SUPPLEMENTAL FILES

Supplemental files outline The Supplemental files are divided in two parts, namely, details on the methods and supplementary benchmarks. Some explanations will be given in the text below, and we provide concrete examples in Supplementary Boxes that can be found in the end of the document. First, following the main text's organization, details on relevant k-mer structures (hash-based, BWT-based) and compression are given (Supplemental Boxes S1-4). Then, we give more insights about some of the set of k-mer sets approaches. In particular, we provide a lower-level description of structures/features that did not appear in the main document. E.g., examples of the BFT and RAMBO structures are given, as well as comparisons between specific approaches (Supplemental Boxes S5-8). Complexities are outlined in Supplemental Table S3. In the Supplemental Benchmark section, we provide the full benchmarks (Supplemental Tables S1 and S2) extracted from the different papers that led to Table 3 in the main document.

Details on the methods..

k-mer index data structures..

Bloom filters, CQF and Othello hashing Examples are presented in Supplemental Box S1.

De Bruijn graph and compacted de Bruijn graph Instances are shown in Supplemental Box S2.

BOSS: BWT-based De Bruijn Graphs The Burrows Wheeler Transform is a text transformation algorithm. It receives a sequence as input, and rearranges its characters in a way that enhances further compression. The transformation is reversible, thus the original sequence can be decoded. BOSS rearranges k-mers to represent the De Bruijn graph in a similar way.
Here, we briefly show how the BOSS scheme works. To begin we describe the following simple — but not space-efficient — representation of a DBG: take each unique $(k+1)$-mer, consisting of a vertex concatenated to the label of an outgoing edge, and sort those $(k+1)$-mers according to their first k symbols taken in reverse order. The resulting sorted list contains all nodes and their adjacent edges sorted such that all outgoing and incoming nodes of a given node can be identified. Thus, it is a working representation, in that all graph operations can be performed, but is far from space efficient since $(k+1)$ symbols need to be stored for each edge. Next, we show that we can essentially ignore the first k symbols, which will lead to a substantial reduction in the total size of the data structure.
First, we make a small alteration to this simple representation by padding the graph to ensure every vertex has an incoming path made of at least k vertices, as well as an outgoing edge. This maintains the fact that a vertex is defined by its previous k edges. For example, say k-mer `CCATA` has no incoming edge; then we add a vertex `$CCAT` and an edge between `$CCAT` and `CCATA`, then between `$$CCA` and `$CCAT`, and so forth. We let W be the last column of the sorted list of $(k+1)$-mers. Next, we flag some of the edges in the representation with a minus symbol to disambiguate edges incoming into the same vertex – which we accomplish by adding a minus symbol to the corresponding symbols in W. Hence, W is a vector of symbols from `{A, C, G, T, $, -A, -C, -G, -T, -$}`. Next, we add a bit vector L which represents whether an edge is the last edge, in W, exiting a given vertex. This means that each node will have a sequence of zero-or-more 0-bits followed by a single 1-bit, e.g., if there is only a single edge outgoing from a node then there is a single 1-bit for that edge. Overall the representation consists of a vector of symbols (W), a bit vector (L) implemented using a rank/select Raman et al. (2002) data structure, and finally an array that records the counts of each character. It may seem surprising but these three vectors provide enough information for representing the DBG and supporting traversal operations. We refer the reader to the original paper for a detailed discussion. Lastly, we note that this representation, which is referred to as BOSS, is due to Bowe et al. Bowe et al. (2012) and was extended for storing colors Muggli et al. (2017) (see Supplemental Box S3 for an example).

Details on compression. To efficiently represent a $n \times c$ color matrix, over n k-mers across c datasets, different schemes have been proposed. A color class is a set of colors common to one or multiple k-mers. It can also be seen as a bit vector, or alternatively, a row of the color matrix. Supplemental Box S4 presents examples of the different techniques: the delta-based encoding used in Mantis+MST (a), the RRR/Elias-Fano coding (b) used e.g. in Mantis and VARI, the lossy compression using BF from Metannot (c), the BRWT principle (d), and the three strategies used in SeqOthello (e).

Set of k-mer sets details.

Color aggregative methods. We first show how the different color aggregative methods combine k-mer sets, indexing techniques and color strategies in Supplemental Box S6.
BFT significantly differs from other methods: an example is shown in Supplemental Box S5. In a BFT, k-mers are divided into a prefix and a suffix part that are recorded in a burst trie. Prefixes are further divided into chunks, which are to be inserted into the root or inner nodes of the tree. Suffixes are in the leaves. Queries start at the tree root and progress through the path that spell the query string. In practice, each leaf stores a set of tuples: some k-mer suffixes along with their corresponding color classes. Bloom filters are also used in the inner nodes, to increase query speed by quickly checking the presence of a chunk.

Tool	Data Processing Time (days)	Max Ext. Memory (GB)	Time (h, wallclock)	Peak RAM (GB)	Index Size (GB)
SBT	3.5^b	300^a	55^b	25^b	200^a
AllSomeSBT	3.5^a	600^a	25^a	35^b	140^a
SSBT	3.5^a	600^a	55^a	5^b	20^a
HowDeSBT	2.5^a	30^a	10^a	N/A	15^a
Mantis	130^a	110^d	20^a	N/A	30^a
SeqOthello	3.5^b	190^b	2^b	15^b	20^b
BIGSI	N/A	N/A	N/A	N/A	145^c

Supplemental Table S1. Space and time requirements to build human RNA-seq indices. The best result for each column is shown in green. ^{a} refers to the HowDeSBT article, ^{b} to the SeqOthello article, ^{c} to the BIGSI article, and ^{d} was obtained through personal correspondence with R. Patro.

Note that the above description of BFT does not capture the full complexity of the data structure, and should only be used to build an initial intuition.

k-mer aggregative methods In Supplemental Box S7, we present indexing and query details, in a similar fashion than Box 3 in the main text, but more in depth. We show the index construction and query steps in SBT, BIGSI, and show how COBS improves on BIGSI's representation while keeping the core idea.
The false positive rate of a BF is monotonically increasing in m/n, where m is the number of k-mers in the dataset and n is the number of bits in the BF. BIGSI uses the same size n for all the BFs, thus the false positive rates of the BFs differ depending on how many k-mers are in the corresponding dataset. COBS avoid this by storing BFs of size adapted to the corresponding dataset.
Then we illustrate contrasts between the k-mer-aggregative methods in Supplemental Box S8. For the different flavors of SBTs, different strategies are used to store information in each node. Supplemental Box S8 shows the improvements in bit-vector representation first brought by SSBT/AllSomeSBT, then by HowDeSBT. In a second Figure, BIGSI, Dream Yara and RAMBO strategies for indexing Bloom filters are compared. In the following, we outline the very recent RAMBO's method.
An example of RAMBO structure is shown in Supplemental Box S8, bottom right panel. RAMBO builds a matrix of C columns and T rows. Cells of the matrix are BFs. At construction, a given dataset is assigned to one cell per column. The corresponding BFs in those cells are each updated so that all the k-mers of the dataset are inserted into each of those BFs. This creates some (necessary) redundancy in the structure. Since several datasets can be assigned to a same cell, BFs become union BFs by informing for the presence/absence of k-mers in more than one dataset. A query is performed on the rows, each union BF giving a row-wise union of sets where the query could be present. The final sets containing the query are deduced by performing an intersection of the different set unions.

Supplementary benchmarks. In Table S1, we show the results of different methods on a collection of 2585 of human blood, brain and breast RNA-seq datasets. This collection was first used in the original SBT paper and became a *de facto* benchmark for the other methods. It contains approximately 4 billion distinct 20-mers. Each reviewed article had its own, different, set of methods for performing a benchmark using this dataset. Here we assembled the results of three different benchmarks performed in 2018 and 2019, for which the important parameters (k value, abundance threshold) were identical. Even if hardware settings where different in the three studies, the presented trends (in particular, impacts on disk and RAM) remain accurate. The data processing time column refers to the time necessary to convert the original sequence files to the k-mer set indices (computation of Bloom filters, CQF with Squeakr, Othello). The maximum external memory column corresponds to the peak disk usage when building the index. The time column is the time required to build the set of k-mer sets index. The index size column is the final index size. BIGSI is not a compressed index, but the authors had explored the possibility to compress using snappy (`https://google.github.io/snappy/`). Parameters used for the different methods were $\theta = 0.9$ and BF size of 2.10^9 for the SBT methods, $k = 20$ as the k-mer size for all methods, and 34 "log slots" for Mantis from the estimation of their paper.

In Table S2 we present the space and time required to build those data structures on bacterial datasets. The table is divided into two sub-tables that correspond to two benchmarks from the literature. Contrary to the human RNA-seq experiments, these two benchmarks were not reconcilable, in terms of used datasets and parameters, thus we chose to present them separately. The first one shows results from Bingmann et al. (2019), containing 1,000 bacterial, viral and parasitic whole-genome DNA files, obtained from the BIGSI paper (`http://ftp.ebi.ac.uk/pub/software/bigsi/nat_biotech_2018/ctx/`). The second one is from Muggli et al. (2019) and contains 4,000 datasets 16,000 Salmonella strains (NCBI BioProject PRJNA183844).

Tool	Max Ext. Memory (GB)	Time (h, wall clock)	Peak RAM (GB)	Index Size (GB)
Table (a)				
SBT	N/A	1.9	11	20
SSBT	N/A	8.0	1.5	3.3
AllsomeSBT	N/A	2.0	7.1	21
HowDeSBT	N/A	21	108	1.9
BIGSI	N/A	1.2	247	28
COBS	N/A	0.01	2.6	3.0
SeqOthello	N/A	0.7	12	4.4
Mantis	N/A	0.4	88	16
Table (b)				
Vari	1,000	11	136	51
Vari-Merge	1,000	12	52	51
Rainbowfish	1,000	11	136	51
BFT	900	52	120	99
Multi-BRWT	1,300	42	156	1,300
Mantis+MST	36	12	52	51

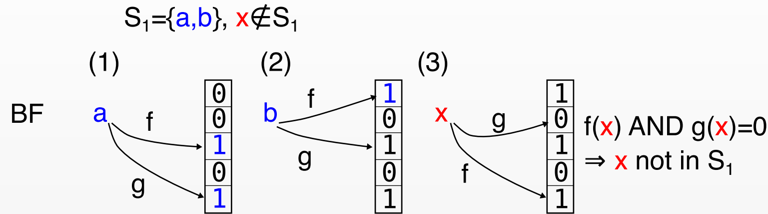
Supplemental Table S2. Space and time required to build indices on bacterial datasets. **Table (a).** The table shows results from a benchmark done in the COBS article Bingmann et al. (2019). The COBS benchmark contains 1,000 microbial DNA files, consisting of various bacterial, viral and parasitic WGS datasets (in the ENA as of December 2016) with an average of 3.4 million distinct k-mers per file. No cutoff on k-mer abundances was used before constructing the data structures. In the COBS benchmark, k was set to 31. In the table, COBS denotes for the COBS *compact* index that allows more than one batches of BFs. **Table (b)** shows results from the Vari-Merge article Muggli et al. (2019) The Vari-Merge benchmark contains 4,000 datasets totalling 1.1 billion distinct k-mers from 16,000 Salmonella strains. Note that it has more genomes than the COBS benchmark, but it possibly contains a lower variability in k-mer content. In the Vari-Merge benchmark, methods were run with $k = 32$, with the exception of BFT that was run with $k = 27$. When applicable, other parameters (for both Table (a) and (b)) were set to their defaults.

	construction	query
SBT	$\mathcal{O}(n \times b)^*$	$\mathcal{O}(Q \times h)$
VARI	$\mathcal{O}(N \times \log(N))$	$\mathcal{O}(Q \times n)$
Mantis	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
SeqOthello	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
BFT	$\mathcal{O}(N \times n)$	$\mathcal{O}(Q \times n)$
BIGSI	$\mathcal{O}(n \times b)$	$\mathcal{O}(Q \times n)$
RAMBO	unreported	$\mathcal{O}(Q \times \sqrt{n} \times log(n))$

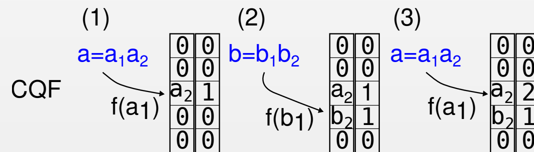
Supplemental Table S3. Time complexities for the construction and query of the main approaches. N is the total number of distinct k-mers, n is the number of datasets, Q the query size (number of k-mers). We denote by b the number of bits in a Bloom filter, and h the number of datasets that contain at least θ% of Q k-mers. We consider k as a relatively small constant (around 21-63). * This time complexity is derived from Theorem 1 in Harris and Medvedev (2020) with the assumption that the size of the Bloom filter b is roughly $\mathcal{O}(N/n)$. Note that there may be an additional complexity cost for building the topology of the tree through clustering. We note that in the worst case (majority of k-mers present in all datasets), the query complexities of SBTs would be $\mathcal{O}(Q \times n)$.

Supplemental Box S1. Hashing techniques

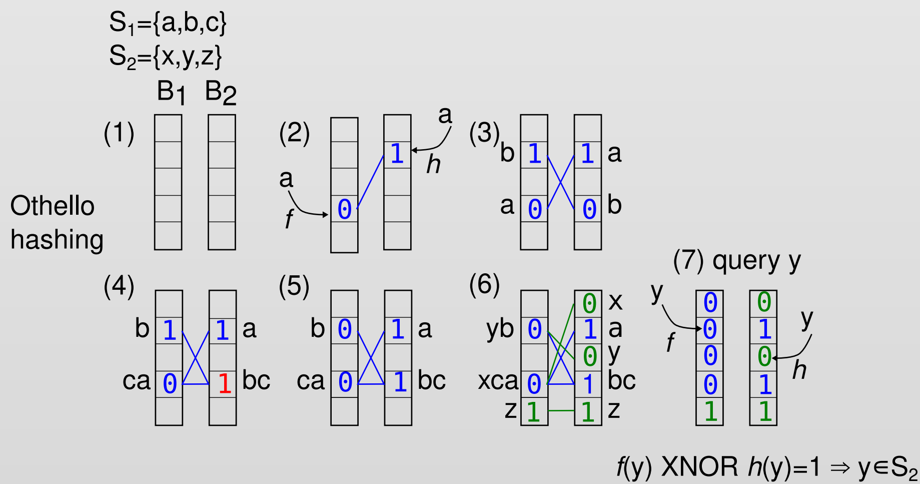
Bloom filters The example filter has a set of two functions f and g. In (1) a is inserted by putting 1s at positions 2 and 4 indicated by both functions. (2) b is inserted similarly. (3) x is queried, $g(x)$ giving a 0 we are certain that it is not present in the filter.



Counting Quotient filter intuition Element a and b are decomposed into a_1a_2 and b_1b_2. a_1, b_1 are quotients, and a_2, b_2 are remainders. (1) During a's insertion, the quotient is used to find the position of the element in the filter, and a_2 is stored. The count is associated (second column). (2) similar operation for b. (3) a is re-inserted, leading to a count of 2.

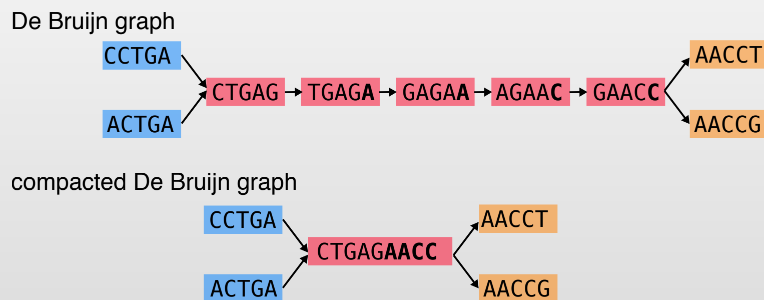


Othello Hashing intuition In the example below (figure), we will focus on the case where two sets S_1 and S_2 are hashed. A larger number of sets can be dealt with. Othello hashing uses two hash functions, denoted here by f and h. The method maintains two arrays B_1 and B_2, see panel (1). In the case of storing two sets, B_1 and B_2 are binary arrays. Elements from S_1 will be mapped to one value x_1 in B_1 and another value x_2 in B_2 such that $x_1 \neq x_2$. Conversely, elements from S_2 will correspond to identical values ($(x_1, x_2) = (0,0)$ or $(1,1)$). In (2), the element a from S_1 is hashed with f and inserted in B_1 at the position given by f and similarly with h and B_2. A different value will be stored in B_1 and in B_2 (0 and 1). The lines between those two values visually represent their association to a. (3) b is hashed the same way than a, ensuring again that two different values are associated to b. (4) Element c is inserted, here we cannot ensure two different values are associated to c without having a contradiction. Thus b's 0 in B_2 is modified (in red). (5) The values associated to b must differ, so in B_1 we modify the 1 associated with b to a 0. (6) x, y, z are inserted, this time they must be associated to pairs of identical elements as they belong to S_2. (7) y is queried by hashing it with f and h and by checking if the associated values are identical (y in S_2) or different (y in S_1).



Supplemental Box S2. De Bruijn graphs

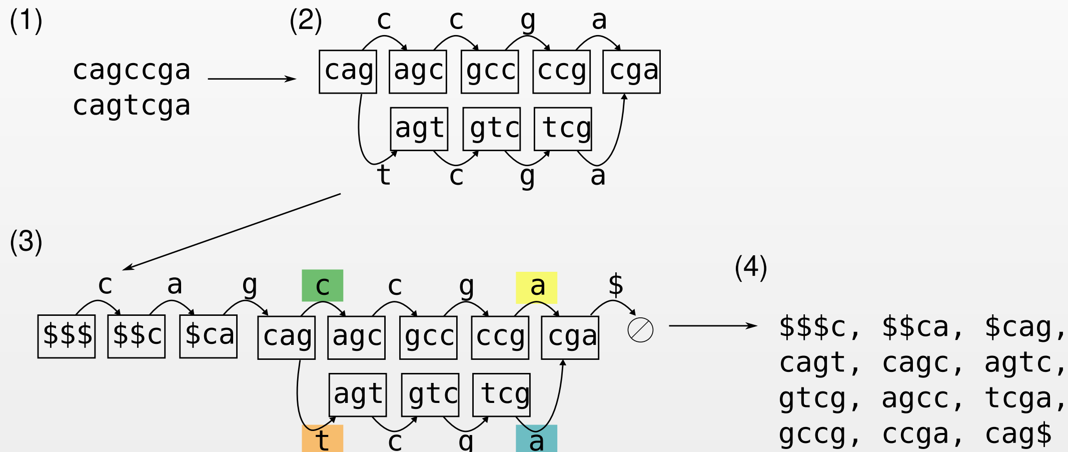
In the example below, the first graph is a regular De Bruijn graph from the 5-mers CCTGA, ACTGA, CTGAG, TGAGA, GAGAA, AGAAC, GAACC, AACCT, AACCG. The node CTGAG has two ingoing edges and only one outgoing, GAACC has two outgoing edges and only one in-going, any other vertex in-between connecting CTGAG and GAACC has only one in-going and outgoing edge. Thus this red path can be compacted.
The second graph is the resultant compacted De Bruijn graph. The red path becomes a single red node, by concatenating CTGAG, **A**, **A**, **C** and **C**. It keeps the same connections than the two flanking nodes. Each resultant node is referred to as a unitig.



For representing the nodes, the first representation uses 5×9 nucleotides, and the compacted representation only uses $5 \times 4 + 9$ nucleotides.

Supplemental Box S3. BOSS graph structure

We describe the BOSS data structure, as per its original flavor Bowe et al. (2012). We build a BOSS from two sequences CAGCCGA and CAGTCGA with $k = 3$. Part (2) is the De Bruijn graph from these sequences (no reverse-complements are considered here). In this representation, each vertex contains a 3-mer, and an edge represents a 4-mer existing in the original sequences, the label of the edge being the last nucleotide of this 4-mer. (3) represents the same information, but with the constraint that any nodes not containing $ must be preceded by k vertices (3 vertices) and must have at least an outgoing edge. Thus supplementary nodes with the padding '$' symbol are added. (4) is the list of $(k+1)$-mers in the (3) graph.



(5)

	nodes	edges labels
0	$$$	c
1	$ca	g
2	cga	$
3	$$c	a
4	gcc	g
5	agc	c
6	gtc	g
7	cag	c
8	cag	t
9	ccg	a
10	tcg	a
11	agt	c

(6)

0	$$$	c	1
1	$ca	g	1
2	cga	$	1
3	$$c	a	1
4	gcc	g	1
5	agc	c	1
6	gtc	g	1
7	cag	c	0
8	cag	t	1
9	ccg	a	1
10	tcg	a-	1
11	agt	c	1

(7)

0	$$**$**	c	1
1	$c**a**	g	1
2	cga	$	1
3	$$**c**	a	1
4	gcc	g	1
5	agc	c	1
6	gtc	g	1
7	ca**g**	c	0
8	cag	t	1
9	ccg	a	1
10	tcg	a-	1
11	ag**t**	c	1

(8)

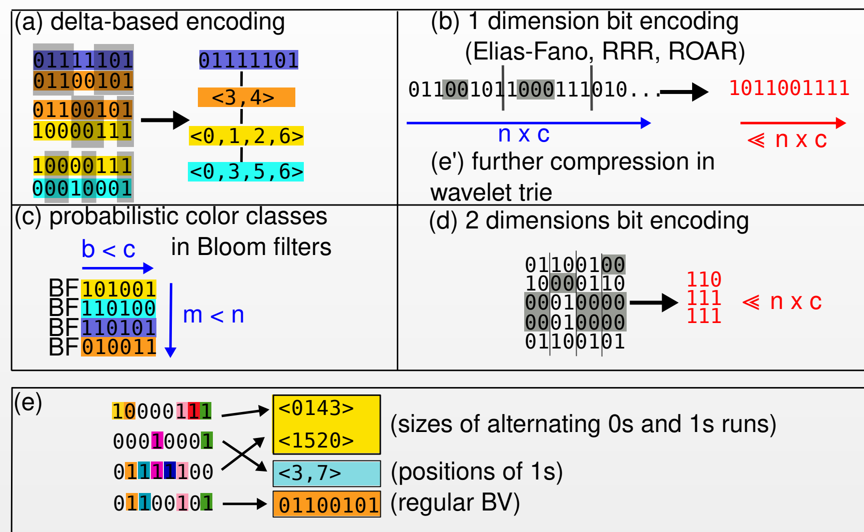
0	c	1
1	g	1
2	$	1
3	a	1
4	g	1
5	c	1
6	g	1
7	c	0
8	t	1
9	a	1
10	a-	1
11	c	1

$	**0**
a	**1**
c	**3**
g	**7**
t	**11**

(5) These $(k+1)$-mers are listed by lexicographic order by reading them in reverse starting from the kth nucleotide to the first (ties are broken by the $k+1$-th nucleotide). This gives a matrix of nucleotides, the last nucleotide of each $(k+1)$-mer being in a separate red column. Each line of the matrix represents a node label in the graph, and the red vector represents the edge labels. (6) In order to denote nodes that have several outgoing edges, a new vector (blue) is used. 1s indicate the last occurrence of a given node, while 0s mark its previous occurrences (they are necessarily contiguous). Here node CAG has two outgoing edges, one labeled by C (green, marked 0) and the other by T (orange, marked 1). Several edges entering the same node share the same label, all but one are marked using a $-$, as for yellow/blue labels. (7) Only the last column of the matrix will be kept in the BOSS. We retain the rank of each first symbol (in red): $ appears at rank 0, A at rank 1, C at rank 3, ... (8) The final information in the BOSS structure. From these tables, DBG operations such as going forward, backward from a given node are shown to be possible in Bowe et al. (2012), but we do not describe them here.

Supplemental Box S4. Details on compression of bit matrices

Color classes can be further compressed. We present here some of the known techniques.

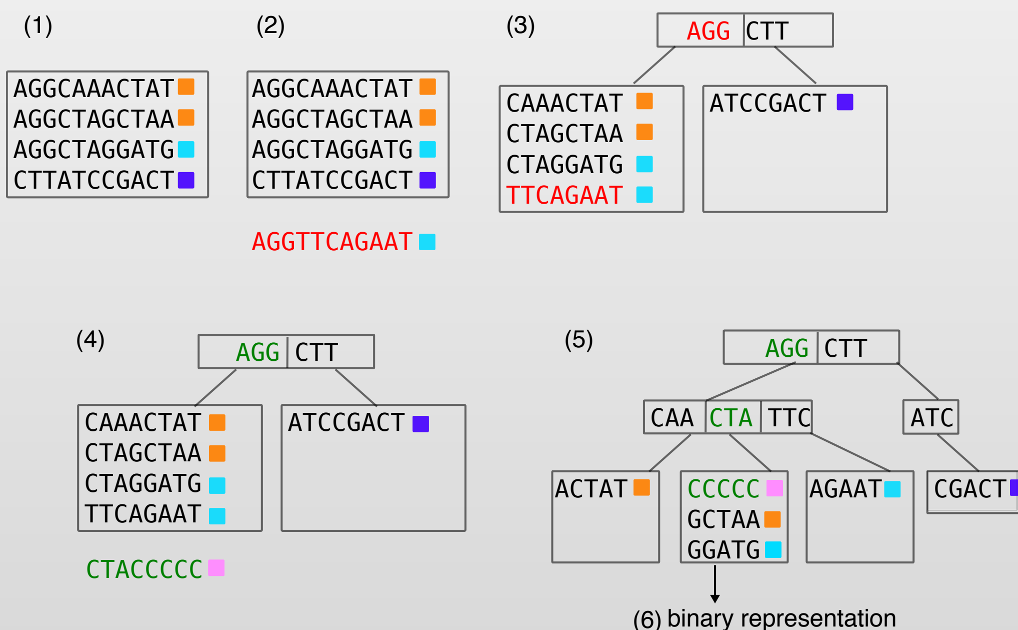


Delta-based encoding (a): Differences between rows in the matrix are encoded. One can e.g. encode the (column-wise) differences between the current row and the first row as 1's, and similarities with 0's. This results in a sparser matrix that can be further compressed, e.g. using (b). In Figure (a), grey zones mark similarities between pairs of vectors. The purple vector is chosen as a reference, and positions of differences with the orange, yellow and blue vectors are recorded into separate lists. Mantis + MST uses this technique, and it also one of SeqOthello's strategies.
Bit encoding techniques (b,d): When encoding a color matrix: rows of the color matrix are first concatenated. The resulting bit vector is then compressed losslessly into a shorter bit vector. In (b), rows of a color matrix (not displayed) are concatenated then compressed by, e.g., finding runs of 0s (denoted in grey) and yields a smaller vector (red vector in the figure). Many tools implement this idea (Vari, Mantis, Rainbowfish, BiFrost, the SBTs). In (d), rectangular blocks of 0s (denoted in grey) in the color matrix are marked and removed. This allows 2-dimensional compression (red matrix), by storing the positions of removed blocks. This solution was proposed by Multi-BWRT.
Using (e), such representations can be further compressed.
Probabilistic color classes (c): In Metannot, instead of recording the exact presence/absence of a k-mer within each dataset, colors are stored in a Bloom filter of size $b < c$. Then, the retrieval of color(s) associated with a k-mer becomes probabilistic (i.e., a color may be wrongly given to a k-mer).
Approximate color classes with hybrid compression (e) Nearly identical rows are grouped, and a representative is chosen for each group. Then, depending on whether bit-vectors associated to the color classes are sparse (small amount of 1s) or dense (high amount of 1s), different compression schemes are used. SeqOthello and BiFrost use this strategy (the example shows SeqOthello's solution). SeqOthello proposes to group similar color profiles, then uses a suitable compression technique depending on the bit-sparsity of each group. A list of integers represents the bit-vector when it has a few 1s (integers are the positions of the 1s). With many 1s, run-length encoding alternatively encodes the consecutive number of 0s and 1s. If the bit-vector has roughly the same amount of 0s and 1s, no compression is used. BiFrost differs a bit, by adapting different bit-encoding techniques as (b) to the different vector sparsities.

Supplemental Box S5. Bloom filter trie

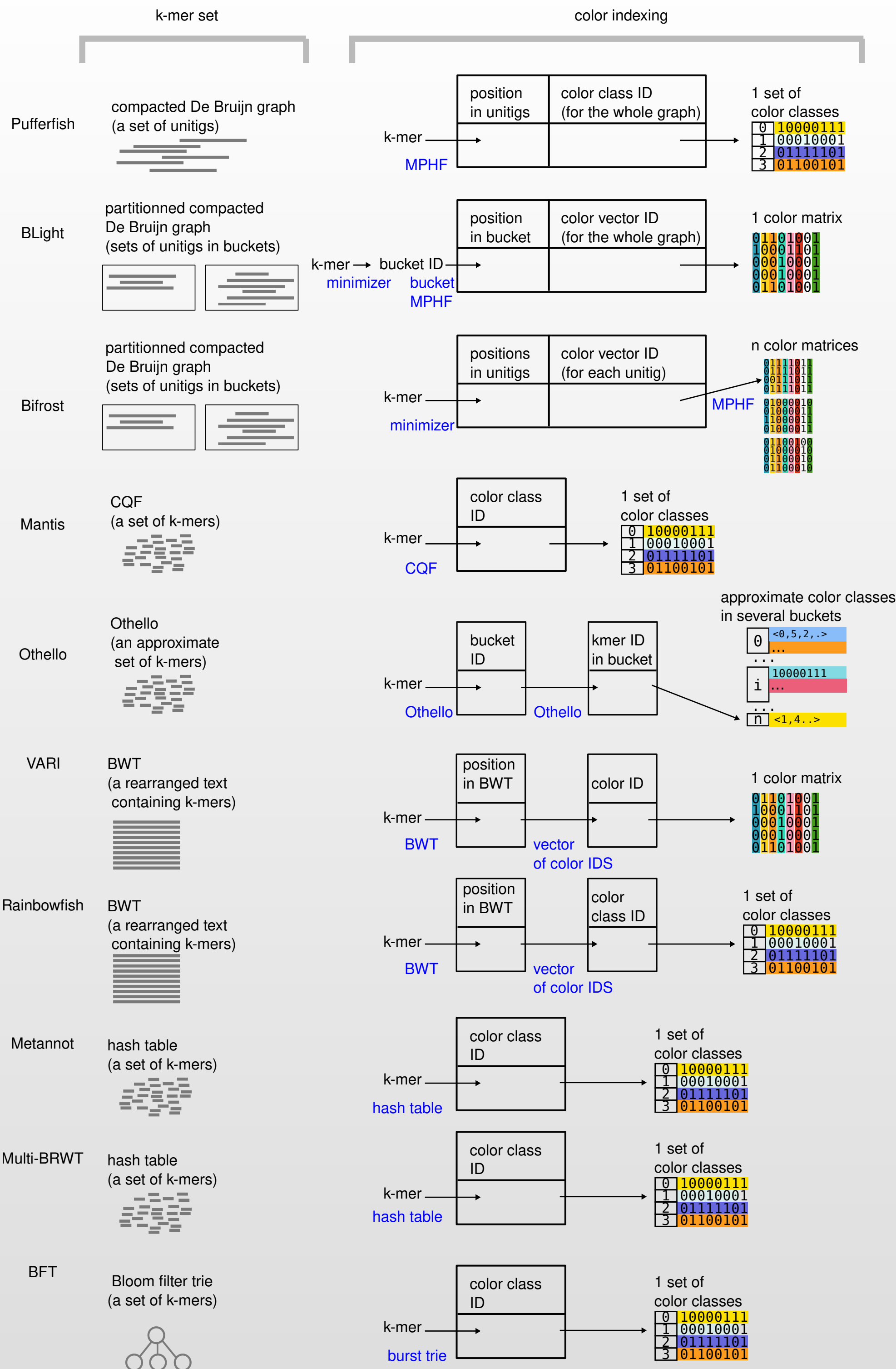
A Bloom filter trie is a tree that stores k-mers in its leaves. A leaf can store at most t k-mer, otherwise it is "burst" (i.e. transformed) into a sub-tree. The new sub-tree consists of a node v and two or more children of v. All prefixes of length p from the sequences in the original leaf are stored in v. All suffixes of length $k-p$ that follow the i-th prefix in v are stored in the i-th child of v.

We now show how to store the following k-mers in a BFT: AGGCTAGCTAA, AGGCAAACTAT, AGGCTAGGATG, CTTATCCGACT, AGGTTCAGAAT, AGGCTACCCCC, with $t=4$ and $p=3$. In Panel (1), the first four k-mers can be inserted in a single leaf, since $t=4$. (2) The fifth k-mer AGGTTCAGAAT (red) cannot be inserted in the leaf, requiring a burst operation. (3) To perform the burst, the prefixes of size p of the five k-mers are stored in the root. Each prefix has a pointer to its corresponding subtree. Suffixes of length $k-p$ are stored in the leaves. (4) Inserting AGGCTACCCCC (green), it is put in the left leaf as its prefix is AGG. This induces a burst of the left leaf which is performed on Panel (5). (6) The tree is not represented explicitly, but instead, binary vectors are introduced to optimize for space. In addition Bloom filters are added in intermediate nodes to speed up queries. Note: k-mers are stored as tuples with their color class.



Supplemental Box S6. Details on building blocks in color aggregative methods.

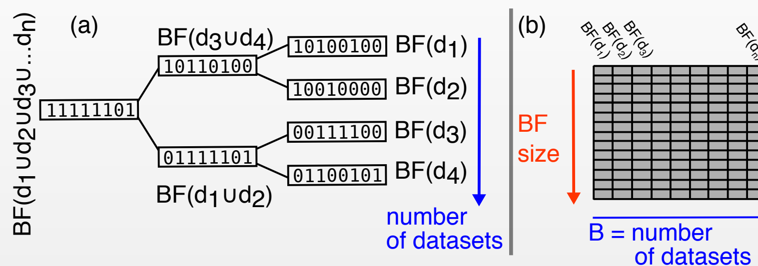
Here we give details on the color aggregative methods strategies, and in particular the k-mer set implementations.



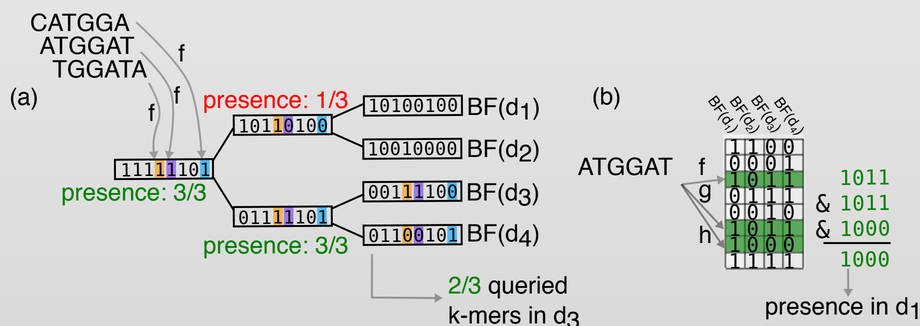
In the Figure above, the terms **compacted DBG, unitigs, BWT and MPHF** are defined in the main text. Some of the techniques use **minimizers**. While there exist multiple definitions in the literature, here a minimizer is the smallest l-mer that appears within a k-mer, with $l < k$. "Smallest" should be understood in terms of lexicographical order. For example in the k-mer GAACT, the minimizer of size 3 is AAC, as all other l-mers (GAA, ACT) are higher in the lexicographical order. Minimizers are used here to create partitions of k-mers.Such partitioning techniques reduce the footprint of position encoding.

Supplemental Box S7. k-mer aggregative methods

Index construction In SBT (Panel (a) below), all BFs in leaves are have the same length, which is related to the estimated total number of distinct k-mers to index. The union BFs of intermediate nodes in the tree are constructed by applying a logical OR on BFs of the children BFs.
In BIGSI (Panel (b)), all BFs must also have the same size. COBS uses the same principle but Bloom filters do not all have the same size.

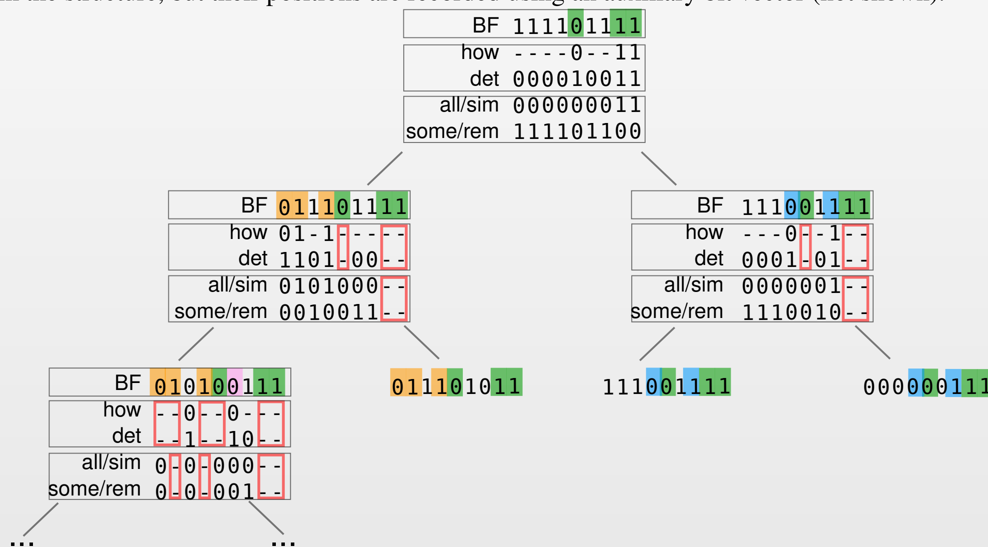


Queries Here we show in more details how queries are performed in SBTs and BIGSI (see Box 3 in the main text for a more abstract sketch). In the Figure below (left figure), we consider a single hash function f, as it was presented in HowDeSBT's paper for instance, and $\theta = 2/3$. For BIGSI (right figure), we present the query step for one k-mer and three hash functions (f, g, h). Note that usual queries are composed of more than one k-mer and aggregate the k-mers results. A given k-mer is hashed, leading to one or several rows to lookup. In Figure (b) below, the query is performed on the green rows. Each queried row informs on the datasets that may contain the query k-mer. All the returned bit vectors are then summarized vertically into a single vector, using a logical AND operation. Positions yielding 1s after this operation correspond to datasets that contain the k-mer (in the Figure example, the k-mer is present only in dataset 1). (c) The same principle is used when matrices of several sizes contain the BFs. Hashes are simply adapted to the different sizes using a modulo.



Supplemental Box S8. Advanced and alternative k-mer aggregative methods

Comparison between SBT approaches For SBTs, different strategies are used to store information in each node. In the example below, the first level of each node is a plain Bloom filter, as used in the original SBT approach. The second level is the $how + det$ representation used in HowDeSBT. The third is the equivalent $all + some$ or $sim + rem$ representations used by AllSomeSBT and SSBT. The three approaches are shown in four nodes.
At some positions (marked in green), the bits will have the same value across all the nodes of the subtree. Those bits are marked as det (determined), and when they are, the how field records their values. In the $sim + rem$ and $all + some$ systems, such bits will have a value set to 1 in all/sim in the root node if and only if they are determined as 1. In this node, the $some/rem$ vector stores values such that $all \cup some = BF$ or $sim \cup rem = BF$, where BF is the Bloom filter of the node.
At the second level of the tree, new bits become determined (orange in the left subtree and blue in the right subtree). The same rules apply. Moreover, bits that were marked in the upper levels are non informative (red boxes). They can be removed from the structure, but their positions are recorded using an auxiliary bit vector (not shown).



Comparison between Bloom filter approaches Bloom filters are grey panels in all figures below. Left: BIGSI and COBS differences reside in the column sizes. COBS queries are as fast as BIGSI, using a system of modulos with the different BF sizes. Middle: the DREAM-Yara index is built by interleaving the bits of each Bloom filters. Bits of the same rank are grouped together in bins of size n. Right: each element in the RAMBO matrix is a Bloom filter (each column of the matrix stacks complete BFs). Contrary to SBTs, RAMBO merges randomly the datasets. Queries in SBTs are top-bottom, RAMBO queries each row and uses the intersection result.