2 is 4.3–4.7x faster, using 5.4–8.4x less memory. The only other tools able to construct this compacted graph successfully are deGSM for $k = 27$ (taking 4.3x as long and requiring 8.4x as much memory as Cuttlefish 2) and BCALM 2 for $k = 55$ (taking over 4.7x as long and 5.4x as much memory as Cuttlefish 2). Finally, when constructing the compacted graph on the 661,405 bacterial genomes, Cuttlefish 2 is the only tested tool able to construct the graph for $k = 27$. For $k = 55$, BCALM 2 also completed, taking about 4.5 days, while Cuttlefish 2 finished under a day, with similar memory-profile. Overall, we observe that for large pan-genome datasets, Cuttlefish 2 is not only considerably faster and more memory-frugal than alternative approaches, but is the only tool able to reliably construct the compacted de Bruijn graph under all the different configurations tested, within the constraints of the experimental system.

Table 4 notes the timing-profiles for BCALM 2 and Cuttlefish 2 without their first step of k-mer and $(k+1)$-mer enumerations, and Table 5 shows some characteristics of the (compacted) de Bruijn graphs for these pan-genome datasets.

2.5. Maximal path cover construction. The execution performance of Cuttlefish 2 in decomposing de Bruijn graphs into maximal vertex-disjoint paths was assessed compared to the only two available tool implementations in literature (57) for this task: (1) ProphAsm (56) and (2) UST (55). For sequencing data, we used: (1) a roundworm read set (ENA DRR008444) from a *Caenorhabditis elegans* nematode (paired-end 300 bp Illumina reads, 5.6 GB com-

Table 2: Time- and memory-performance results for constructing compacted de Bruijn graphs from whole-genome reference collections.

Dataset (genome count)	k	Thread-count	BIFROST	DEGSM	BCALM 2	CUTTLEFISH 2 Default memory	CUTTLEFISH 2 Unrestricted memory
Human gut (30K)	27	8	∗		10h 06m (21.5)	**01h 39m (15.2)**	01h 39m (32.5)
		16	∗		09h 05m (22.0)	**01h 01m (15.5)**	59m (32.5)
	55	8	10h 53m (290.1)	Δ	11h 49m (**18.6**)	**04h 14m** (20.6)	03h 42m (44.4)
		16	09h 10m (290.3)		09h 45m (**19.2**)	**03h 50m** (20.9)	03h 10m (44.3)
Human (100)	27	8	∗	19h 23m (235.8)	‡	**04h 32m (27.7)**	04h 09m (59.7)
		16	∗	14h 07m (235.8)	‡	**03h 19m (28.1)**	02h 49m (59.7)
	55	8	∗	†	2d 23h 31m (302.9)	**15h 08m (56.0)**	13h 47m (121.8)
		16	∗	†	∗	**12h (56.2)**	11h 33m (121.8)
Bacterial archive (661K)	27	16	X	X	‡	**16h 38m (48.7)**	16h 24m (104.9)
	55				4d 10h 11m (63.3)	**22h 44m (59.9)**	22h 20m (129.5)

Each cell contains the running time in wall clock format, and the maximum memory usage in gigabytes, in parentheses. All the inputs being genomic sequences, the frequency threshold f_0 is used as 1 with all the tools. The relevant execution details, i.e. setting policy of the maximum memory usage (and maximum disk usage, if applicable) for DEGSM, BCALM 2, and CUTTLEFISH 2 are the same as described in Table 1.
The best performance with respect to each metric in each row is highlighted, and only the default-memory mode is considered for CUTTLEFISH 2 for such. The ∗'s and the †'s denote that the corresponding executions failed to complete due to hardware shortage of memory and disk-space, respectively. The ‡'s in the BCALM 2 executions denote abnormal terminations, reporting an encountered logic-error. The Δ in the DEGSM cells for the human gut genomes dataset indicate that the DEGSM executions were stuck in an intermediate stage indefinitely, and they were allowed to run for at least 2 days before being explicitly terminated. For the bacterial archive, we did not execute BIFROST and DEGSM (denoted with the X's) as it is anticipated that insufficient resources would be available for the executions, given their resource-usages on the smaller datasets. Supplementary Table 3 also includes the intermediate disk-usages incurred by the tools, besides time and memory.

pressed FASTQ); (2) the gut microbiome read set (ENA PRJEB33098) noted earlier; and (3) the human read set (NIST HG004) noted earlier. For whole-genome data, we used sequences from: (1) a roundworm reference (*Caenorhabditis elegans*, VC2010) (67); (2) a human reference (*Homo sapiens*, GRCh38); and (3) 7 real humans, collected from Baier et al. (49). Table 3 presents the summary results of the benchmarking.

We note that CUTTLEFISH 2 outperforms the alternative tools for constructing maximal path covers in terms of the time and memory required. In the context of this task, CUTTLEFISH 2 also offers several qualitative benefits over these tools. For example, PROPHASM exposes only a single-threaded implementation. Further, it is restricted to values of $k \leq 32$ and only accepts genomic sequences as input (and thus is not applicable for read sets). UST first makes use of BCALM 2 for maximal unitigs extraction—which we observed to be outperformed by CUTTLEFISH 2 in the earlier experiments—and then employs a sequential graph traversal on the compacted graph to extract a maximal path cover. For this problem, CUTTLEFISH 2 bypasses the compacted graph construction, and directly extracts a maximal cover.

We observe that compared to the tools, CUTTLEFISH 2 is competitive on single-threaded executions. While on moderate-sized datasets using multiple threads, it was 2–3.8x faster than UST using 2.2–12.6x less memory on sequencing data, and for reference sequences it was 2.8–6.1x faster than UST using 2.9–6.3x less memory.

We also provide a comparison of the maximal unitig-based and the maximal path cover-based representations of de Bruijn graphs in Suppl. Table 6. We observe that, for the human read set, the path cover representation requires 19–24% less space than the unitigs. For the human genome reference and 7 humans pan-genome references, these reductions are 14–22%, and 20–25%, respectively. From the statistics of both the representations on the gut microbiome read set, it is evident that the corresponding de Bruijn graphs are highly branching, as might be expected for metagenomic data. The space reductions with path cover in these graphs are 33–36%.

2.6. Structural characteristics. Given an input dataset $\mathcal{R}$ and a fixed frequency threshold f_0 for the edges (i.e. $(k+1)$-mers), the structure of the de Bruijn graph $G(\mathcal{R}, k)$ is completely determined by the k-mer-size—the edge- and the vertex-counts depend on k, and the asymptotic characteristics of the algorithm are dictated only by the k-mer size k and the hash function space-time tradeoff factor γ (see Sec. 4.4). We evaluated how CUTTLEFISH 2 is affected practically by changes in the k-value. The human read set (NIST HG004) noted earlier was used for these evaluations.

For a range of increasing k-values (and a constant γ), we measured the execution performance of CUTTLEFISH 2, and the following metrics of the maximal unitigs it produced: the number of unitigs, the average and the maximum unitig lengths, along with the N50 [3] and the NGA50 [4] scores for

[3] Length ℓ of the longest contig such that all the contigs having lengths $\geq \ell$ sum in size to at least 50% of the sum size of the contigs.

[4] Analogous to N50, except for: (1) breaking the contigs into their constituent blocks that can be aligned to an associated reference sequence, and (2) replacing the sum size of contigs with the reference length.

Table 3: Time- and memory-performance results for decomposing de Bruijn graphs into maximal vertex-disjoint paths.

Short-read sets

Dataset	k	Thread-count	UST	CUTTLEFISH 2	
				Default memory	Un-restricted memory
Round-worm	27	1	22m (3.7)	**11m (2.9)**	09m (11.2)
		8	07m (3.6)	**02m (2.9)**	02m (11.1)
	55	1	24m (3.2)	**19m (2.9)**	15m (11.2)
		8	08m (3.3)	**02m (2.9)**	02m (11.2)
Gut micro-biome	27	1	09h 02m (39.2)	**04h 30m (3.1)**	04h 02m (26.8)
		8	03h 10m (39.2)	**53m (3.3)**	37m (26.9)
	55	1	10h 36m (34.8)	**06h 59m (3.6)**	05h 51m (69.9)
		8	03h 24m (34.8)	**01h 13m (3.8)**	49m (69.9)
Human	27	8	04h 56m (13.1)	**01h 18m (3.2)**	01h 01m (11.3)
	55		04h 56m (7.7)	**02h 29m (3.5)**	01h 11m (11.3)

Whole-genome references

Dataset	k	Thread-count	PROPHASM	UST	CUTTLEFISH 2	
					Default memory	Un-restricted memory
Round-worm	27	1	**03m** (3.9)	08m (5.6)	**03m (2.0)**	03m (3.1)
		8		02m **(0.8)**	**01m** (2.0)	01m (2.0)
	55	1	--	10m (7.3)	**04m (2.8)**	04m (3.9)
		8		02m **(1.2)**	**01m** (2.8)	01m (3.4)
Human	27	1	01h 54m (91.8)	03h 59m (38.6)	**01h 28m (3.1)**	01h 29m (11.2)
		8		01h 19m (10.3)	**14m (3.2)**	12m (11.3)
	55	1	--	04h 55m (30.2)	**02h 16m (3.2)**	02h 07m (11.3)
		8		01h 02m (10.0)	**22m (3.4)**	19m (11.2)
7-humans	27	1	*	04h 38m (20.7)	**01h 58m (3.1)**	01h 46m (11.2)
		8		01h 49m (20.2)	**18m (3.2)**	15m (11.2)
	55	1	--	05h 55m (20.7)	**02h 48m (3.4)**	02h 28m (11.2)
		8		01h 38m (20.2)	**27m (3.6)**	21m (11.2)

Each cell contains the running time in wall clock format, and the maximum memory usage in gigabytes, in parentheses. The frequency thresholds f_0 used for the read sets are as follows: (i) roundworm: 12 ($k = 27$) and 8 ($k = 55$); (ii) gut microbiome: 2; and (iii) human: 14 ($k = 27$) and 9 ($k = 55$). For the reference sequences, f_0 is 1.
The best performance with respect to each metric in each row is highlighted, where for CUTTLEFISH 2 only its default-memory mode is considered. The $*$ denotes that the corresponding PROPHASM execution could not complete due to hardware memory shortage.

contig-contiguity. Across the varying k's, Table 4 reports the performance- and the unitig-metrics.

The unitig-metrics on this data convey what one might expect—as k increases, so do the average and the maximum lengths of the maximal unitigs, and the N50 and NGA50 metrics, since the underlying de Bruijn graph typically gets less tangled as the k-mer size exceeds repeat lengths (69). It is worth noting that, since we consider here just the extraction of unitigs, and no graph cleaning or filtering steps (e.g. bubble popping and tip clipping), we expect the N50 to be fairly short.

Perhaps the more interesting observation to be gleaned from the results is the scaling behavior of CUTTLEFISH 2 in terms of k. While the smallest k-value leads to the fastest overall graph construction, with increase in the machine-word count to encode the k-mers, the increase in runtime is rather moderate with respect to k, which follows the expected asymptotics (see Sec. 4.4.1). On the other hand, we observe that this increase can be offset by allowing CUTTLEFISH 2 to execute with more memory (which helps in the bottleneck step, $(k+1)$-mer enumeration). We also note that, while the timing-profile exhibits reasonably good scalability over the parameter k, the effect on the required memory is rather small—it is not directly determined by the k-value, rather is completely dictated by the distinct k-mer count (see Sec. 4.4.2).

2.7. Parallel scaling.

We assessed the scalability of CUTTLEFISH 2 across a varying number of processor-threads. For this experiment, we downsampled the human read set NIST HG004 from 70x to 20x coverage and used this as input. We set k to 27 and 55, and executed CUTTLEFISH 2 with thread-counts ranging in 1–32. For $k = 27$, Fig. 2a shows the time incurred by each step of the algorithm, and Fig. 2b demonstrates their speedups (i.e. factor of improvement in the speed of execution with different number of processor-threads). Suppl. Fig. 2 shows these metrics for $k = 55$.

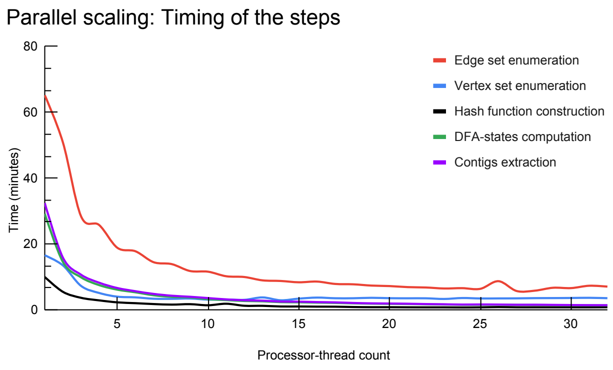
On the computation system used, we observe that all steps of CUTTLEFISH 2 scale well until about 8 threads. Beyond 8 threads, most steps but the minimal perfect hash construction continue to scale. Fig. 2a shows that the most time-intensive step in the algorithm is the initial edge set enumeration. This step, along with vertex enumeration and DFA states computation, continue to show reasonably good scaling behavior until about 20 threads, then gradually saturating. The final step of unitigs extraction seems to scale well up to the maximum thread-count we tested with (32 in this case).

It is worth reiterating that all experiments were performed on standard hard drives, and that the most resource-intensive step of edge enumeration can be quite input-output (IO) bound, while the rest of the steps also iterate through the in-disk set of edges or vertices—bound by disk-read speed. So one might expect different (and quite possibly better) scaling behavior for the IO-heavy operations when executing on faster external storage, e.g. in the form of SSD or NVMe drives (70). This is further evidenced by Kokot et al. (71), who show that KMC 3, the method used for the edge and the vertex enumeration steps in CUTTLEFISH 2, could have considerable gains in speed on large datasets when executed on SATA SSDs.

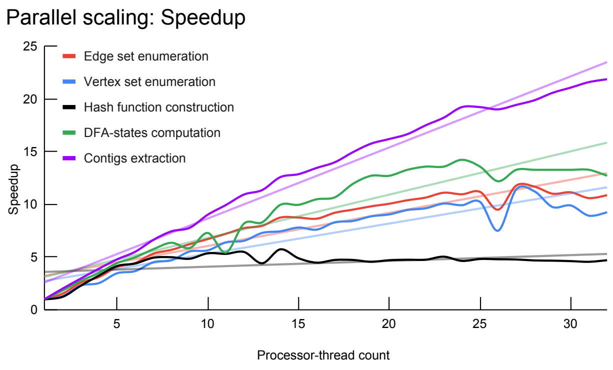
Table 4: Time- and memory-performance of CUTTLEFISH 2 for constructing the compacted de Bruijn graph from the human read set NIST HG004, and some corresponding metrics of the output maximal unitigs, over a range of k-mer sizes.

		Performance-metrics		Unitig-metrics				
k	k-mer count	Default memory	Unrestricted memory	Count	Avg. length (bp)	Max. length (bp)	N50 (bp)	NGA50 (bp)
27	2,547,479,119	1h 12m (3.19)	54m (11.29)	80,465,421	58	20,648	62	425
41	2,771,918,177	2h 19m (3.48)	1h 05m (11.26)	44,768,246	102	29,381	186	769
55	2,900,387,834	2h 12m (3.54)	1h 04m (11.28)	28,510,532	156	32,725	386	1,030
69	2,978,629,926	2h 42m (3.66)	1h 11m (19.49)	20,361,009	214	45,495	552	1,256
83	3,029,739,673	2h 39m (3.68)	1h 04m (22.34)	16,220,627	269	45,359	645	1,435
97	3,066,350,056	3h 05m (3.78)	1h 06m (30.57)	13,938,567	316	57,338	675	1,543
111	3,093,353,953	2h 53m (3.75)	1h 08m (32.18)	12,683,849	354	57,402	660	1,596
125	3,111,450,986	3h 01m (3.80)	1h 16m (42.18)	11,855,026	386	57,416	634	1,617

In performance-metrics, the running times are in wall clock format, and the maximum memory usages are in gigabytes, in parentheses. The frequency threshold f_0 for the $(k+1)$-mers is kept fixed at 5. The number of threads used in all the executions is 8. The setting policy of the execution modes (i.e. default-memory and unrestricted-memory) for CUTTLEFISH 2 is as described in Table 1. NGA50 is calculated using the tool `abyss-samtobreak`, having aligned the output contigs to the genome reference GRCh38 using BWA-MEM (68).



(a) Time incurred by each individual step.



(b) Speedup of each individual step.

Figure 2: Parallel-scaling metrics for CUTTLEFISH 2 across 1–32 processor threads, using $k = 27$ on the (downsampled) human read set NIST HG004, with the frequency threshold $f_0 = 4$.

3. Discussion

In this paper, we present CUTTLEFISH 2, a new algorithm for constructing the compacted de Bruijn graph, which is very fast and memory-frugal, and highly-scalable in terms of the extent of the input data it can handle. CUTTLEFISH 2 builds upon the work of Khan and Patro (44), which already advanced the state-of-the-art in reference-based compacted de Bruijn graph construction. CUTTLEFISH 2 simultaneously addresses the limitation and the bottleneck of CUTTLEFISH, by substantially generalizing the work to allow graph construction from both raw sequencing reads and reference genome sequences, while offering a more efficient performance profile. It achieves this, in large part, through bypassing the need to make multiple passes over the original input for very large datasets.

As a result, CUTTLEFISH 2 is able to construct compacted de Bruijn graphs much more quickly, while using less memory—both often multiple *times*—than the numerous other methods evaluated. Since the construction of the graph resides upstream of many computational genomics analysis pipelines, and as it is typically one of the most resource-intensive steps in these approaches, CUTTLEFISH 2 could help remove computational barriers to using the de Bruijn graph in analyzing the ever-larger corpora of genomic data.

In addition to the advances it represents in the compacted graph construction, we also demonstrate the ability of the algorithm to compute another spectrum-preserving string set of the input sequences—maximal path covers that have recently been adopted in a growing variety of applications in the literature (57). A simple restriction on the considered graph structure allows CUTTLEFISH 2 to build this construct much more efficiently than the existing methods.

As the scale of the data on which the de Bruijn graph and its variants must be constructed increases, and as the de Bruijn graph itself continues to find ever-more widespread uses in genomics, we anticipate that CUTTLEFISH 2 will enable its use in manifold downstream applications that may not have been possible earlier due to computational challenges, paving the way for an even more widespread role for the de Bruijn graph in high-throughput computational genomics.

CUTTLEFISH 2 is implemented in `C++14`, and is available open-source at `https://github.com/COMBINE-lab/cuttlefish`.

4. Methods

4.1. Related work. Here we briefly discuss the other compacted de Bruijn graph construction algorithms included in the experiments against which we compare CUTTLEFISH 2.

The BCALM algorithm (50) partitions the k-mers from the input that pass frequency filtering into a collection of disk-buckets according to their minimizers (72), and processes each bucket sequentially as per the minimizer-ordering—loading all the strings of the bucket into memory, joining (or, *compacting*) them maximally while keeping the resulting paths non-branching in the underlying de Bruijn graph, and distributing each resultant string into some other yet-to-be-processed bucket for potential further compaction, or to the final output. As is, BCALM is inherently sequential. BCALM 2 (47) builds upon this use of minimizers to partition the graph, but it modifies the k-mer partitioning strategy so that multiple disk-buckets can be compacted correctly in parallel, and then glues the further compactable strings from the compacted buckets.

ABYSS-BLOOM-DBG is the maximal unitigs assembler of the ABYSS 2.0 assembly tool (27). It first saves all the k-mers from the input reads into a cascading Bloom filter (62) to discard the likely-erroneous k-mers. Then it identifies the reads that consist entirely of retained k-mers, and extends them in both directions within the de Bruijn graph through identifying neighbors using the Bloom filter, while discarding the potentially false-positive paths based on their spans—producing the maximal unitigs.

DEGSM first enumerates all the $(k+2)$-mers of the input that pass frequency filtering. Then using a parallel external sorting over partitions of this set, it groups together the $(k+2)$-mers with the same middle k-mer, enabling it to identify the branching vertices in the de Bruijn graph. Then it merges the k-mers from the sorted buckets in a strategy so as to produce a Burrows-Wheeler Transform (73) of the maximal unitigs.

BIFROST (45) constructs an approximate compacted de Bruijn graph first by saving the k-mers from the input in a Bloom filter (62), and then for each potential non-erroneous k-mer, it extracts the maximal unitig containing it by extending the k-mer in both directions using the Bloom filter. Then using a k-mer counting based strategy, it refines the graph by removing the false edges induced by the Bloom filter.

4.2. Definitions. A *string* s is an ordered sequence of symbols drawn from an alphabet Σ. For the purposes of this paper, we assume all strings to be over the alphabet $\Sigma = \{A, C, G, T\}$, the DNA alphabet where each symbol has a reciprocal complement—the complementary pairs being $\{A, T\}$ and $\{C, G\}$. For a symbol $c \in \Sigma$, $\overline{c}$ denotes its complement. $|s|$ denotes the length of s. A k-*mer* is a string with length k. s_i denotes the i'th symbol in s (with 1-based indexing). A *substring* of s is a string entirely contained in s, and $s_{i..j}$ denotes the substring of s located from its i'th to the j'th indices, inclusive. $\mathrm{pre}_\ell(s)$ and $\mathrm{suf}_\ell(s)$ denote the prefix and the suffix of s with length ℓ respectively, i.e. $\mathrm{pre}_\ell(s) = s_{1..\ell}$ and $\mathrm{suf}_\ell(s) = s_{|s|-\ell+1..|s|}$, for some $0 < \ell \leq |s|$. For two strings x and y with $\mathrm{suf}_\ell(x) = \mathrm{pre}_\ell(y)$, the ℓ-*length glue* operation $\odot^\ell$ is defined as $x \odot^\ell y = x \cdot y_{\ell+1..|y|}$, where $\cdot$ denotes the *append* operation.

For a string s, its *reverse complement* $\overline{s}$ is the string obtained through reversing the order of the symbols in s, and replacing each symbol with its complement, i.e. $\overline{s} = \overline{s_{|s|}} \cdot \ldots \cdot \overline{s_2} \cdot \overline{s_1}$. The *canonical form* $\widehat{s}$ of s is defined as the string $\widehat{s} = \min(s, \overline{s})$, according to some consistent ordering of the strings in $\Sigma^{|s|}$. In this paper, we adopt the lexicographic ordering of the strings.

Given a set $\mathcal{S}$ of strings and an integer $k > 0$, let $\mathcal{K}$ and $\mathcal{K}_{+1}$ be respectively the sets of k-mers and $(k+1)$-mers present as substrings in some $s \in \mathcal{S}$. The (directed) *node-centric de Bruijn graph* $G_1(\mathcal{S}, k) = (\mathcal{V}_1, \mathcal{E}_1)$ is a directed graph where the vertex set is $\mathcal{V}_1 = \mathcal{K}$, and the edge set $\mathcal{E}_1$ is induced by $\mathcal{V}_1$: a directed edge $e = (u, v) \in \mathcal{E}_1$ iff $\mathrm{suf}_{k-1}(u) = \mathrm{pre}_{k-1}(v)$. The (directed) *edge-centric de Bruijn graph* $G_2(\mathcal{S}, k) = (\mathcal{V}_2, \mathcal{E}_2)$ is a directed graph where the edge set is $\mathcal{E}_2 = \mathcal{K}_{+1}$: each $e \in \mathcal{K}_{+1}$ constitutes a directed edge (v_1, v_2) where $v_1 = \mathrm{pre}_k(e)$ and $v_2 = \mathrm{suf}_k(e)$, and the vertex set $\mathcal{V}_2$ is thus induced by $\mathcal{E}_2$. [5]

In this work, we adopt the edge-centric definition of de Bruijn graphs. Hence, we use the terms k-mer and vertex and the terms $(k+1)$-mer and edge interchangeably. We introduce both variants of the graph here as we compare (in Section 2) our algorithm with some other methods that employ the node-centric definition.

We use the *bidirected* variant of de Bruijn graphs in the proposed algorithm, and redefine $\mathcal{K}_{+1}$ to be the set of canonical $(k+1)$-mers $\widehat{z}$ such that z or $\overline{z}$ appears as substring in some $s \in \mathcal{S}$. [6] For a bidirected edge-centric de Bruijn graph $G(\mathcal{S}, k) = (\mathcal{V}, \mathcal{E})$ — (i) the vertex set $\mathcal{V}$ is the set of canonical forms of the k-mers present as substrings in some $e \in \mathcal{K}_{+1}$, and (ii) the edge set is $\mathcal{E} = \mathcal{K}_{+1}$, where an $e \in \mathcal{E}$ connects the vertices $\widehat{\mathrm{pre}_k(e)}$ and $\widehat{\mathrm{suf}_k(e)}$. A vertex v has exactly two *sides*, referred to as the *front* side and the *back* side.

For a $(k+1)$-mer z such that $\widehat{z} \in \mathcal{K}_{+1}$, let $u = \widehat{\mathrm{pre}_k(z)}$ and $v = \widehat{\mathrm{suf}_k(z)}$. z induces an edge from the vertex u to the vertex v, and it is said to *exit* u and *enter* v. z connects to u's back iff $\mathrm{pre}_k(z)$ is in its canonical form, i.e. $\mathrm{pre}_k(z) = u$, and otherwise it connects to u's front. Conversely, z connects to v's front iff $\mathrm{suf}_k(z) = v$, and otherwise to v's back. Concisely put, z exits through u's back iff z's prefix k-mer is canonical, and enters through v's front iff z's suffix k-mer is canonical. The edge corresponding to z can be expressed as $\big((u, s_u), (v, s_v)\big)$: it connects from the side s_u of the vertex u to the side s_v of the vertex v.

[5] As per this definition, $\mathcal{V}_2 = \mathcal{K}$. We describe in Section 4.3 a practical consideration that implies that $\mathcal{V}_2$ need not necessarily be the same as $\mathcal{K}$ when some *filtering* is applied on the input $\mathcal{S}$ to generate $\mathcal{K}_{+1}$.

[6] This is to account for the DNA being double-stranded, and a genomic string may come from either of these oppositely-oriented complementary strands.

We prove in Lemma 1 (see Suppl. Sec. 3) that the two $(k+1)$-mers z and $\bar{z}$ correspond to the same edge, but in reversed directions: $\bar{z}$ induces the edge $\big((v, s_v), (u, s_u)\big)$—opposite from that of z. The two edges for z and $\bar{z}$ constitute a bidirected edge $e = \big\{(u, s_u), (v, s_v)\big\}$, corresponding to $\hat{z} \in \mathcal{E}$. While crossing e during a graph traversal, we say that e *spells* the $(k+1)$-mer z when the traversal is from (u, s_u) to (v, s_v), and it spells $\bar{z}$ in the opposite direction.

A *walk* $w = (v_0, e_1, v_1, \dots, v_{n-1}, e_n, v_n)$ is an alternating sequence of vertices and edges in $G(\mathcal{S}, k)$, where the vertices v_i and v_{i+1} are connected with the edge e_{i+1}, and e_i and e_{i+1} are incident to different sides of v_i. $|w|$ denotes its length in vertices, i.e. $|w| = n+1$. v_0 and v_n are its *endpoints*, and the v_i for $0 < i < n$ are its *internal vertices*. The walks $(v_0, e_1, \dots, e_n, v_n)$ and $(v_n, e_n, \dots, e_1, v_0)$ denote the same walk but in opposite *orientations*. If the edge e_i spells the $(k+1)$-mer l_i, then w spells $l_1 \odot^k l_2 \odot^k \dots \odot^k l_n$. If $|w| = 1$, then it spells v_0. A *path* is a walk without any repeated vertex.

A *unitig* is a path $p = (v_0, e_1, v_1, \dots, e_n, v_n)$ such that either $|p| = 1$, or in $G(\mathcal{S}, k)$:

1. each internal vertex v_i has exactly one incident edge at each of its sides, the edges being e_i and e_{i+1}
2. and v_0 has only e_1 and v_n has only e_n incident to their sides to which e_1 and e_n are incident to, respectively.

A *maximal unitig* is a unitig $p = (v_0, e_1, v_1, \dots, e_n, v_n)$ such that it cannot be extended anymore while retaining itself a unitig: there exists no x, y, e_0, or e_{n+1} such that $(x, e_0, v_0, \dots, e_n, v_n)$ or $(v_0, e_1, \dots, v_n, e_{n+1}, y)$ is a unitig. Fig. 3a illustrates an example of the de Bruijn graph and the relevant constructs defined.

A *compacted de Bruijn graph* $G_c(\mathcal{S}, k)$ is obtained through collapsing each maximal unitig of the de Bruijn graph $G(\mathcal{S}, k)$ into a single vertex, through contracting its constituent edges (74). Fig. 3b shows the compacted form of the graph in Fig. 3a. Given a set $\mathcal{S}$ of strings and an integer $k > 0$, the problem of constructing the compacted de Bruijn graph $G_c(\mathcal{S}, k)$ is to compute the maximal unitigs of $G(\mathcal{S}, k)$. [7]

A *vertex-disjoint path cover* $\mathcal{P}$ of $G(\mathcal{S}, k) = (\mathcal{V}, \mathcal{E})$ is a set of paths such that each vertex $v \in \mathcal{V}$ is present in exactly one path $p \in \mathcal{P}$. Unless stated otherwise, we refer to this construct simply as path cover. A *maximal path cover* is a path cover $\mathcal{P}$ such that no two paths in $\mathcal{P}$ can be joined into one single path, i.e. there exists no $p_1, p_2 \in \mathcal{P}$ such that $p_1 = (v_0, e_1, \dots, e_n, x)$ and $p_2 = (x, e'_1, \dots, e'_m, v'_m)$, for some $x \in \mathcal{V}$. Fig. 3a provides examples of such.

4.3. Algorithm. Given a set $\mathcal{R}$, either of short-reads sequenced from some biological sample, or of reference sequences, the construction of the compacted de Bruijn graph $G_c(\mathcal{R}, k)$ for some $k > 0$ is a data reduction problem in computational genomics. A simple algorithm to construct the compacted graph G_c involves constructing the ordinary de Bruijn graph $G(\mathcal{R}, k)$ at first, and then applying a graph traversal algorithm (75) to extract all the maximal non-branching paths in G. However, this approach requires constructing the ordinary graph and retaining it in memory for traversal (or organizing it in a way that it can be paged into memory for efficient traversal). Both of these tasks can be infeasible for large enough input samples. This prompts the requirement of methods to construct G_c directly from $\mathcal{R}$, bypassing G. CUTTLEFISH 2 is an asymptotically and practically efficient algorithm performing this task.

Another practical aspect of the problem is that the sequenced reads typically contain errors (76). This is offset through increasing the sequencing depth—even if a read $r \in \mathcal{R}$ contains some erroneous symbol at index i of the underlying sequence being sampled, a high enough sequencing depth should ensure that some other reads in $\mathcal{R}$ contain the correct nucleotide present at index i. Thus, in practice, these erroneous symbols need to be identified—usually heuristically—and the vertices and the edges of the graph corresponding to them need to be taken into account. CUTTLEFISH 2 discards the edges, i.e. $(k+1)$-mers, that each occur less than some threshold parameter f_0, and only operates with the edges having frequencies $\geq f_0$.

4.3.1. Implicit traversals over $G(\mathcal{R}, k)$**.** When the input is a set $\mathcal{S}$ of references, the CUTTLEFISH algorithm (44) makes a complete graph traversal on the reference de Bruijn graph [8] $G(\mathcal{S}, k)$ through a linear scan over each $s \in \mathcal{S}$. Also, the concept of *sentinels* [9] in $G(\mathcal{S}, k)$ ensures that a unitig can not span multiple input sequences. In one complete traversal, the branching vertices are characterized through obtaining a particular set of neighborhood relations; and then using another traversal, the maximal unitigs are extracted.

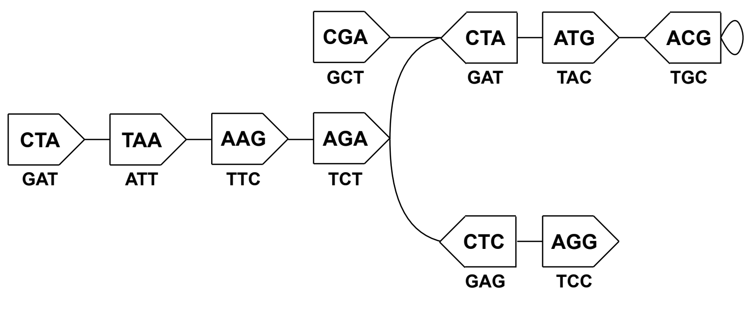
For a set $\mathcal{R}$ of short-reads however, a linear scan over a read $r \in \mathcal{R}$ may not provide a walk in $G(\mathcal{R}, k)$, since r may contain errors, which break a contiguous walk. Additionally, the concept of sentinels is not applicable for reads. Therefore, unitigs may span multiple reads. For a unitig u spanning the reads r_1 and r_2 consecutively, it is not readily inferrable that r_2 is to be scanned after r_1 (or the reverse, for an oppositely-oriented traversal) while attempting to extract u directly from $\mathcal{R}$, as the reads are not linked in the input for this purpose. Hence, contrary to the reference-input algorithm (44) where complete graph traversal is possible with just $|\mathcal{R}|$ different walks when $\mathcal{R}$ consists of reference sequences, CUTTLEFISH 2 resorts to making implicit piecewise-traversals over $G(\mathcal{R}, k)$.

For the purpose of determining the branching vertices, the piecewise-traversal is completely coarse—each walk traverses just one edge. Making such walks, CUTTLEFISH 2 retains just enough adjacency information for the vertices (i.e.

[7] Discarding orientations: the two unitigs $(v_0, \dots, v_n)$ and $(v_n, \dots, v_0)$ are topologically the same.

[8] Introduced by Khan and Patro (44), based on the input to the de Bruijn graph constructions being either reference sequences or sequencing reads, the graphs are distinguished as either *reference* or *read de Bruijn graphs*.

[9] A vertex v is a sentinel if the first or the last k-mer x of an input string corresponds to v. Let v's empty side in x be s_v. The graph $G(\mathcal{S}, k)$ is modified such that s_v connects to a special branching vertex Υ—preventing unitigs containing v to span farther through s_v.



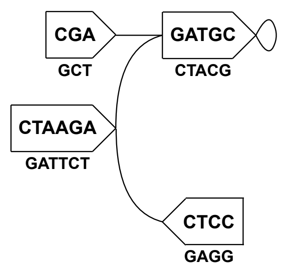


Figure 3: A (bidirected) edge-centric de Bruijn graph $G(\mathcal{S}, k)$ for a set $\mathcal{S} = \{CTAAGAT, CGATGCA, TAAGAGG\}$ of strings and k-mer size $k = 3$ in (a), and its compacted form $G_c(\mathcal{S}, k)$ in (b).
In the graphs, the vertices are denoted with pentagons—the flat and the cusped ends depict the $\mathtt{front}$ and the $\mathtt{back}$ sides respectively, and each edge corresponds to some 4-mer(s) in $s \in \mathcal{S}$. In (a), the vertices are the canonical forms of the k-mers in $s \in \mathcal{S}$. The canonical string $\widehat{t}$ associated to each vertex v is labelled inside v, to be spelled in the direction from v's $\mathtt{front}$ to its $\mathtt{back}$. Using $\widehat{t}$, we also refer to v. The label beneath v is $\overline{\widehat{t}}$, and is to be spelled in the opposite direction (i.e. $\mathtt{back}$ to $\mathtt{front}$). For example, consider the 4-mer CGAT, an edge e in $G(\mathcal{S}, 3)$. e connects the 3-mers $x = pre_3(e) = CGA$ and $y = suf_3(e) = GAT$, the vertices being $u = \widehat{x} = CGA$ and $v = \widehat{y} = ATC$ respectively. x is canonical and thus e exits through u's $\mathtt{back}$; whereas y is non-canonical and hence e enters through v's $\mathtt{back}$.
(CTA, TAA, AAG) is a walk, a path, and also a unitig (edges not listed). (CGA, ATC, ATG) is a walk and a path, but not a unitig—the internal vertex ATC has multiple incident edges at its $\mathtt{back}$. The unitig (CTA, TAA, AAG) is not maximal, as it can be extended farther through AAG's $\mathtt{back}$. Then it becomes maximal and spells CTAAGA. There are four such maximal unitigs in $G(\mathcal{S}, 3)$, and contracting each into a single vertex produces $G_c(\mathcal{S}, 3)$, in (b).
There are two different maximal path covers of $G(\mathcal{S}, 3)$: spelling $\{CTAAGATGC, CGA, CCTC\}$ and $\{CTAAGAGG, CGATGC\}$.

only the internal edges of the unitigs) so that the unitigs can then be piecewise-constructed without the input $\mathcal{R}$. Avoiding the erroneous vertices is done through traversing only the *solid* edges ($(k+1)$-mers occurring $\geq f_0$ times in $\mathcal{R}$, where f_0 is a heuristically-set input parameter).
Stitching together the pieces of a unitig is accomplished by making another piecewise-traversal over $G(\mathcal{R}, k)$, not by extracting those directly from the input sequences (opposed to CUTTLEFISH (44)). Each walk comprises the extent of a maximal unitig—the edges retained by the earlier traversal are used to make the walk and to stitch together the unitig.
In fact, the graph traversal strategy of CUTTLEFISH for reference inputs $\mathcal{S}$ is a specific case of this generalized traversal, where a complete graph traversal is possible through a *linear* scan over the input, as each $s \in \mathcal{S}$ constitutes a complete walk over $G(\{s\}, k)$. Besides, the maximal unitigs are tiled linearly in the sequences $s \in \mathcal{S}$, and determining their terminal vertices is the core problem in that case; as extraction of the unitigs can then be performed through walking between the terminal vertices by scanning the $s \in \mathcal{S}$.

4.3.2. A deterministic finite automaton model for vertices.

While traversing the de Bruijn graph $G(\mathcal{R}, k) = (\mathcal{V}, \mathcal{E})$ for the purpose of determining the maximal unitigs, it is sufficient to only keep track of information for each side s_v of each vertex $v \in \mathcal{V}$ that can categorize it into one of the following classes:

1. no edge has been observed to be incident to s_v yet
2. s_v has multiple distinct incident edges
3. s_v has exactly one distinct incident edge, for which there are $|\Sigma| = 4$ possibilities (see Lemma 2, Suppl. Sec. 3).

Thus there are six classes to which each s_v may belong, and since v has two sides, it can be in one of $6 \times 6 = 36$ distinct configurations. Each such configuration is referred to as a *state*.
CUTTLEFISH 2 treats each vertex $v \in \mathcal{V}$ as a Deterministic Finite Automaton (DFA) $M_v = (\mathcal{Q}, \Sigma', \delta, q_0, \mathcal{Q}')$:

States: $\mathcal{Q}$ is the set of the possible 36 states for the automaton. Letting the number of distinct edges at the $\mathtt{front}$ with f and at the $\mathtt{back}$ with b for a vertex v with a state q, and based on the incidence characteristics of v, the states $q \in \mathcal{Q}$ can be partitioned into four disjoint *state-classes*: (1) *fuzzy-front fuzzy-back* $(f \neq 1, b \neq 1)$, (2) *fuzzy-front unique-back* $(f \neq 1, b = 1)$, (3) *unique-front fuzzy-back*, $(f = 1, b \neq 1)$, and (4) *unique-front unique-back* $(f = 1, b = 1)$.

Input symbols: $\Sigma' = \{(s, c) : s \in \{\mathtt{front}, \mathtt{back}\}, c \in \Sigma\}$ is the set of the input symbols for the automaton. Each incident edge e of a vertex u is provided as input to u's automaton. For u, an incident edge $e = \{(u, s_u), (v, s_v)\}$ can be succinctly encoded by its incidence side s_u and a symbol $c \in \Sigma$—the $(k+1)$-mer $\widehat{z}$ for e is one of the following, depending on s_u and whether $\widehat{z}$ is exiting or entering u: $u \cdot c$, $\overline{u} \cdot c$, $c \cdot u$, or $c \cdot \overline{u}$.

Transition function: $\delta : \mathcal{Q} \times \Sigma' \rightarrow \mathcal{Q} \setminus \{q_0\}$ is the function controlling the state-transitions of the automaton. Fig. 4 illustrates the state-transition diagram for an automaton as per δ.

Initial state: $q_0 \in \mathcal{Q}$ is the initial state of the automaton. This state corresponds to the configuration of the associated vertex at which no edge has been observed to be incident to either of its sides.

Accept states: $\mathcal{Q}' = \mathcal{Q} \setminus \{q_0\}$ is the set of the states corresponding to vertex-configurations having at least one

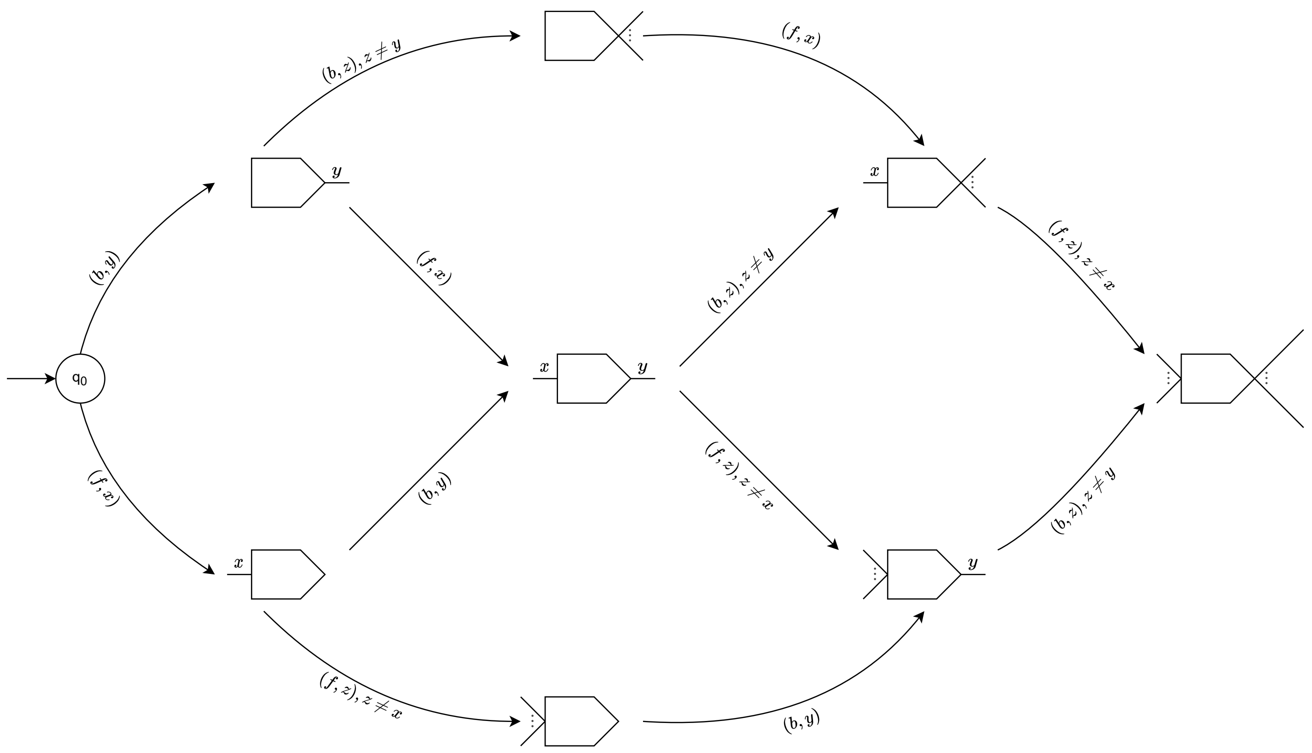


Figure 4: The state-transition diagram for an automaton $M_v = (\mathcal{Q}, \Sigma', \delta, q_0, \mathcal{Q}')$. Each node in the diagram represents a collection of states $q \in \mathcal{Q}$, and q_0 is the initial state of M_v. A node may represent multiple states collectively. For example, the node at the center of the figure with the symbols x and y at its flat and cusped ends respectively represents 16 states (all the ones from the state-class *unique-front unique-back*). Thus each node $\mathcal{Q}_k$ represents a disjoint subset of $\mathcal{Q}$. The pictorial shape of $\mathcal{Q}_k$ is similar to that of a de Bruijn graph vertex (see Fig. 3), and denotes the incidence characteristics of the vertices having their automata in states in $\mathcal{Q}_k$: for a vertex v with its automaton in state $q_k \in \mathcal{Q}_k$, a unique symbol at side s of $\mathcal{Q}_k$ means that v has a distinct edge at side s, ellipsis means multiple unique edges, and absence of any symbol means no edge has been observed incident to that side.
A directed edge $(\mathcal{Q}_i, \mathcal{Q}_j)$ labelled with (s, c) denotes transitions from a state $q_i \in \mathcal{Q}_i$ to a state $q_j \in \mathcal{Q}_j$, and (s, c) symbolizes the corresponding input fed to an automaton at state q_i for that transition to happen. That is, $\delta\big(q_i, (s, c)\big) = q_j$. Thus these edges pictorially encode the transition function δ. For the automaton M_v of a vertex v, an input (s, c) means that an edge e is being added to its side $s \in \{f, b\}$; along with s and v, the character $c \in \Sigma$ succinctly encodes e. f and b are shorthands for $\mathtt{front}$ and $\mathtt{back}$, respectively. Self-transition is possible for each state $q \in \mathcal{Q}'$, and are not shown here for brevity.

edge. [10]

4.3.3. Algorithm overview. We provide here a high-level overview of the CUTTLEFISH $2(\mathcal{R}, k, f_0)$ algorithm. The input to the algorithm is a set $\mathcal{R}$ of strings, an odd integer $k > 0$, and an abundance threshold $f_0 > 0$; the output is the set of strings spelled by the maximal unitigs of the de Bruijn graph $G(\mathcal{R}, k)$.

```
CUTTLEFISH 2($\mathcal{R}, k, f_0$)
1  $\mathcal{E} \leftarrow$ ENUMERATE-EDGES($\mathcal{R}, k, f_0$)
2  $\mathcal{V} \leftarrow$ EXTRACT-VERTICES($\mathcal{E}$)
3  $h \leftarrow$ CONSTRUCT-MINIMAL-PERFECT-HASH($\mathcal{V}$)
4  $S \leftarrow$ COMPUTE-AUTOMATON-STATES($\mathcal{E}, h$)
5  $\mathcal{U} \leftarrow$ EXTRACT-MAXIMAL-UNITIGS($\mathcal{V}, h, S$)
```

[10]Formally, $\mathcal{Q}'$ is the set of states reachable from q_0 through transitions as per some definite patterns of input symbols. For our purposes, recognizing specific input patterns is not a concern—rendering this parameter redundant—we define it as the set of the final states an automaton can be in having fed all its inputs.

CUTTLEFISH $2(\mathcal{R}, k, f_0)$ executes in five high-level stages, and Fig. 1 illustrates these steps. Firstly, it enumerates the set $\mathcal{E}$ of edges, i.e. $(k+1)$-mers that appear at least f_0 times in $\mathcal{R}$. Then the set $\mathcal{V}$ of vertices, i.e. k-mers are extracted from $\mathcal{E}$. Having the distinct k-mers, it constructs a minimal perfect hash function h over $\mathcal{V}$. At this point, a hash table structure is set up for the automata—the hash function being h, and each hash bucket having enough bits to store a state encoding. Then, making a piecewise traversal over $G(\mathcal{R}, k)$ using $\mathcal{E}$, CUTTLEFISH 2 observes all the adjacency relations in the graph, making appropriate state transitions along the way for the automata of the vertices u and v for each edge $\{(u, s_u), (v, s_v)\}$. After all the edges in $\mathcal{E}$ are processed, each automaton M_v resides in its correct state. Due to the design characteristic of the state-space $\mathcal{Q}$ of M_v, the internal vertices of the unitigs in $G(\mathcal{R}, k)$, as well as the non-branching sides of the branching vertices have their incident edges encoded in their states. CUTTLEFISH 2 retrieves these unitig-internal edges and stitches them together in chains until branching vertices are encountered, thus extracting the maximal unitigs implicitly through another piecewise traversal, with each

walk spanning a maximal unitig.
These major steps in the algorithm are detailed in the following subsections. Then we analyze the asymptotic characteristics of the algorithm in Section 4.4. Finally, we provide a proof of its correctness in Theorem 1 (see Suppl. Sec. 3).

4.3.4. Edge set construction. The initial enumeration of the edges, i.e. $(k+1)$-mers from the input set $\mathcal{R}$ is performed with the KMC 3 algorithm (71). KMC 3 enumerates the ℓ-mers of its input in two major steps. Firstly, it partitions the ℓ-mers based on *signatures*—a restrictive variant of *minimizers* [11]. Maximal substrings from the input strings with all their ℓ-mers having the same signature (referred to as *super ℓ-mers*) are partitioned into bins corresponding to the signatures. Typically the number of bins is much smaller than the number of possible signatures, and hence each bin may contain strings from multiple signatures (set heuristically to balance the bins). In the second phase, for each partition, its strings are split into substrings called (ℓ, x)-*mers*—an extension of ℓ-mers. These substrings are then sorted using a most-significant-digit radix sort (77) to unify the repeated ℓ-mers in the partition. For $\ell = k+1$, the collection of these deduplicated partitions constitute the edge set $\mathcal{E}$.

4.3.5. Vertex set extraction. CUTTLEFISH 2 extracts the distinct canonical k-mers—vertices of $G(\mathcal{R}, k)$—from $\mathcal{E}$ through an extension of KMC 3 (71) (See Suppl. Sec. 2.1). For such, taking $\mathcal{E}$ as input, KMC 3 treats each $(k+1)$-mer $e \in \mathcal{E}$ as an input string, and enumerates their constituent k-mers applying its original algorithm. Using $\mathcal{E}$ instead of $\mathcal{R}$ as input reduces the amount of work performed in this phase through utilizing the work done in the earlier phase—skipping another pass over the entire input set $\mathcal{R}$, which can be computationally substantial.

4.3.6. Hash table structure setup. An associative data structure is required to store the transitioning states of the automata of the vertices of $G(\mathcal{R}, k)$. That is, association of some encoding of the states to each canonical k-mer is required. Some off-the-shelf hash table can be employed for this purpose. Due to hash collisions, general-purpose hash tables typically need to store the keys along with their associated data—the key set $\mathcal{V}$ may end up taking $k \log_2 |\Sigma| = 2k$ bits/k-mer in the hash table [12]. In designing a more efficient hash table structure, CUTTLEFISH 2 makes use of the facts that: (i) the set $\mathcal{V}$ of keys is static—no alien keys will be encountered while traversing the edges in $\mathcal{E}$, since $\mathcal{V}$ is constructed from $\mathcal{E}$; and (ii) $\mathcal{V}$ has been enumerated at this point. A *Minimal Perfect Hash Function* (MPHF) is applicable in this setting. Given a set $\mathcal{K}$ of keys, a *perfect hash function* over $\mathcal{K}$ is an injective function $h : \mathcal{K} \to \mathbb{Z}$, i.e. $\forall k_1, k_2 \in \mathcal{K}$, $k_1 \neq k_2 \Leftrightarrow h(k_1) \neq h(k_2)$. h is minimal when its image is $[0, |\mathcal{K}|)$, i.e. an MPHF is an injective function $h : \mathcal{K} \to [0, |\mathcal{K}|)$. By definition, an MPHF does not incur collisions. Therefore when used as the hash function for a hash table, it obviates the requirement to store the keys with the table structure. Instead, some encoding of the MPHF needs to be stored in the structure.

To associate the automata to their states, CUTTLEFISH 2 uses a hash table with an MPHF as the hash function. An MPHF h over the vertex set $\mathcal{V}$ is constructed with the BBHASH algorithm (78). For the key set $\mathcal{V}_0 = \mathcal{V}$, BBHASH constructs h through a number of iterations. It maps each key $v \in \mathcal{V}_0$ to $[1, \gamma|\mathcal{V}_0|]$ with a classical hash function h_0, for a provided parameter $\gamma > 0$. The collision-free hashes in the hash codomain $[1, \gamma|\mathcal{V}_0|]$ are marked in a bit-array A_0 of length $\gamma|\mathcal{V}_0|$. The colliding keys are collected into a set $\mathcal{V}_1$, and the algorithm is iteratively applied on $\mathcal{V}_1$. The iterations are repeated until either some $\mathcal{V}_n$ is found empty, or a maximum level is reached. The bit-arrays A_i for the iterations are concatenated into an array A, which along with some other metadata, encode h. A has an expected size of $\gamma e^{1/\gamma}|\mathcal{V}|$ bits (78). γ trades off the encoding size of h with its computation time. $\gamma = 2$ provides a reasonable trade-off, with the size of h being ≈ 3.7 bits/vertex. [13] Note that, the size is independent of k, i.e. the size of the keys.

For the collection of hash buckets, CUTTLEFISH 2 uses a linear array (80) of size $|\mathcal{V}|$. Since each bucket is to contain some state $q \in \mathcal{Q}$, $\lceil \log_2 |\mathcal{Q}| \rceil = \lceil \log_2 36 \rceil = 6$ bits are necessary (and also sufficient) to encode q. Therefore CUTTLEFISH 2 uses 6 bits for each bucket. The hash table structure is thus composed of an MPHF h and a linear array S: for a vertex v, its (transitioning) state q_v is encoded at the index $h(v)$ of S, and in total the structure uses ≈ 9.7 bits/vertex.

4.3.7. Automaton states computation. Given the set $\mathcal{E}$ of edges of a de Bruijn graph $G(\mathcal{R}, k)$ and an MPHF h over its vertex set $\mathcal{V}$, the COMPUTE-AUTOMATON-STATES$(\mathcal{E}, h)$ algorithm computes the state of the automaton M_v of each $v \in \mathcal{V}$.

```
COMPUTE-AUTOMATON-STATES($\mathcal{E}, h$)
1  $n \leftarrow |keys(h)|$   // number of distinct keys for $h$,
                               i.e. the vertex-count
2  $S \leftarrow$ buckets table with $n$ states initialized to $q_0$
3  for each $e \in \mathcal{E}$
4      TRANSITION-STATES($e$)

TRANSITION-STATES($e$)
1  $u \leftarrow pre_k(e)$,  $v \leftarrow suf_k(e)$
2  $s_u \leftarrow$ EXIT-SIDE($\hat{u}, u$)
3  $s_v \leftarrow$ ENTRANCE-SIDE($\hat{v}, v$)
4  $c_u \leftarrow e_{k+1}$ if $s_u = back$, $\overline{e_{k+1}}$ o/w
5  $c_v \leftarrow e_1$ if $s_v = front$, $\overline{e_1}$ o/w

6  $S[h(\hat{u})] \leftarrow \delta\big(S[h(\hat{u})], (s_u, c_u)\big)$
7  $S[h(\hat{v})] \leftarrow \delta\big(S[h(\hat{v})], (s_v, c_v)\big)$
```

[11]For a given $j < \ell$, a j-minimizer of an ℓ-mer x is the smallest j-mer substring of x according to some specified function.

[12]This can be improved by having 4^p different hash tables for $\mathcal{V}$, for a fixed prefix length $p \leq k$. Each hash table then accounts for keys of length $(k-p)$.

[13]It can be as low as ≈ 3 bits/vertex with $\gamma = 1$, at the expense of slower hashing. The theoretical lower limit for MPHFs is ≈ 1.44 bits/key (79).

It initializes each automaton M_w with q_0—the initial state corresponding to no incident edges. Then for each edge $e = \{(\hat{u}, s_u), (\hat{v}, s_v)\} \in \mathcal{E}$, connecting the vertex $\hat{u}$ via its side s_u to the vertex $\hat{v}$ via its side s_v, it makes appropriate state transitions for M_u and M_v, the automata of $\hat{u}$ and $\hat{v}$ respectively. For each endpoint $\hat{w}$ of e, (s_w, c_w) is fed to M_w, where $c_w \in \Sigma$. Together with $\hat{w}$, s_w and c_w encode e. The setting policy for c_w is described in the following. Technicalities relating to loops are accounted for in the CUTTLEFISH 2 implementation, but are omitted from discussion for simplicity.

e has two associated $(k+1)$-mers: z and $\bar{z}$. Say that $z = u \odot^{k-1} v$. Based on whether $u = \hat{u}$ holds or not, e is incident to either $\hat{u}$'s back or front. As defined (see Section 4.2), if it is incident to the back, then $z = \hat{u} \cdot c$; otherwise, $z = \bar{\hat{u}} \cdot c$, where $c = e_{k+1}$. In these cases respectively, $\bar{z} = \bar{c} \cdot \bar{\hat{u}}$, and $\bar{z} = \bar{c} \cdot \hat{u}$. For consistency, CUTTLEFISH 2 always uses a fixed form of e for $\hat{u}$—either z or $\bar{z}$—to provide it as input to M_u: the one containing the k-mer u in its canonical form. So if e is at $\hat{u}$'s back, the $\hat{u} \cdot c$ form is used for e, and (back, c) is fed to M_u; otherwise, e is expressed as $\bar{c} \cdot \hat{u}$ and $(\mathrm{front}, \bar{c})$ is the input for M_u. The encoding (s_v, c') of e for $\hat{v}$ is set similarly.

4.3.8. Maximal Unitigs Extraction. Given the set $\mathcal{V}$ of vertices of a de Bruijn graph $G(\mathcal{R}, k)$, an MPHF h over $\mathcal{V}$, and the states-table S for the automata of $v \in \mathcal{V}$, the EXTRACT-MAXIMAL-UNITIGS$(\mathcal{V}, h, S)$ algorithm assembles all the maximal unitigs of $G(\mathcal{R}, k)$.

The algorithm iterates over the vertices in $\mathcal{V}$. For some vertex $\hat{v} \in \mathcal{V}$, let p be the maximal unitig containing $\hat{v}$. p can be broken into two subpaths: p_b and p_f, overlapping only at $\hat{v}$. The EXTRACT-MAXIMAL-UNITIGS$(\mathcal{V}, h, S)$ algorithm extracts these subpaths separately, and joins them at $\hat{v}$ to construct p. Then p's constituent vertices are marked by transitioning their automata to some special states (not shown in Fig. 4), so that p is extracted only once.

The subpaths p_b and p_f are extracted by initiating two walks: one from each of $\hat{v}$'s sides back and front, using the WALK-MAXIMAL-UNITIG$(\hat{v}, s_v)$ algorithm. Each walk continues on until a *flanking vertex* $\hat{x}$ is encountered. For a vertex $\hat{x}$, let q_x denote the state of $\hat{x}$'s automaton and $\mathcal{C}_x$ denote q_x's state-class. Then $\hat{x}$ is noted to be a flanking vertex iff:

1) either $\mathcal{C}_x$ is not *unique-front unique-back*;
2) or $\hat{x}$ connects to the side s_y of a vertex $\hat{y}$ such that:
   (a) $\mathcal{C}_y$ is *fuzzy-front fuzzy-back*; or
   (b) $s_y = \mathrm{front}$ and $\mathcal{C}_y$ is *fuzzy-front unique-back*; or
   (c) $s_y = \mathrm{back}$ and $\mathcal{C}_y$ is *unique-front fuzzy-back*.

Lemma 3 (see Suppl. Sec. 3) shows that the flanking vertices in $G(\mathcal{R}, k)$ are precisely the endpoints of its maximal unitigs. The WALK-MAXIMAL-UNITIG$(\hat{v}, s_v)$ algorithm initiates a walk w from $\hat{v}$, exiting through its side s_v. It fetches $\hat{v}$'s state q_v from the hash table. If q_v is found to be not belonging to the state-class *unique-front unique-back* due to s_v having

```
EXTRACT-MAXIMAL-UNITIGS($\mathcal{V}, h, S$)
 1  $\mathcal{U} \leftarrow \phi$
 2  for each $\hat{v} \in \mathcal{V}$
 3      $q_v \leftarrow S[h(\hat{v})]$
 4      if not IS-MARKED($q_v$)
 5          $p_b \leftarrow$ WALK-MAXIMAL-UNITIG($\hat{v}$, back)
 6          $p_f \leftarrow$ WALK-MAXIMAL-UNITIG($\hat{v}$, front)
 7          $p \leftarrow \overline{p_f} \odot^k p_b$
 8          MARK-VERTICES($p$)
 9          $\mathcal{U} \leftarrow \mathcal{U} \cup \{\hat{p}\}$
10  return $\mathcal{U}$
```

```
WALK-MAXIMAL-UNITIG($\hat{v}, s_v$)
 1  anchor $\leftarrow \hat{v}$
 2  $q \leftarrow S[h(\hat{v})]$
 3  $v \leftarrow \hat{v}$ if $s_v$ is back, $\bar{\hat{v}}$ o/w
 4  $p \leftarrow v$
 5  repeat
 6      if IS-FUZZY-SIDE($q, s_v$)
 7          break      // at a branching / dead-end side
 8      $e \leftarrow$ EDGE-EXTENSION($q, s_v$)
 9      $v \leftarrow suf_{k-1}(v) \cdot e$      // walk to the next vertex
10      if $\hat{v}$ == anchor
11          break      // a cyclic maximal unitig
12      $q \leftarrow S[h(\hat{v})]$
13      $s_v \leftarrow$ ENTRANCE-SIDE($\hat{v}, v$)
14      if IS-FUZZY-SIDE($q, s_v$)
15          break      // reached a different maximal unitig
16      $p \leftarrow p \cdot e$
17      $s_v \leftarrow$ EXIT-SIDE($\hat{v}, v$)      // get to the other side
18  return $p$
```

$\neq 1$ incident edges, then $\hat{v}$ is a flanking vertex of its containing maximal unitig p, and p has no edges at s_v. Hence w terminates at $\hat{v}$. Otherwise, s_v has exactly one incident edge. The walk algorithm makes use of the fact that, the vertex-sides s_u that are internal to the maximal unitigs in $G(\mathcal{R}, k)$ contain their adjacency information encoded in the states q_u of their vertices $\hat{u}$'s automata, once the COMPUTE-AUTOMATON-STATES$(\mathcal{E}, h)$ algorithm is executed. Thus, it decodes q_v to get the unique edge $e = \{(\hat{u}, s_u), (\hat{v}, s_v)\}$ incident to s_v. Through e, w reaches the neighboring vertex $\hat{u}$, at its side s_u. $\hat{u}$'s state q_u is fetched, and if q_u is found not to be in the class *unique-front unique-back* due to s_u having > 1 incident edges, then both $\hat{u}$ and $\hat{v}$ are flanking vertices (for different maximal unitigs), and w retracts to and stops at $\hat{v}$. Otherwise, e is internal to p, and w switches to the other side of $\hat{u}$, proceeding on similarly. It continues through vertices $\hat{v}_i$ in this manner until a flanking vertex of p is reached, stitching together the edges along the way to construct a subpath of p.

A few constant-time supplementary procedures are used throughout the algorithm. IS-FUZZY-SIDE(q, s) determines whether a vertex with the state q has 0 or > 1 edges at its side s. EDGE-EXTENSION(q, s) returns an encoding of the edge incident to the side s of a vertex with state q. ENTRANCE-SIDE$(\hat{v}, v)$ $\big($and EXIT-SIDE$(\hat{v}, v)\big)$ returns the side used to enter (and exit) the vertex $\hat{v}$ when its k-mer form

v is observed.

4.3.9. Maximal path-cover extraction. We discuss here how CUTTLEFISH 2 might be modified so that it can extract a maximal path cover of a de Bruijn graph $G(\mathcal{R}, k)$. For such, only the COMPUTE-AUTOMATON-STATES step needs to be modified, and the rest of the algorithm remains the same. Given the edge set $\mathcal{E}$ of the graph $G(\mathcal{R}, k)$ and an MPHF h over its vertex set $\mathcal{V}$, COMPUTE-AUTOMATON-STATES-PATH-COVER$(\mathcal{E}, h)$ presents the modified DFA states computation algorithm.

The maximal path cover extraction variant of CUTTLEFISH 2 works as follows. It starts with a trivial path cover $\mathcal{P}_0$ of $G(\mathcal{R}, k)$: each $v \in \mathcal{V}$ constitutes a single path, spanning the subgraph $G'(\mathcal{V}, \emptyset)$. Then it iterates over the edges $e \in \mathcal{E}$ (with $|\mathcal{E}| = m$) in arbitrary order. We will use $\mathcal{P}_i$ to refer to the path cover after having visited i edges. At any given point of the execution, the algorithm maintains the invariant that $\mathcal{P}_i$ is a maximal path cover of the graph $G'(\mathcal{V}, \mathcal{E}')$, where $\mathcal{E}' \subseteq \mathcal{E}$ (with $|\mathcal{E}'| = i$) is the set of the edges examined until that point. When examining the $(i+1)$'th edge $e = \{(u, s_u), (v, s_v)\}$, it checks whether e connects two different paths in $\mathcal{P}_i$ into one single path: this is possible iff the sides s_u and s_v do not have any incident edges already in $\mathcal{E}'$, i.e. the sides are empty in $G'(\mathcal{V}, \mathcal{E}')$. If this is the case, the paths are joined in $\mathcal{P}_{i+1}$ into a single path containing the new edge e. Otherwise, the path cover remains unchanged so that $\mathcal{P}_{i+1} = \mathcal{P}_i$. By definition, $\mathcal{P}_{i+1}$ is a path cover of $G'(\mathcal{V}, \mathcal{E}' \cup \{e\})$, as e could only affect the paths (at most two) in $\mathcal{P}_i$ containing u and v, while the rest are unaffected and retain maximality—thus the invariant is maintained. By induction, $\mathcal{P}_m$ is a path cover of $G(\mathcal{V}, \mathcal{E})$ once all the edges have been examined, i.e. when $\mathcal{E}' = \mathcal{E}$.

By making state transitions for the automata only for the edges present internal to the paths $p \in \mathcal{P}_m$, the COMPUTE-AUTOMATON-STATES-PATH-COVER$(\mathcal{E}, h)$ algorithm seamlessly captures the subgraph $G_{\mathcal{P}_m}$ of $G(\mathcal{R}, k)$ that is induced by the path cover $\mathcal{P}_m$. $G_{\mathcal{P}_m}$ consists of a collection of disconnected maximal paths, and thus there exists no branching in $G_{\mathcal{P}_m}$. Consequently, each of these maximal paths is a maximal unitig of $G_{\mathcal{P}_m}$. The subsequent EXTRACT-MAXIMAL-UNITIGS algorithm operates using the DFA states collection S computed at this step, and therefore it extracts precisely these maximal paths.

```
COMPUTE-AUTOMATON-STATES-PATH-COVER($\mathcal{E}, h$)
1  $n \leftarrow |keys(h)|$  // number of distinct keys for $h$,
                              i.e. the vertex-count
2  $S \leftarrow$ buckets table with $n$ states initialized to $q_0$
3  for each $e \in \mathcal{E}$
4      $u \leftarrow pre_k(e)$,  $v \leftarrow suf_k(e)$
5      $s_u \leftarrow$ EXIT-SIDE($e, \hat{u}$)
6      $s_v \leftarrow$ ENTRANCE-SIDE($e, \hat{v}$)
7      $q_u \leftarrow S[h(\hat{u})]$, $q_v \leftarrow S[h(\hat{v})]$

8      if IS-EMPTY-SIDE($q_u, s_u$) and
         IS-EMPTY-SIDE($q_v, s_v$)
9          TRANSITION-STATES($e$)
```

4.3.10. Parallelization. CUTTLEFISH 2 is highly parallelizable on a shared-memory multi-core machine. The ENUMERATE-EDGES and the EXTRACT-VERTICES steps, using KMC 3 (71), are parallelized in their constituent phases via parallel distribution of the input $(k+1)$-mers (and k-mers) into partitions, and sorting multiple partitions in parallel.

The COMPUTE-MINIMAL-PERFECT-HASH step using BBHASH (78) parallelizes the construction through distributing disjoint subsets $\mathcal{V}_i$ of the vertices to the processor-threads, and the threads process the $\mathcal{V}_i$'s in parallel.

The next two steps, COMPUTE-AUTOMATON-STATES and EXTRACT-MAXIMAL-UNITIGS, both (piecewise) traverse the graph through iterating over $\mathcal{E}$ and $\mathcal{V}$ respectively. The processor-threads are provided disjoint subsets of $\mathcal{E}$ and $\mathcal{V}$ to process in parallel. Although the threads process different edges in COMPUTE-AUTOMATON-STATES, multiple threads may access the same automaton into the hash table simultaneously, due to edges sharing endpoints. Similarly in EXTRACT-MAXIMAL-UNITIGS, though the threads examine disjoint vertex sets, multiple threads simultaneously constructing the same maximal unitig from its different constituent vertices can access the same automaton concurrently, at the walks' meeting vertex. CUTTLEFISH 2 maintains exclusive access to a vertex to one thread at a time through a sparse set $\mathcal{L}$ of locks. Each lock $l \in \mathcal{L}$ guards a disjoint set $\mathcal{V}_i$ of vertices, roughly of equal size. With p processor-threads and assuming all p threads accessing the hash table at the same time, the probability of two threads concurrently probing the same lock at the same turn is $\left(1 - (1 - 1/|\mathcal{L}|)^{\binom{p}{2}}\right)$—this is minuscule with an adequate $|\mathcal{L}|$. [14]

4.4. Asymptotics.

In this section, we analyze the computational complexity of the CUTTLEFISH 2$(\mathcal{R}, k, f_0)$ algorithm when executed on a set $\mathcal{R}$ of strings, given a k value, and a threshold factor f_0 for the edges in $G(\mathcal{R}, k)$. $\mathcal{E}$ is the set of the $(k+1)$-mers occurring $\geq f_0$ times in $\mathcal{R}$, and $\mathcal{V}$ is the set of the k-mers in $\mathcal{E}$. Let ℓ be the total length of the strings $r \in \mathcal{R}$, n be the vertex-count $|\mathcal{V}|$, and m be the edge-count $|\mathcal{E}|$.

4.4.1. Time complexity. CUTTLEFISH 2 represents j-mers with 64-bit machine-words—packing 32 symbols into a single word. Let w_j denote the number of words in a j-mer, i.e. $w_j = \lceil \frac{j}{32} \rceil$.

Note that the number of $(k+1)$-mers in $\mathcal{R}$ is upper-bounded by ℓ. The ENUMERATE-EDGES step uses the KMC 3 (71) algorithm. At first, it partitions the $(k+1)$-mers into buckets based on their signatures. With a rolling computation, determining the signature of a $(k+1)$-mer takes an amortized constant time. Assigning a $(k+1)$-mer to its bucket then takes time $\mathcal{O}(w_{k+1})$, and the complete distribution takes $\mathcal{O}(w_{k+1}\ell)$. [15] As each $(k+1)$-mer consists of w_{k+1} words, radix-sorting a bucket of size

[14] The optimal (in regard to probability) value $|\mathcal{L}| = |\mathcal{V}|$ is not used due to the locks' memory usage.

[15] This bound is not tight, as KMC 3 actually distributes sequences longer than $(k+1)$-mers—reducing computation. See Section 4.3.4.

B_i takes time $\mathcal{O}(B_i w_{k+1})$. So in the second step, for a total of b buckets for $\mathcal{R}$, the cumulative sorting time is $\sum_{i=1}^{b} \mathcal{O}(B_i w_{k+1}) = \mathcal{O}(w_{k+1} \sum_{i=1}^{b} B_i) = \mathcal{O}(w_{k+1}\ell)$. Thus ENUMERATE-EDGES takes time $\mathcal{O}(\ell w_{k+1})$.

The EXTRACT-VERTICES step applies KMC 3 (71) with $\mathcal{E}$ as input, and hence we perform a similar analysis as earlier. Each $e \in \mathcal{E}$ comprises two k-mers. So partitioning the k-mers takes time $\mathcal{O}(2mw_k)$, and radix-sorting the buckets takes $\mathcal{O}(w_k \sum B_i) = \mathcal{O}(2mw_k)$. Therefore EXTRACT-VERTICES takes time $\mathcal{O}(mw_k)$.

The CONSTRUCT-MINIMAL-PERFECT-HASH step applies the BBHASH (78) algorithm to construct an MPHF h over $\mathcal{V}$. It is a multi-pass algorithm—each pass i tries to assign final hash values to a subset $\mathcal{K}_i$ of keys. Making a bounded number of passes over sets $\mathcal{K}_i$ of keys—shrinking in size—it applies some classical hash h_i on the $x \in \mathcal{K}_i$ in each pass. For some $x \in \mathcal{K}_i$, iff $h_i(x)$ is free of hash collisions, then x is not propagated to $\mathcal{K}_{i+1}$. Provided that the h_i's are uniform and random, each key $v \in \mathcal{V}$ is hashed with the h_i's an expected $\mathcal{O}(e^{1/\gamma})$ times (78), an exponentially decaying function on the γ parameter. Given that h_i's are constant time on machine-words, computing $h_i(v)$ takes time $\mathcal{O}(w_k)$. Then the expected time to assign its final hash value to a $v \in \mathcal{V}$ is $H(k) = \mathcal{O}(w_k e^{1/\gamma})$. Therefore CONSTRUCT-MINIMAL-PERFECT-HASH takes an expected time $\mathcal{O}(nH(k))$. Note that, querying h, i.e. computing $h(v)$ also takes time $H(k)$, as the query algorithm is multi-pass and similar to the construction.

The COMPUTE-AUTOMATON-STATES step initializes the n automata with the state q_0, taking time $\mathcal{O}(n)$. Then for each edge $e \in \mathcal{E}$, it fetches its two endpoints' states from the hash table in time $2H(k)$, updating them if required. In total there are $2m$ hash accesses, and thus COMPUTE-AUTOMATON-STATES takes time $\mathcal{O}(n + mH(k))$.

The EXTRACT-MAXIMAL-UNITIGS step scans through each vertex $v \in \mathcal{V}$, and walks the entire maximal unitig p containing v. The state of each vertex in p is decoded to complete the walk—requiring $|p|$ hash table accesses, taking time $|p|H(k)$. If the flanking vertices of p are non-branching, then the walk also visits their neighboring vertices that are absent in p, at most once per each endpoint. Once extracted, all the vertices in p are marked so that p is not extracted again later on—this takes time $|p|H(k)$, and can actually be done in time $\mathcal{O}(|p|)$ by saving the hash values of the path vertices while constructing p. Thus traversing all the u_i's in the maximal unitigs set $\mathcal{U}$ takes time $\left(H(k) \sum\limits_{u_i \in \mathcal{U}} (|u_i| + 2) + \sum\limits_{u_i \in \mathcal{U}} |u_i|\right) = nH(k)$. $\sum_{u_i \in \mathcal{U}} |u_i|$ equates to n because the set of the maximal unitigs $\mathcal{U}$ forms a vertex decomposition of $G(\mathcal{R}, k)$ (47). Thus EXTRACT-MAXIMAL-UNITIGS takes time $\mathcal{O}(nH(k))$.

In the brief analysis for the last three steps, we do not include the time to read the edges $(\mathcal{O}(mw_{k+1}))$ and the vertices $(\mathcal{O}(nw_k))$ into memory, as they are subsumed by other terms.

Thus, CUTTLEFISH $2(\mathcal{R}, k, f_0)$ has an expected running time $\mathcal{O}(\ell w_{k+1} + mw_k + (n+m)H(k))$, where $w_j = \lceil \frac{j}{32} \rceil$, $H(k) = \mathcal{O}(w_k e^{1/\gamma})$, and $\gamma > 0$ is a constant. It is evident that the bottleneck is the initial ENUMERATE-EDGES step, and it asymptotically subsumes the running time.

4.4.2. Space complexity. Here, we analyze the working memory (i.e. RAM) required by the CUTTLEFISH 2 algorithm. The ENUMERATE-EDGES step with KMC 3 (71) can work within a bounded memory space. Its partitioning phase distributes input k-mers into disk bins, and the k-mers are kept in working memory within a total space limit S, before flushes to disk. Radix-sorting the bins are done through loading bins into memory with sizes within S, and larger bins are broken into sub-bins to facilitate bounded-memory sort. As we discuss below, the graph traversal steps require a fixed amount of memory, determined linearly by n. As n is not computed until the completion of EXTRACT-VERTICES, we approximate it within the KMC 3 algorithm (see Suppl. Sec. 2.1), and then bound the memory for the KMC 3 execution appropriately. The next step of EXTRACT-VERTICES is also performed similarly within the same memory-bound.

The CONSTRUCT-MINIMAL-PERFECT-HASH step with BBHASH (78) processes the key set $\mathcal{V}$ in fixed-sized chunks. Each pass i with key set $\mathcal{V}_i$ has a bit-array A_i to mark $h_i(v)$ for all the $v \in \mathcal{V}_i$, along with an additional bit-array C_i to detect the hash collisions. Both A_i and C_i have the size $\gamma|\mathcal{V}_i|$. The finally concatenated A_i's is the output data structure A for the algorithm, and some C_i is present only during the pass i. A has an expected size of $\gamma e^{1/\gamma} n$ bits (78). $|C_0| = \gamma|\mathcal{V}_0| = \gamma n$, and this is the largest collision array in the algorithm's lifetime. Thus, an expected loose upper-bound of the memory usage in this step is $\mathcal{O}(|A| + |C_0|) = \mathcal{O}((e^{1/\gamma} + 1)\gamma n)$ bits.

At this point in the algorithm, a hash table structure is set up for the automata. Together, the hash function h and the hash buckets collection S take an expected space of $(\gamma e^{1/\gamma} n + n\lceil \log_2 |\mathcal{Q}| \rceil) = (\gamma e^{1/\gamma} + 6)n$ bits.

The COMPUTE-AUTOMATON-STATES step scans the edges in $\mathcal{E}$ in fixed-sized chunks. For each $e \in \mathcal{E}$, it queries and updates the hash table for the endpoints of e as required. Similarly, the EXTRACT-MAXIMAL-UNITIGS step scans the vertices in $\mathcal{V}$ in fixed-sized chunks, and spells the containing maximal unitig of some $v \in \mathcal{V}$ through successively querying the hash table for the path vertices. The spelled paths are dumped to disk at a certain cumulative threshold size. Thus the only non-trivial memory usage by these steps is from the hash table. Therefore these graph traversal steps use $((\gamma e^{1/\gamma} + 6)n + \mathcal{O}(1))$ bits.

When $\gamma \leq 6$, the hash table (i.e. the hash function and the bucket collection) is the the dominant factor in the algorithm's memory usage, and CUTTLEFISH $2(\mathcal{R}, k, f_0)$ consumes expected space $\mathcal{O}((\gamma e^{1/\gamma} + 6)n)$. If $\gamma > 6$ is set, then it could be possible for the hash function construction memory to dominate. In practice, we adopt $\gamma = 2$, and the observed memory usage is $\approx 9.7n$ bits, translating to ≈ 1.2 bytes per distinct k-mer.

Funding
This work has been supported by the US National Institutes of Health (R01 HG009937) (RP), US National Science Foundation (CCF-1750472, and CNS-1763680) (RP), Poland National Science Centre (project DEC-2019/33/B/ST6/02040) (SD), and Faculty of Automatic Control, Electronics and

Computer Science at Silesian University of Technology (statutory research project 02/080/BKM_21/0020) (MK). The funders had no role in the design of the method, data analysis, decision to publish, or preparation of the manuscript.

Declaration of interests

RP is a co-founder of Ocean Genomics, inc.

References

1. U.S. National Library of Medicine. NCBI insights : The entire corpus of the sequence read archive (SRA) now live on two cloud platforms!, 2020.
2. Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, et al. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, July 2015. doi: 10.1371/journal.pbio.1002195.
3. Stephen Nayfach, Simon Roux, Rekha Seshadri, et al. A genomic catalog of earth's microbiomes. *Nature Biotechnology*, 39(4):499–509, April 2021. ISSN 1546-1696. doi: 10.1038/s41587-020-0718-6.
4. Dirk Gevers, Rob Knight, Joseph F. Petrosino, et al. The human microbiome project: A community resource for the healthy human microbiome. *PLOS Biology*, 10(8):1–5, August 2012. doi: 10.1371/journal.pbio.1001377.
5. Paul Muir, Shantao Li, Shaoke Lou, et al. The real cost of sequencing: scaling computation to keep pace with data generation. *Genome Biology*, 17(1):1–9, 2016.
6. N.G. de Bruijn. A combinatorial problem. *Nederl. Akad. Wetensch., Proc*, 49:758–764, 1946.
7. I. J. Good. Normal recurring decimals. *Journal of the London Mathematical Society*, s1-21 (3):167–169, 1946. doi: https://doi.org/10.1112/jlms/s1-21.3.167.
8. Jared T. Simpson and Mihai Pop. The theory and practice of genome sequence assembly. *Annual Review of Genomics and Human Genetics*, 16(1):153–172, 2015. doi: 10.1146/annurev-genom-090314-050032. PMID: 25939056.
9. Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001. ISSN 0027-8424. doi: 10.1073/pnas.171285098.
10. Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. Toward perfect reads: self-correction of short reads via mapping on de bruijn graphs. *Bioinformatics*, 36(5):1374–1381, February 2019. ISSN 1367-4803. doi: 10.1093/bioinformatics/btz102.
11. Leena Salmela and Eric Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, August 2014. ISSN 1367-4803. doi: 10.1093/bioinformatics/btu538.
12. Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, et al. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinformatics*, 16(1):288, September 2015. ISSN 1471-2105. doi: 10.1186/s12859-015-0709-7.
13. Zamin Iqbal, Mario Caccamo, Isaac Turner, et al. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, February 2012. ISSN 1546-1718. doi: 10.1038/ng.1028.
14. Daniel L. Cameron, Jan Schröder, Jocelyn Sietsma Penington, et al. GRIDSS: sensitive and specific genomic rearrangement detection using positional de Bruijn graph assembly. *Genome Research*, 27(12):2050–2060, December 2017. ISSN 1549-5469. doi: 10.1101/gr.222109.117. 29097403[pmid].
15. Fatemeh Almodaresi, Mohsen Zakeri, and Rob Patro. PuffAligner: a fast, efficient and accurate aligner based on the pufferfish index. *Bioinformatics*, June 2021. ISSN 1367-4803. doi: 10.1093/bioinformatics/btab408. btab408.
16. Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. deBGA: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32(21):3224–3232, July 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw371.
17. Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, et al. An incrementally updatable and scalable system for large-scale sequence search using LSM trees. *BioRxiv*, 2021. doi: 10.1101/2021.02.05.429839.
18. Yuzhen Ye and Haixu Tang. Utilizing de bruijn graph of metagenome assembly for metatranscriptome analysis. *Bioinformatics*, 32(7):1001–1008, August 2015. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv510.
19. Phelim Bradley, N. Claire Gordon, Timothy M. Walker, et al. Rapid antibiotic-resistance predictions from genome sequence data for staphylococcus aureus and mycobacterium tuberculosis. *Nature Communications*, 6(1):10063, December 2015. ISSN 2041-1723. doi: 10.1038/ncomms10063.
20. Mingjie Wang, Yuzhen Ye, and Haixu Tang. A de Bruijn graph approach to the quantification of closely-related genomes in a microbial community. *Journal of Computational Biology*, 19 (6):814–825, 2012. doi: 10.1089/cmb.2012.0058. PMID: 22697249.
21. Yu Peng, Henry C. M. Leung, Siu-Ming Yiu, et al. IDBA-tran: a more robust de novo de bruijn graph assembler for transcriptomes with uneven expression levels. *Bioinformatics*, 29 (13):i326–i334, June 2013. ISSN 1367-4803. doi: 10.1093/bioinformatics/btt219.
22. Manfred G. Grabherr, Brian J. Haas, Moran Yassour, et al. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nature Biotechnology*, 29(7):644–652, July 2011. ISSN 1546-1696. doi: 10.1038/nbt.1883.
23. Nicolas L. Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525–527, May 2016. ISSN 1546-1696. doi: 10.1038/nbt.3519.
24. Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10): 958–968.e6, 2021. ISSN 2405-4712. doi: https://doi.org/10.1016/j.cels.2021.08.009.
25. Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *Nature Methods*, 17(2):155–158, February 2020. ISSN 1548-7105. doi: 10.1038/s41592-019-0669-3.
26. Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, et al. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016. ISSN 0027-8424. doi: 10.1073/pnas.1604560113.
27. Shaun D. Jackman, Benjamin P. Vandervalk, Hamid Mohamadi, et al. ABySS 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome Research*, 27(5):768–777, May 2017. ISSN 1549-5469. doi: 10.1101/gr.214346.116. 28232478[pmid].
28. Dinghua Li, Chi-Man Liu, Ruibang Luo, et al. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics*, 31(10):1674–1676, January 2015. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv033.
29. Xiang Li, Qian Shi, and Mingfu Shao. On bridging paired-end RNA-seq data. *BioRxiv*, 2021. doi: 10.1101/2021.02.26.433113.
30. C. Titus Brown, Dominik Moritz, Michael P. O'Brien, et al. Exploring neighborhoods in large metagenome assembly graphs using spacegraphcats reveals hidden sequence diversity. *Genome Biology*, 21(1):164, July 2020. ISSN 1474-760X. doi: 10.1186/s13059-020-02066-4.
31. Laurent David, Riccardo Vicedomini, Hugues Richard, and Alessandra Carbone. Targeted domain assembly for fast functional profiling of metagenomic datasets with S3A. *Bioinformatics*, 36(13):3975–3981, April 2020. ISSN 1367-4803. doi: 10.1093/bioinformatics/btaa272.
32. Sven D. Schrinner, Rebecca Serra Mari, Jana Ebler, et al. Haplotype threading: accurate polyploid phasing from long reads. *Genome Biology*, 21(1):252, September 2020. ISSN 1474-760X. doi: 10.1186/s13059-020-02158-1.
33. Bo Liu, Yadong Liu, Junyi Li, et al. deSALT: fast and accurate long transcriptomic read alignment with de Bruijn graph-based index. *Genome Biology*, 20(1):274, December 2019. ISSN 1474-760X. doi: 10.1186/s13059-019-1895-9.
34. Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. *Nature Communications*, 11(1):6327, December 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-19777-8.
35. Ilia Minkin and Paul Medvedev. Scalable pairwise whole-genome homology mapping of long genomes with BubbZ. *IScience*, 23(6):101224, 2020. ISSN 2589-0042. doi: https://doi.org/10.1016/j.isci.2020.101224.
36. Hélène Lopez-Maestre, Lilia Brinza, Camille Marchet, et al. SNP calling from RNA-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence. *Nucleic Acids Research*, 44(19):e148–e148, July 2016. ISSN 0305-1048. doi: 10.1093/nar/gkw655.
37. Gustavo AT Sacomoto, Janice Kielbassa, Rayan Chikhi, et al. KIS SPLICE: de-novo calling alternative splicing events from RNA-seq data. *BMC Bioinformatics*, 13(6):S5, April 2012. ISSN 1471-2105. doi: 10.1186/1471-2105-13-S6-S5.
38. Kadir Dede and Enno Ohlebusch. Dynamic construction of pan-genome subgraphs. *Open Computer Science*, 10(1):82–96, 2020. doi: doi:10.1515/comp-2020-0018.
39. John A. Lees, T. Tien Mai, Marco Galardini, et al. Improved prediction of bacterial genotype-phenotype associations using interpretable pangenome-spanning regressions. *MBio*, 11(4): e01344–20, 2020. doi: 10.1128/mBio.01344-20.
40. Roland Wittler. Alignment- and reference-free phylogenomics with colored de Bruijn graphs. *Algorithms for Molecular Biology*, 15(1):4, April 2020. ISSN 1748-7188. doi: 10.1186/s13015-020-00164-3.
41. Alan Cleary, Thiruvarangan Ramaraj, Indika Kahanda, et al. Exploring frequented regions in pan-genomic graphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(5):1424–1435, 2019. doi: 10.1109/TCBB.2018.2864564.
42. Buwani Manuweera, Joann Mudge, Indika Kahanda, et al. Pangenome-wide association studies with frequented regions. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '19, page 627–632, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366663. doi: 10.1145/3307339.3343478.
43. Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, et al. PanTools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):i487–i493, August 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw455.
44. Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1): i177–i186, July 2021. ISSN 1367-4803. doi: 10.1093/bioinformatics/btab309.
45. Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome Biology*, 21(1):249, September 2020. ISSN 1474-760X. doi: 10.1186/s13059-020-02135-8.
46. H. Guo, Y. Fu, Y. Gao, et al. deGSM: memory scalable construction of large scale de bruijn graph. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Early Access: 1–1, 2019. ISSN 1557-9964.
47. Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, June 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw279.
48. 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, September 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw609.
49. Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform. *Bioinformatics*, 32(4):497–504, October 2015. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv603.
50. Rayan Chikhi, Antoine Limasset, Shaun Jackman, et al. On the representation of de bruijn graphs. In Roded Sharan, editor, *Research in Computational Molecular Biology*, pages 35–55, Cham, 2014. Springer International Publishing. ISBN 978-3-319-05269-4.
51. Shoshana Marcus, Hayan Lee, and Michael C. Schatz. SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, November 2014. ISSN 1367-4803. doi: 10.1093/bioinformatics/btu756.
52. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 321455363.
53. Camille Marchet, Mael Kerbiriou, and Antoine Limasset. BLight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, April 2021. ISSN 1367-4803. doi: 10.1093/bioinformatics/btab217.
54. Camille Marchet, Zamin Iqbal, Daniel Gautheret, et al. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1): i177–i185, July 2020. ISSN 1367-4803. doi: 10.1093/bioinformatics/btaa487.

55. Amatur Rahman and Paul Medvedev. Representation of k-mer sets using spectrum-preserving string sets. In Russell Schwartz, editor, *Research in Computational Molecular Biology*, pages 152–168, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45257-5.
56. Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*, 22(1):96, April 2021. ISSN 1474-760X. doi: 10.1186/s13059-021-02297-z.
57. Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent a set of k-long DNA sequences. *ACM Comput. Surv.*, 54(1), March 2021. ISSN 0360-0300. doi: 10.1145/3445967.
58. Justin M. Zook, David Catoe, Jennifer McDaniel, et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(1):160025, June 2016. ISSN 2052-4463. doi: 10.1038/sdata.2016.25.
59. Joan Mas-Lloret, Mireia Obón-Santacana, Gemma Ibáñez-Sanz, et al. Gut microbiome diversity detected by high-coverage 16S and shotgun sequencing of paired stool and colon sample. *Scientific Data*, 7(1):92, March 2020. ISSN 2052-4463. doi: 10.1038/s41597-020-0427-5.
60. Adina Chuang Howe, Janet K. Jansson, Stephanie A. Malfatti, et al. Tackling soil diversity with the assembly of large, complex metagenomes. *Proceedings of the National Academy of Sciences*, 111(13):4904–4909, 2014. ISSN 0027-8424. doi: 10.1073/pnas.1402564111.
61. Inanc Birol, Anthony Raymond, Shaun D. Jackman, et al. Assembling the 20 gb white spruce (picea glauca) genome from whole-genome shotgun sequencing data. *Bioinformatics*, 29 (12):1492–1497, May 2013. ISSN 1367-4803. doi: 10.1093/bioinformatics/btt178.
62. Burton H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692.
63. Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011. ISSN 1367-4803. doi: 10.1093/bioinformatics/btr011.
64. Liang Zhao, Jin Xie, Lin Bai, et al. Mining statistically-solid k-mers for accurate NGS error correction. *BMC Genomics*, 19(10):912, December 2018. ISSN 1471-2164. doi: 10.1186/s12864-018-5272-y.
65. Pranvera Hiseni, Knut Rudi, Robert C. Wilson, et al. HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. *Microbiome*, 9(1):165, July 2021. ISSN 2049-2618. doi: 10.1186/s40168-021-01114-w.
66. Grace A. Blackwell, Martin Hunt, Kerri M. Malone, et al. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):1–16, November 2021. doi: 10.1371/journal.pbio.3001421.
67. Jun Yoshimura, Kazuki Ichikawa, Massa J. Shoura, et al. Recompleting the caenorhabditis elegans genome. *Genome Research*, 29(6):1009–1022, June 2019. ISSN 1549-5469. doi: 10.1101/gr.244830.118. 31123080[pmid].
68. Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, 2013.
69. Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, June 2013. ISSN 1367-4803. doi: 10.1093/bioinformatics/btt310.
70. Sungmin Lee, Hyeyoung Min, and Sungroh Yoon. Will solid-state drives accelerate your bioinformatics? in-depth profiling, performance analysis and beyond. *Briefings in Bioinformatics*, 17(4):713–727, September 2015. ISSN 1467-5463. doi: 10.1093/bib/bbv073.
71. Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, May 2017. ISSN 1367-4803. doi: 10.1093/bioinformatics/btx304.
72. Michael Roberts, Wayne Hayes, Brian R. Hunt, et al. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, July 2004. ISSN 1367-4803. doi: 10.1093/bioinformatics/bth408.
73. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, Digital Equipment Corp., 1994.
74. Jonathan Gross and Jay Yellen. *Graph Theory and Its Applications*. CRC Press, Inc., USA, 1999. ISBN 849339820.
75. Jon Kleinberg and Eva Tardos. Graphs. In *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005. ISBN 321295358.
76. Xiaotu Ma, Ying Shao, Liqing Tian, et al. Analysis of error profiles in deep next-generation sequencing data. *Genome Biology*, 20(1):50, March 2019. ISSN 1474-760X. doi: 10.1186/s13059-019-1659-6.
77. Marek Kokot, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. Sorting data on ultra-large scale with RADULS. In Stanisław Kozielski, Dariusz Mrozek, Paweł Kasprowski, et al., editors, *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, pages 235–245, Cham, 2017. Springer International Publishing. ISBN 978-3-319-58274-0.
78. Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.SEA.2017.25.
79. Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984. doi: 10.1137/0605009.
80. Guillaume Marçais. Compact vector: Bit packed vector of integral values, 2020. https://github.com/gmarcais/compact_vector, Accessed on June 18, 2020.