University of Verona

Department of Computer Science

Doctoral Program in Computer Science

GGCAT: an optimized tool for fast compacted and colored de Bruijn graph construction

Andrea Cracco

Advisor

Prof. Romeo Rizzi

Coordinator

Prof. Umberto Castellani

Cycle XXXVIII — S.S.D. INFO-01/A

In loving memory of my grandfather Lelio Cracco

Abstract

De Bruijn graphs are a widely used data structure in bioinformatics, often serving as the basis for more complex analyses. In this thesis, we present a tool called GGCAT and its updated version GGCAT 2 to compute compacted and optionally colored de Bruijn graphs for DNA datasets. GGCAT outperforms other state-of-the-art tools available for this task in time, memory, and temporary disk space usage on many different kinds of data, from pangenomes to sequencing reads. This is due to a set of optimized algorithms designed to reduce both runtime and memory usage. GGCAT 2 is a major overhaul of the first version, with improvements in all performance metrics, and in particular a large reduction in the required temporary disk space through a novel compaction technique. It supports multiple representations of de Bruijn graphs, from classical maximal unitigs to the more compact simplitigs and minimum-size eulertigs, all with custom algorithms able to handle extremely large datasets. The tool is open source and available on GitHub.

Contents

1 Introduction

A *de Bruijn graph* of order k is a directed graph in which each node represents a string of length k (a *k-mer*), and there is a directed edge from a node u to a node v whenever the length-$(k-1)$ suffix of u equals the length-$(k-1)$ prefix of v. In bioinformatics, one typically builds a de Bruijn graph whose nodes are exactly the k-mers present in a set of DNA sequences, making it a compact representation of the sequence content of the data (see Section 1.3.3 for formal definitions). A key derived object is a *maximal unitig*: a maximal path in the graph whose internal nodes each have exactly one incoming and one outgoing edge, representing a variation-free region of the input sequences. The string spelling such a path is stored instead of the full path, yielding a compacted de Bruijn graph.

De Bruijn graphs are appealing for several reasons. First, by increasing the abundance threshold, one can have a very simple but effective method of filtering out sequencing errors (i.e., erroneous k-mers). Second, having a graph structure allows for a smaller representation of the data in the presence of repeated regions, since equal substrings are represented only once in the graph. Third, by focusing on maximal unitigs, one can discover *variation-free* regions.

Originally, maximal unitigs were introduced for the genome assembly problem; for example, assembly contigs are usually unitigs of a corrected assembly graph [1, 2]. However, by replacing each maximal unitig path with a single edge labeled by the string spelled by that unitig, one gets an equivalent graph of much smaller size. Such a graph is usually called a *compacted de Bruijn graph*. Since the first and the last k-mer of every maximal unitig is a node of the compacted de Bruijn graph, it often suffices to compute only the strings labeling the maximal unitigs of a de Bruijn graph, not the actual graph structure. Moreover, reverse complements need special handling, as they are treated as equivalent to their forward counterpart. Many applications use a compacted graph because it has equivalent representation power but significantly smaller size. In fact, if one just wants to represent the k-mers of a dataset in plain text form, there exist more efficient equivalent representations [3, 4, 5], and even minimum-size representations such as eulertigs [6] and matchtigs [7], but all of these use maximal unitigs as a starting point.

A popular variant of a de Bruijn graph is the *colored* de Bruijn graph, originally introduced for de novo assembly and genotyping of variants [8]. Such a graph is built from a *collection* of datasets, for example different sequencing datasets or different (full) genome sequences. For every k-mer, colored de Bruijn graphs also store the identifiers (*colors*) of the datasets in which the k-mer appears. One can think of a

colored de Bruijn graph as a compressed representation of the k-mers in a *collection* of datasets, but which retains enough information (i.e., the color of every k-mer) in order to identify each dataset in this combined graph. Later applications include pangenomics [9], RNA-seq quantification [10], bacterial genome querying [11], alignment and reference-free phylogenomics [12], microbiome research [13], just to name a few. Colored de Bruijn graphs also underpin a number of state-of-the-art genomic indexes for efficient pseudoalignment and querying of large genome collections, such as Themisto [14], Fulgor [15], and MetaGraph [16].

While computing the maximal unitigs of a de Bruijn graph can be done with a simple linear-time algorithm, the practical hardness of the problem stems from the fact that for most datasets the initial de Bruijn graph does not fit into main memory. Practical tools computing compacted de Bruijn graphs have to cleverly use the disk to store intermediate data, partition the data in order to use many CPU cores efficiently, and minimize CPU, RAM and I/O bottlenecks. The first tool that made efficient use of minimizer-based bucketing for computing maximal unitigs was BCALM2 [17]. BCALM2 first does a k-mer counting step inspired by KMC2 [18] and a filtering pass based on the multiplicity of the k-mers. Then it finds k-mers that should be joined together by bucketing them on their left/right minimizers [19] (corresponding to the minimizers of the leftmost and rightmost $(k-1)$-mers). Each bucket is processed independently and in parallel to find all the possible extensions and as a last step BCALM2 glues the unitigs that were produced in different buckets together using a union-find data structure. In this thesis we propose a tool to produce a compacted and optionally colored de Bruijn graph, partially inspired by the main algorithmic ideas from BCALM2, that brings many novel algorithms and optimizations to handle extremely large collections of genomes.

1.1 Structure of the thesis

This thesis is divided into four chapters:

- This first chapter introduces the basic concepts that are needed to understand the rest of the thesis, such as de Bruijn graphs, their compacted representations as well as some algorithmic techniques used in GGCAT.
- The second chapter explains the details and implementation of the first version of GGCAT, focusing on the novel algorithmic techniques that make it faster and more memory efficient than previous tools.
- The third chapter focuses on the improvements made in the second version of GGCAT, analyzing the differences with respect to the first version and the new algorithms introduced.
- The fourth chapter is a broader analysis of the project architecture of GGCAT, with a focus on the applied software engineering practices that guarantee code quality, maintainability and correctness of the implementation.

Some parts of the first chapter and most of the second chapter are adapted from [20].

1.2 Notation

We now define some operators that will be used in this thesis. Throughout this thesis, Σ denotes the DNA alphabet $\{A, C, G, T\}$. We use Σ^k to denote the set of all strings of length k over Σ, and Σ^* for the set of all strings of any length over Σ. We denote concatenation of two strings x and y as $x \cdot y$. If x is a substring of y, we write $x \in y$. We denote the length of a string x as $|x|$. Given a string x of length at least k, we denote by $\mathsf{pre}_{k(x)}$ the prefix of x of length k, and by $\mathsf{suf}_{k(x)}$ the suffix of x of length k. For two strings x and y such that $\mathsf{suf}_{k(x)} = \mathsf{pre}_{k(y)}$, we denote by $x \odot^k y$ the string $x \cdot \mathsf{suf}_{|y|-k}(y)$ (the *merge* of x and y). Given a set or multiset R of strings, we define $\mathsf{ends}_{k(R)} = \left\{\mathsf{pre}_{k(x)}, \mathsf{suf}_{k(x)} : x \in R\right\}$. Given a multiset R of strings, and a string x, we denote by $\mathsf{occ}(x, R)$ the number of occurrences of x in the strings of R, with different occurrences within the same string counted individually. Given a k-mer $q \in \Sigma^k$, we denote by $q^{-1} \in \Sigma^k$ the reverse complement of q (written q^{-1} in the typeset formulas). Given a multiset R of strings, and a string x, if $x \neq x^{-1}$, we define $\mathsf{occ}_{cn}(x, R) = \mathsf{occ}(x, R) + \mathsf{occ}(x^{-1}, R)$, otherwise $\mathsf{occ}_{cn}(x, R) = \mathsf{occ}(x, R)$. We analogously define the *appear* function $\mathsf{app}(x, R) = \min(1, \mathsf{occ}(x, R))$ and the canonical appear function $\mathsf{app}_{cn}(x, R) = \min(1, \mathsf{occ}_{cn}(x, R))$.

1.3 Key concepts

In this section we introduce some key definitions and concepts that will be useful to understand the rest of the thesis.

1.3.1 k-mers

A *k-mer* is a string of a given length k over the alphabet Σ. Given a k-mer q, we say that q is a k-mer of a string x if $q \in x$ (in this case, we also say that q *appears*, or *occurs* in x). Given a set or a multiset R of strings, we say that q *appears* in R, and write $q \in R$, if q appears in some string in R.

As an example, the string $ACGAACCGGG$ with $k = 4$ has the following k-mers:

- $ACGA$
- $CGAA$
- $GAAC$
- $AACC$
- $ACCG$
- $CCGG$
- $CGGG$

1.3.2 Reverse complements and canonical k-mers

In DNA sequences, every nucleotide has a complementary base, with A being complementary to T, and C being complementary to G. The *reverse complement* of a DNA sequence is obtained by reversing the sequence and replacing every nucleotide

with its complementary base. For example, the reverse complement of the sequence "GGCAT" is "ATGCC". While dealing with genomic data, it is often unknown whether a sequence comes from the forward strand or the reverse strand of the DNA molecule. To handle this uncertainty, de Bruijn graph-based tools often consider both a k-mer and its reverse complement as equivalent entities. The most used approach to achieve this is by using *canonical k-mers*. A canonical k-mer is defined as the smallest according to a specific ordering (either lexicographic or hash-based) between a k-mer and its reverse complement. By using canonical k-mers, tools can effectively treat both strands of DNA equally, simplifying the representation and analysis of genomic data in de Bruijn graphs. If k is odd, it is guaranteed that a k-mer and its reverse complement are different, because the complement of the middle character can never be equal to itself and in the reversed order it always occupies the same position. However, if k is even, some k-mers are equal to their reverse complement, for example "ATAT" is equal to its reverse complement "ATAT". This can lead to some ambiguities, as no clear orientation can be assigned to these k-mers. In GGCAT both odd and even length canonical k-mers are supported, and in the output only one orientation of every k-mer is reported.

1.3.3 de Bruijn graphs

De Bruijn graphs are a data structure useful for representing overlaps between sequences of symbols. Invented independently by both de Bruijn [21] and I. J. Good [22] in the 1940s, they have found widespread application in various fields, including bioinformatics. A (complete) de Bruijn graph of order k over an alphabet of size n is a directed graph with n^k nodes where each node is labeled with a unique string of length k (called a k-mer) and there is a directed edge from a node v to a node u iff the $(k-1)$-suffix of v is equal to the $(k-1)$-prefix of u. An example of a complete de Bruijn graph is shown in Figure 1.

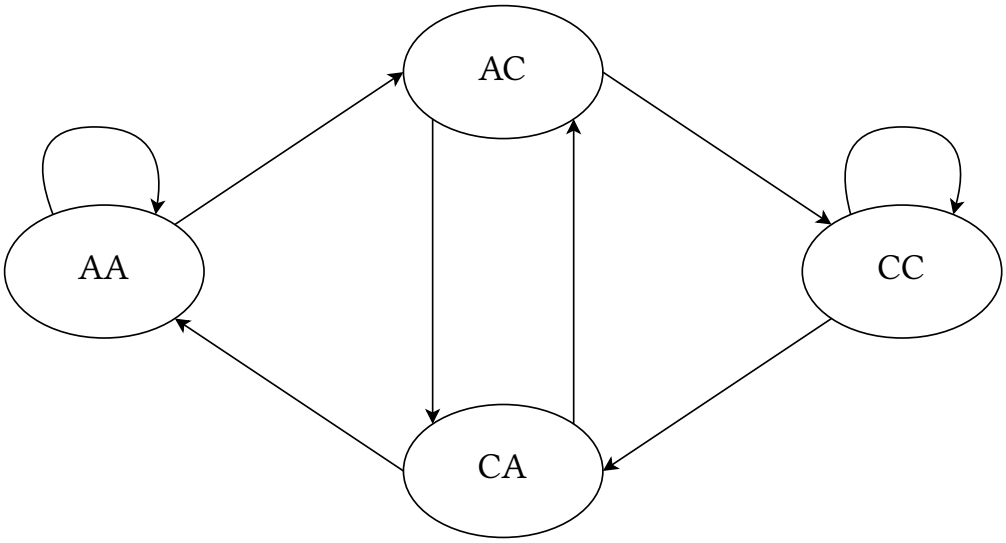

Figure 1: Example of a complete de Bruijn graph of order 2 on the alphabet $\{A, C\}$.

In most applications a de Bruijn graph is not complete, that is it has only a subset of the possible k-mers as nodes. This allows the creation of de Bruijn graphs that represent the k-mers contained in an input set S of sequences. An example of a non-complete de Bruijn graph is shown in Figure 2.

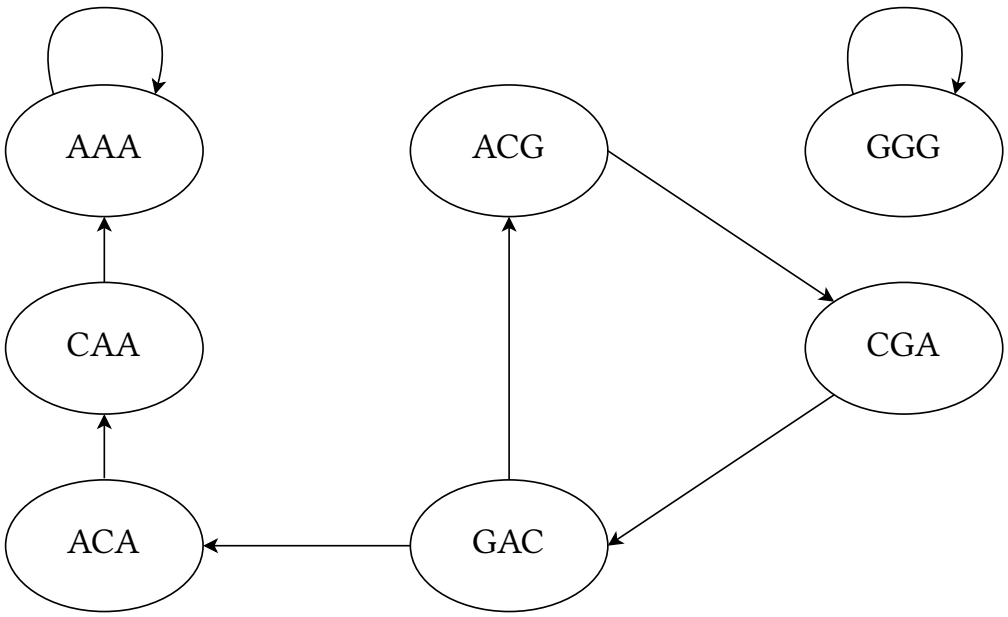

Figure 2: Example of a non-complete de Bruijn graph of order 3 on the subset of sequences $\{ACG, CGA, GAC, ACA, CAA, AAA, GGG\}$.

In the above example, the de Bruijn graph has only 7 nodes, and every possible edge satisfying the de Bruijn graph definition is present. Note that it is not guaranteed anymore that the graph is connected, nor that every node has at least one incoming or outgoing edge.

1.3.4 de Bruijn graphs in bioinformatics

In their non-complete version, de Bruijn graphs are one of the most fundamental data structures in computational genomics, appearing in countless applications, for example, assembly and analysis of sequencing data [23], RNA-seq data [24], read error correction [25, 26], alignment [27], compression [28], and rearrangement detection [29], just to name a few.

Equivalently, in the edge-centric convention introduced in Section 1.3.3, to build a de Bruijn graph of order k for a multiset of DNA sequences, for every k-mer in the input dataset strings one adds an edge from the node corresponding to its prefix of length $k-1$, to the node corresponding to its suffix of length $k-1$. To account for the desired sequencing depth and for error detection, de Bruijn graphs built on DNA sequences usually also have an associated *abundance* threshold a, so that edges (and thus nodes) are added only for k-mers appearing at least a times in the input strings.

1.3.5 Branching nodes and maximal unitigs representation

There is a 1-to-1 correspondence between a de Bruijn graph and a k-mer set: the graph can be fully reconstructed from its node labels alone, since edges are implicitly defined by prefix/suffix overlaps of length $k-1$. De Bruijn graphs can be represented in many ways, and one of the simplest is by listing all the k-mers of the nodes of the graph, with the edges being implicitly defined. While this representation is simple, it is not optimal from a space point of view: k-mers of two adjacent nodes overlap on $k-1$ equal characters, so storing both separately wastes $k-1$ characters. If we could collapse these two nodes into a single string, we would store only $k+1$ characters instead of $2k$. To choose which nodes to collapse there are various strategies, that lead to representations with different properties. The most used representation is obtained by finding non-branching parts of the graph called *unitigs*, built by collapsing two nodes u and v connected by an edge iff the edge between them is the only outgoing edge of u and the only incoming edge of v.

By collapsing all the possible edges we obtain *maximal unitigs*, which represent *variation-free* regions of the input sequences. Figure 3 highlights the edges that can be collapsed from the graph in Figure 2.

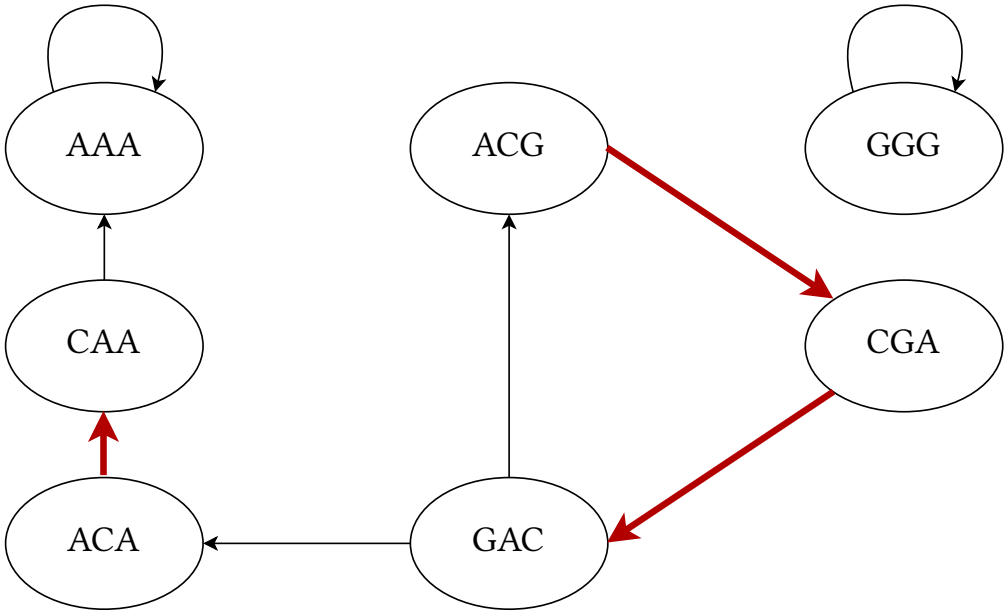

Figure 3: Unitig collapsed edges in the graph from Figure 2 are highlighted.

From this set of highlighted edges, we can build the graph using as nodes the set of maximal unitigs, as shown in Figure 4.

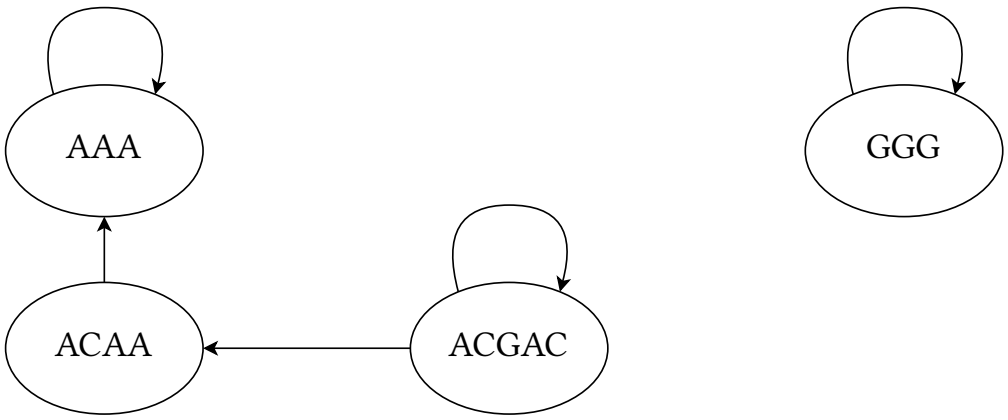

Figure 4: Compacted graph built from collapsing the edges in Figure 3, giving the set of maximal unitigs $M = \{GGG, AAA, ACGAC, ACAA\}$.

Note that as shown in Figure 4, maximal unitigs can be circular, i.e., they can have loop edges in the graph. In this case they are called *circular unitigs*. For a circular unitig of length l, there are $l - k + 1$ different ways of representing it as a string, one for each possible starting k-mer. For the scope of this thesis and in the GGCAT output, we consider all the rotations of a circular unitig as equivalent representations.

In practice this maximal unitigs representation is much more space efficient than the naive one, as many parts of a genome are variation-free. The efficiency of this representation can already be noticed in the proposed example, as it uses only 15 characters instead of the 21 needed to store all the k-mers.

1.3.6 Maximal unitigs string-based definition

For the rest of the thesis, we consider an alternative definition of maximal unitigs that does not explicitly use a de Bruijn graph. This has several advantages: it connects to the recent literature on *spectrum preserving string sets* (unitigs being one such type of sets) [3, 4, 7], it naturally extends to reverse complements without introducing heavy definitions related to bidirected de Bruijn graphs, and ultimately it matches the GGCAT algorithm, which proceeds bottom-up by iteratively merging k-mers and unitigs as long as possible (i.e., the existence of branches in the de Bruijn graph is checked implicitly via k-mer queries).

Given multisets of strings R and U, we say that R and U have the same *k-mer set* if any k-mer that appears in one of the sets also appears in the other set. Analogously, we say that R and U have the same *canonical k-mer set* if any k-mer q that appears in one of the sets, q or q^{-1} appears in the other set. We can equivalently express the fact that sets R and U have the same non-canonical k-mer set with the condition:

$$\forall q \in \Sigma^k \ \mathsf{occ}(q, R) \geq 1 \text{ iff } \mathsf{occ}(q, U) \geq 1.$$

Likewise, R and U have the same *canonical k-mer set* if:

$$\forall q \in \Sigma^k \ \mathsf{occ}_{cn}(q, R) \geq 1 \text{ iff } \mathsf{occ}_{cn}(q, U) \geq 1.$$

We now give an equivalent *string-centric* definition of the *set* U of maximal unitigs of a multiset R of strings, under our formalism. As a warm-up, we start with the case when we do not have reverse complements.

First, we require that all strings in the *set* U have length at least k, meaning unitigs contain at least one edge. Second, we require that R and U have the same k-mer set. Third, if a $(k-1)$-mer appears at least two times in U, then it cannot be an internal node in any unitig. In other words, we forbid merging two separate unitigs at a branching $(k-1)$-mer, since such branching $(k-1)$-mer must appear in at least one other string in U:

$$\forall q \in \Sigma^{k-1} \text{ if } \mathsf{occ}(q, U) > 1 \text{ then } q \text{ appears only as prefix or suffix of strings in } U.$$

Note that the above property also ensures that no two equivalent cyclic unitigs are in U.

To also impose maximality, we state that a $(k-1)$-mer is a prefix or a suffix of a unitig if and only if it is either branching, a sink, or a source:

$$\forall q \in \mathsf{ends}_{k-1}(U), \sum_{c \in \Sigma} \mathsf{app}(q \cdot c, U) \neq 1 \text{ or } \sum_{c \in \Sigma} \mathsf{app}(c \cdot q, U) \neq 1.$$

1.3.7 Canonical maximal unitigs

Having defined maximal unitigs without reverse complements, we now give the string-centric definition of maximal unitigs also assuming reverse complements (which we call *canonical maximal unitigs*). In fact, we give a more general one, also handling a required abundance threshold of the k-mers in R, further underlining the flexibility of our string-centric view.

Definition 1.1 (Canonical maximal unitigs):

Given a multiset R of strings and integers $k \geq 2$ and $a \geq 1$, we say that a *set* U of strings is the set of *canonical maximal unitigs* of R with k-mer size k and abundance threshold a if the following conditions hold:

0. $\forall q \in U, |q| \geq k$, and $\forall q, q' \in U, q \neq q'^{-1}$ (note that $q \neq q'$ is guaranteed by the fact that U is a set);
1. $\forall q \in \Sigma^k \ \mathsf{occ}_{cn}(q, R) \geq a$ iff $\mathsf{occ}_{cn}(q, U) \geq 1$ (same canonical k-mer set after applying the abundance threshold);
2. $\forall q \in \Sigma^{k-1}$, if $\mathsf{occ}_{cn}(q, U) > 1$ then q and q^{-1} appear only as prefix or suffix of strings in U (unitigs do not span over branching $(k-1)$-mers);

3. $\forall q \in \mathit{ends}_{k-1}(U), \sum_{c\in\Sigma} \mathit{app}_{cn}(q \cdot c, U) \neq 1$ or $\sum_{c\in\Sigma} \mathit{app}_{cn}(c \cdot q, U) \neq 1$ (maximality).

1.3.8 Spectrum-preserving string sets

Other than maximal unitigs, GGCAT natively supports other SPSS-based representations of the de Bruijn graph. Formally, a spectrum-preserving string set (SPSS) [3] is a set of strings that contains exactly the same set of k-mers as the original input set of strings. In other words, sliding a window of length k over all the strings in the SPSS yields the same set of k-mers as sliding the same window over the original input strings. The maximal unitigs representation is an example of SPSS for the labels of the uncompressed de Bruijn graph, as every node label (of length k) appears as a substring of exactly one maximal unitig. However, maximal unitigs are not the only possible SPSS. Other representations that can be even smaller are described in the next sections, with some of them being space-optimal under some constraints. There is interest in having smaller SPSS representations, as they can be used to represent the same set of k-mers more efficiently.

Simplitigs

Simplitigs are a way of representing SPSS introduced by [4]. They extend the number of edges of the de Bruijn graph that can be collapsed, by removing the constraint that only edges between non-branching nodes can be collapsed. Simplitigs are built by greedily collapsing edges in the graph until no more edges can be collapsed. Note that when an edge e between two nodes u and v is collapsed, the other outgoing edges of u and incoming edges of v cannot be collapsed again, so by the local choices made there are multiple possible equivalent simplitig representations for the same set of k-mers, that can potentially vary in size and number of strings. Taking again the example from Figure 2, the graph can be collapsed on one edge among the two orange edges in Figure 5, leading to two possible different simplitig representations: $\{GGG, ACGAC, ACAAA\}$ or $\{GGG, ACGACAAA\}$.

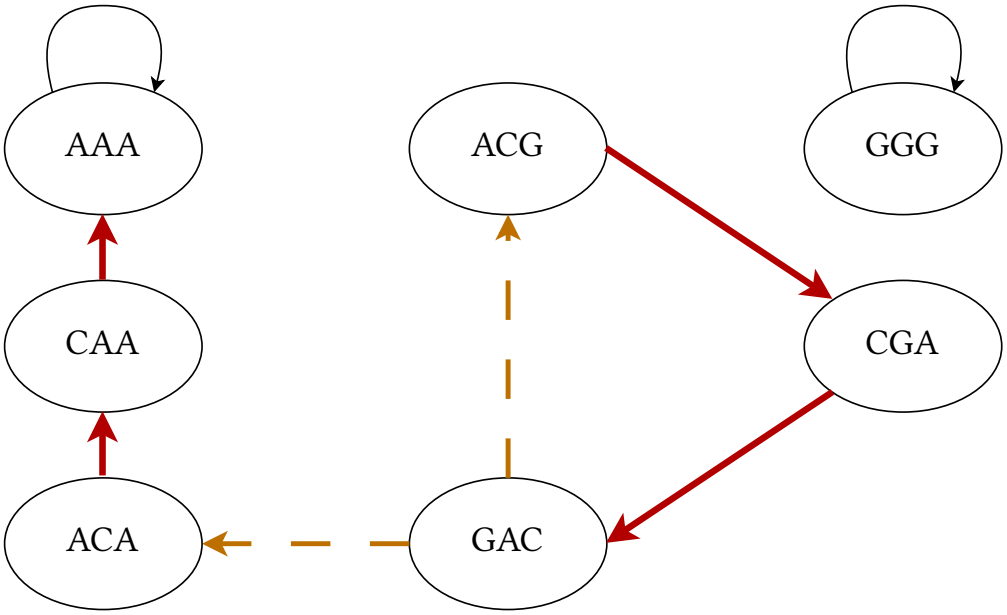

Figure 5: Simplitig collapsed edges from Figure 2 are highlighted. Only one of the two orange edges can be collapsed.

Note that the two representations have different sizes, so the greedy strategy does not guarantee an optimal representation.

Eulertigs

Eulertigs [6] are a SPSS representation derived from an improvement of the simplitigs representation, that guarantees a minimal number of sequences, and therefore minimum cumulative length, among all simplitig representations without repeated k -mers. They are built by finding an Eulerian walk in a modified "eulerized" version of the de Bruijn graph, where fake edges are added between nodes having an unbalanced number of incoming and outgoing edges to make every connected component of the graph Eulerian. Then the Eulerian walks for each component are split at the fake edges to obtain the final set of strings. Figure 6 shows the eulerization of the graph from Figure 2. In this example, one edge is added between the nodes that have unbalanced incoming and outgoing degrees.

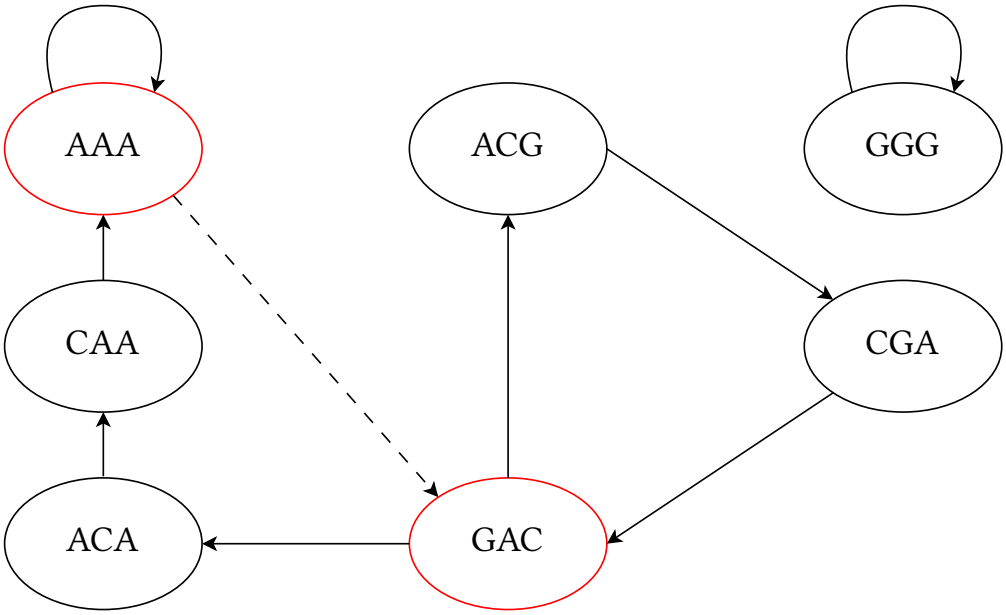

Figure 6: Extra dotted edges are added to the graph in Figure 2 to make all components Eulerian. The nodes with an imbalance in incoming and outgoing edges are outlined in red.

The Eulerian walk in this modified graph leads to the following eulertig representation: $\{GGG, ACGACAAA\}$, which is the same as one of the two possible simplitig representations, but it is guaranteed to be minimal in size among all of them.

1.3.9 Node-centric vs edge-centric de Bruijn graphs

De Bruijn graphs can be defined in two different ways, called *node-centric* and *edge-centric*. In the *node-centric* definition, a de Bruijn graph of order k has nodes containing k-mers and edges of size $k+1$ representing the overlap between two k -mers, while in the *edge-centric* definition the edges are of size k and the nodes are of size $k-1$. In an edge-centric graph, nodes are not represented explicitly, but only if they are connected by at least one edge. In GGCAT we use the edge-centric definition, so k becomes the minimum size of a unitig and the $(k-1)$-mer extremities of maximal unitigs represent the branching nodes of the graph.

Equivalence of representations

We now prove the equivalence between the (maximal) unitigs of the node-centric de Bruijn graph and the (maximal) unitigs of the edge-centric de Bruijn graph, built on the same set of strings R (i.e., their spellings are exactly the same strings). Note that we give this proof only for directed graphs.

For ease of notation, in this section we will denote the edge-centric graph of R as $G_k^e(R)$. The *node-centric* graph for R, which we denote as $G_k^n(R)$, is formally defined by adding a node for every k-mer of R, and an edge between two nodes x and y if $\mathit{suf}_{k-1}(x) = \mathit{pre}_{k-1}(y)$. In a node-centric graph, a path P (containing at least one node) is said to be a *node-centric unitig* if all nodes of P, except the last node, have out-degree equal to one, and all nodes of P, except the first one, have in-degree equal to one [17]. A node-centric unitig is said to be *maximal* if it cannot be extended by a node on either side [17]. See Figure 7 for an example. We will use the term *unitig* to refer to a unitig in the edge-centric graph and *node-centric unitig* to refer to the unitigs just defined in the node-centric graph.

Figure 7: Top: Illustration of the node-centric de Bruijn graph $G_4^n(R)$, where we assume R is the set consisting of all 4-mers that label the nodes of the shown graph. Bottom: the edge-centric de Bruijn graph $G_4^e(R)$, built on the same set R. In both graphs we draw in red their corresponding maximal unitigs (i.e., node-centric and edge-centric, respectively). In the node-centric graph, the node-centric unitigs in the middle of the figure do not include the node ACGT, since its out-degree is different from one, and do not include the node CTTG since its in-degree is different from one. Every unitig is labeled with its spelling, also in red. Notice that these spellings are the same in both graphs.

The *spelling* of a path $P = (x_1, ..., x_t)$ in $G_k^n(R)$ is analogously defined as the string $x_1 \odot^{k-1} ... \odot^{k-1} x_t$. As in the edge-centric case, by a node-centric unitig we will refer to either a path P in the node-centric graph, or to the spelling of P.

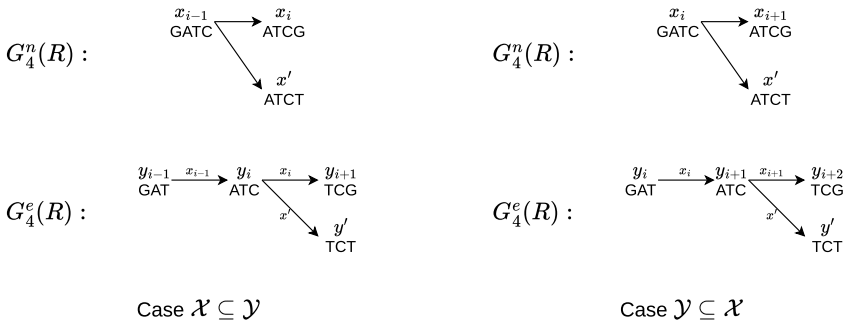

Figure 8: Illustration of the two analogous cases in the proof of Theorem 1. For concreteness, we also draw some example strings on the nodes of graphs. In the edge-centric graphs on the bottom, we additionally label the edges with their corresponding k-mer.

Theorem 1:

Let R be a set of strings, and let $\mathcal{X}$ be the set of all node-centric unitigs of $G_k^n(R)$ and let $\mathcal{Y}$ be the set of all unitigs of $G_k^e(R)$. Then $\mathcal{X} = \mathcal{Y}$.

Proof:

We prove the theorem by proving $\mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq \mathcal{X}$.

$\mathcal{X} \subseteq \mathcal{Y}$: Let $X \in \mathcal{X}$; we show that $X \in \mathcal{Y}$ holds. Let $X = (x_1, ..., x_p)$, $p \geq 1$, and let $y_i := \mathtt{pre}_{k-1}(x_i)$, for each $i \in \{1, ..., p\}$, and let $y_{p+1} := \mathtt{suf}_{k-1}(x_p)$. Since $x_1, ..., x_p$ are k-mers of R, then the edges (y_i, y_{i+1}) exist in $G_k^e(R)$ for all $i \in \{1, ..., p\}$. We claim that $Y := (y_1, ..., y_{p+1})$ is a unitig in $G_k^e(R)$ (clearly, by construction, Y has the same spelling as X). First, note that Y contains at least one edge, since $p \geq 1$.

Suppose for a contradiction that Y is not a unitig in $G_k^e(R)$, namely that there is some internal node y_i, for some $i \in \{2, ..., p\}$, having in-degree or out-degree different from one; suppose w.l.o.g., that it has out-degree different from one. Since the edge (y_i, y_{i+1}) exists in $G_k^e(R)$, this means that the out-degree of y_i is non-zero, and thus at least two. Let (y_i, y') be another outgoing edge from

y_i ($y' \neq y_{i+1}$). Refer to Figure 8(left) for an illustration of this configuration. Therefore, $x' := y_i \odot^{k-2} y'$ is a node in $G_k^n(R)$. Moreover, since $i \geq 2$, there is a node y_{i-1} preceding y_i in the unitig Y, and thus a node x_{i-1} preceding x_i in the unitig X. Consider now the nodes x_{i-1}, x_i, x' in $G_k^n(R)$. By definition, we have that $\mathit{suf}_{k-1}(x_{i-1}) = y_i = \mathit{pre}_{k-1}(x_i)$, and $\mathit{suf}_{k-1}(x_{i-1}) = y_i = \mathit{pre}_{k-1}(x')$. Thus, the out-degree of x_{i-1} in $G_k^n(R)$ is at least two. Since x_{i-1} is not the last node of the unitig X (since $i \leq p$), this contradicts the initial assumption that X was a node-centric unitig.

$\mathcal{Y} \subseteq \mathcal{X}$: Let $Y \in \mathcal{Y}$; we show that $Y \in \mathcal{X}$ holds. Let $Y = (y_1, ..., y_p)$, $p \geq 2$ (since edge-centric unitigs are defined to contain at least one edge), and let $x_i := y_i \odot^{k-2} y_{i+1}$, for each $i \in \{1, ..., p-1\}$. Since $x_1, ..., x_{p-1}$ are k-mers of R, then they are all nodes in $G_k^n(R)$, and $X := (x_1, ..., x_{p-1})$ (clearly, by construction X has the same spelling as Y) is a path in $G_k^n(R)$ (note that $p - 1 \geq 1$, since $p \geq 2$). We claim that X is a node-centric unitig in $G_k^n(R)$.

Suppose for a contradiction that this is not the case. W.l.o.g., we can assume that there is x_i, for some $i \in \{1, ..., p-2\}$, having out-degree different from one, and thus at least two. Let $x' \neq x_{i+1}$ be another out-neighbor of x_i, and let $y' := \mathit{suf}_{k-1}(x')$. Refer to Figure 8(right) for an illustration of this configuration. Consider now the nodes y_{i+1}, y_{i+2}, y' (recall $i \leq p-2$) in $G_k^e(R)$. By definition, (y_{i+1}, y_{i+2}) is an edge in $G_k^e(R)$; moreover (y_{i+1}, y') is also an edge in $G_k^e(R)$, because $x' = y_{i+1} \odot^{k-2} y'$ is a node in $G_k^n(R)$. Thus, the out-degree of the node y_{i+1} is at least two in $G_k^e(R)$. Since $i \leq p-2$, this means that y_{i+1} is a node of the unitig Y, different from its last node y_p, having out-degree at least two, which contradicts the fact that Y is a unitig in $G_k^e(R)$. ■

Next, we prove the same equivalence, but for *maximal* unitigs.

Corollary:

Let R be a set of strings, and let $\mathcal{X}$ be the set of all maximal node-centric unitigs of $G_k^n(R)$ and let $\mathcal{Y}$ be the set of all maximal unitigs of $G_k^e(R)$. Then $\mathcal{X} = \mathcal{Y}$.

Proof:

We prove one inclusion only, since the other one follows completely symmetrically. Let M be a maximal node-centric unitig in $G_k^n(R)$. We want to prove that M is also a maximal unitig in $G_k^e(R)$. Suppose for a contradiction that M is not maximal in $G_k^e(R)$. Then there exists a unitig M' of $G_k^e(R)$, $M \in M'$ with $|M'| > |M|$. By Theorem 1, we have that M' is also a node-centric unitig in $G_k^n(R)$. Since $M \in M'$, this contradicts the maximality of M in $G_k^n(R)$. Thus, M is a maximal unitig of $G_k^e(R)$. ■

1.3.10 Minimizers

Minimizers are a common technique that allows k-mers to be partitioned into different groups, while trying to preserve contiguity between k-mers that fall into the same group. The core idea is to assign to each k-mer a shorter representative substring (called the minimizer) which is the smallest substring (either lexicographically or using a hash function) of length m (with $m < k$) within that k-mer. By doing that, sequences that share overlapping k-mers are likely to share the same minimizer, clustering related k-mers together. Since contiguous k-mers overlap by $k - 1$ characters, extending the current k-mer window by one position only requires adding a new character and removing the first one. This property enables efficient rolling computation of minimizers across the sequence using sliding window algorithms. An example of minimizers is shown in Figure 9.

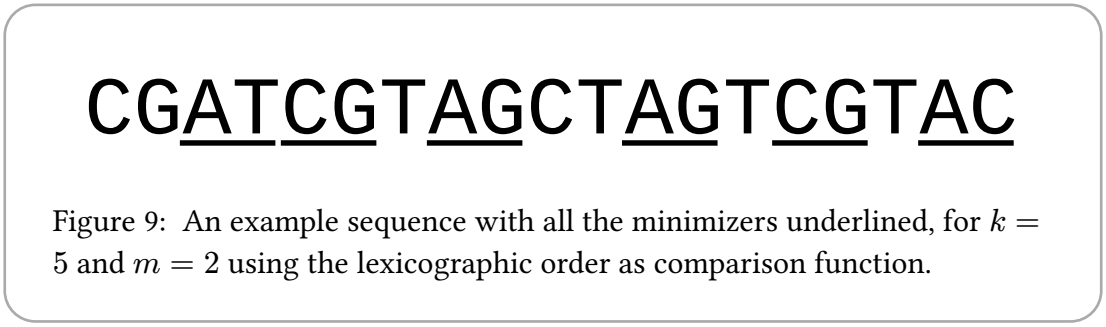

Figure 9: An example sequence with all the minimizers underlined, for $k = 5$ and $m = 2$ using the lexicographic order as comparison function.

GGCAT makes extensive use of minimizers to optimize memory consumption and computational efficiency. By organizing k-mers according to their minimizers, GGCAT is able to partition the graph construction problem into smaller subproblems that can be solved independently using less main memory. As a result, the use of minimizers allows GGCAT to scale effectively to large genomic datasets, maintaining performance and memory efficiency even with a huge number of input sequences.

1.3.11 Superkmers

Superkmers are a way of partitioning k-mers by their minimizer, while preserving a certain amount of contiguity between adjacent k-mers and thus reducing the memory required. A superkmer is defined as a maximal substring of a sequence such that all the k-mers contained within that substring share the same minimizer. An example of superkmers is shown in Figure 10.

Figure 10:

Superkmers corresponding to each minimizer from the sequence in Figure 9, for $k = 5$ and $m = 2$. The superkmers are:

- CGATCGT with minimizer AT
- TCGTA with minimizer CG
- CGTAGCTA with minimizer AG
- GCTAGTCG with minimizer AG
- GTCGTA with minimizer CG
- CGTAC with minimizer AC

Superkmers are already widely used in k-mer based tools, such as BCALM2 [17] and KMC3 [30].

1.3.12 Buckets

In this thesis, the term *buckets* refers to files that are used to store temporary data on disk so that it can be processed in smaller, independent groups. Algorithms using them are usually divided into two phases. The first is a preprocessing phase in which the input data is partitioned into these files according to a specific criterion. In the second phase this partitioning allows the algorithm to process each subset of data individually, without needing to keep the entire dataset in memory at the same time.

Although bucketing is not a fundamental requirement for most algorithms, it is a useful technique that can significantly reduce the working set of data that must be loaded and processed simultaneously, and in many cases it leads to more efficient execution, particularly when working with large genomic datasets or other high-throughput data sources that exceed available RAM.

The key point for an effective bucketing strategy is how the data is partitioned. It should ensure that solving each subproblem, defined by the data contained within a single bucket, produces a local solution that is a valid part of the global solution. Achieving this often requires retaining extra information that captures relationships between data elements in different buckets. This metadata is later used in a reconstruction or merging phase, where the partial solutions are integrated to produce the global result.

In practice, bucketing provides a balance between scalability and computational efficiency. By dividing a large problem into smaller, manageable subproblems, it allows algorithms to operate on datasets much larger than the available main memory while still maintaining good performance characteristics. Tools like GGCAT and other k-mer based large-scale genome tools rely heavily on this approach to handle massive collections of k-mers, ensuring that intermediate computations remain tractable even when processing multi-terabyte-scale inputs.

1.3.13 Bit representation of DNA sequences

The 4 bases in DNA sequences can be efficiently represented using a 2-bit encoding scheme, where each base is mapped to a unique 2-bit binary code. The typical mapping is derived from bitwise operations on the ASCII values of the characters, as it allows very fast conversions between characters and their 2-bit representation. The ASCII values and 2-bit representations for the bases are:

Base	ASCII Value (Decimal)	ASCII Value (Binary)	2-bit Representation
A	65	01000**00**1	00
C	67	01000**01**1	01
G	71	01000**11**1	11
T	84	01010**10**0	10

The 2-bit representation can be obtained by taking the second and third least significant bits of the ASCII binary representation, as they uniquely identify each base. This representation is 4 times more space efficient than using a full byte per base, so it is very important to use it when processing large genomic datasets.

1.3.14 Hashing

Hashing is a very powerful technique widely used in computer science to map arbitrary data to fixed-size integers. It finds applications in many fields, such as data structures (e.g., hash tables), cryptography, and data deduplication. A hash function f is defined as a function that takes an input x in a certain domain and produces a fixed-size integer. Many different hash functions exist, with different properties, but in general a good hash function should have the following properties:

- Determinism: for the same input x, the output $f(x)$ is always the same.
- Uniformity: the output values should be uniformly distributed across the output range.
- Randomness: small changes in the input should produce large and unpredictable changes in the output.
- Collision resistance: for a large output range, it should be hard to find two different inputs x and y such that $f(x) = f(y)$.
- Efficiency: the hash function should be computationally efficient to compute. In case it is needed to compute hashes over windowing sequences (e.g., k-mers in

a DNA sequence), it should be possible to compute the hash of the next window efficiently from the previous one (rolling hash).

Since GGCAT operates on fixed-length windows of input sequences (i.e., k-mers), it uses mainly rolling hash functions. GGCAT uses two kinds of hash functions, as it requires different properties for different parts of the algorithm. The first kind is used to compute the minimizer values of the k-mers, and it needs to be very fast to compute and uniform. It does not require collision resistance, as a collision in minimizers does not affect the correctness of the algorithm. For this purpose, GGCAT uses the NtHash function [31]. The second kind of hash function is used to identify k-mers uniquely, so it is paramount that it has a low (or even zero) collision rate. In this case it is not important that the hash function is uniformly distributed. To achieve this, GGCAT uses two kinds of functions: for small k values (up to 64) it uses a simple identity rolling hash that encodes the 2-bit sequence representation, while for larger k values it uses a custom implementation of the Rabin-Karp rolling hash [32].

NtHash

NtHash [31] is a specialized hash function designed specifically for hashing DNA sequences. It supports rolling hash computations, which makes it particularly efficient for applications that require hashing of overlapping k-mers in genomic sequences. NtHash uses a combination of bitwise operations (rotations and xors) to update the hash value. The pseudocode for computing the hash of a k-mer using NtHash is shown below:

```
NtHash(kmer, k):
1  h = 0
2  for i ← 0 to k − 1:
3      h = h xor rotl(bhash(kmer[i]), k − i − 2)
4  return h
```

where `bhash` is a function mapping each nucleotide to a different random integer value, and `rotl` is a function that performs a left bitwise rotation on the input integer by the specified number of bits.

To update the hash value when moving to the next k-mer, NtHash uses the following algorithm:

```
NtHash_update(h, out_char, inc_char, k):
1  h = rotl(h, 1) xor bhash(inc_char)        // Add the new character
2  h = h xor rotl(bhash(out_char), k - 1)    // Remove the old character
3  return h
```

Contrary to the original definition, in GGCAT the `bhash` function is implemented using a multiplication with a large prime number, as it has been experimentally found to be faster than using a lookup table.

Identity rolling hash

The identity rolling hash is used in GGCAT for small k values, with the smallest integer size that can hold all the bits of the 2-bit representation (as shown in Section 1.3.13) of a k-mer. The unused high order bits of the integer are set to zero. This hash has the extra property of being bijective, so the k-mer can be reconstructed exactly from its hash value and there cannot be collisions.

Rabin-Karp rolling hash

Rabin-Karp rolling hash [32] is a widely used rolling hash function that is particularly effective for string matching applications. It is useful for computing hashes of a substring with the ability to efficiently add or remove characters from either end of the substring. The Rabin-Karp hash function is computed with a polynomial having coefficients derived from the characters of the string, while the exponents are based on the position of the characters:

$$h(s) = \sum_{i=0}^{k-1} s[i] * b^{k-i-1} \bmod q$$

By choosing coprime b and q, the Rabin-Karp hash function reduces periodicity and collisions. In practice b is chosen as a large prime number, while q is usually a power of 2 corresponding to the width of the integer type used. This choice makes modulo operations implicit: the polynomial can be evaluated using the natural wrapping behavior of unsigned integer arithmetic, which is equivalent to reducing modulo q after each operation.

To remove a character it is sufficient to subtract its contribution (multiplied by the appropriate power of b depending on its position) from the hash value. To add a new character, the current hash value should have all its characters shifted to the left by increasing all the powers of b by one (which is equivalent to multiplying the hash value by b), and then adding the contribution of the new character to the hash:

$$r(h, o, i, k) = \left(\left(h - o * b^{k-1}\right) * b + i\right) \bmod q$$

Here h is the previous hash value, o is the character being removed, i is the character being added, and k is the length of the substring.

In GGCAT, a slightly modified version of the Rabin-Karp rolling hash was developed, that instead of using the exact integer value of the character, maps it to a random integer using a fixed hash function. This modification improves the unifor-

mity of the hash values for small k values, which, combined with a large b, improves the collision resistance of the hash for short sequences.

Canonical hashing

As introduced in Section 1.3.2, treating reverse-complement k-mers as equivalent is the most common approach when dealing with sequencing data whose strand origin is unknown. To make sure that a k-mer and its reverse complement map to the same hash value, GGCAT uses canonical hashing. A canonical hash is built by computing two hash functions, one for the k-mer and one for its reverse complement, and then taking the minimum of the two hash values. Taking the minimum ensures that no matter which orientation the k-mer is in, it will always have the same canonical hash.

1.4 Compared tools

This section describes the tools most relevant to GGCAT. Since GGCAT's design is most directly inspired by BCALM2, a detailed description of BCALM1/2 is provided first, followed by shorter descriptions of other related tools.

1.4.1 BCALM1

BCALM1 [33] (Chikhi & Rizk, 2013) was one of the first tools to construct compacted de Bruijn graphs in low memory. It uses a Bloom filter to approximately represent the set of k-mers present in the input reads. In a first pass, all k-mers are inserted into the Bloom filter. A second pass then identifies which k-mers are genuine (present in the Bloom filter) and begins constructing compacted unitigs. Because Bloom filters have a small false-positive rate, a verification step is needed to remove spurious extensions, requiring an additional data structure. The main advantage of BCALM1 is its very low memory usage due to the Bloom filter, but the multiple passes and the overhead from false-positive removal are the main bottlenecks of this strategy.

1.4.2 BCALM2

BCALM2 [17] (Chikhi, Limasset & Medvedev, 2016) is the main inspiration of GGCAT. It significantly improved over BCALM1 by replacing the Bloom filter with a minimizer-based bucketing strategy, eliminating the need for multiple passes and false-positive handling.

The algorithm operates in three phases:

- *k-mer partitioning*: each k-mer x is put in either one or two buckets according to its left and right minimizers:
  - $lmm(x)$, the minimizer of the $k-1$ prefix of x
  - $rmm(x)$, the minimizer of the $k-1$ suffix of x

Then two bucket indices b_l, b_r are derived from the $lmm(x)$, $rmm(x)$ values. If $lmm(x) = rmm(x)$, x is written only in one bucket; otherwise, it is written in both b_l and b_r.

- *Bucket compaction*: k-mers are compacted in memory to produce local unitigs with a standard approach. If a local unitig extremity k-mer x has $lmm(x) \neq rmm(x)$, it is marked as lonely and the whole unitig is put in a special reunite file; otherwise, if the local unitig has no lonely extremities, it is considered complete and written to output.
- *Lonely unitigs joining*: The reunite file is read and the local unitigs are joined on their matching k-mers using a standard single-threaded union-find algorithm.

1.4.3 KMC3

KMC3 [30] is a widely used tool for k-mer counting, and it is used as a preprocessing step by some de Bruijn graph assemblers, such as Cuttlefish 2, to deduplicate k-mers and apply the abundance cutoff. The main idea behind KMC3 is to partition the input reads into superkmers (see Section 1.3.11), and to store those superkmers into on-disk buckets according to signatures derived from their minimizers. Then each bucket is processed independently and in parallel to deduplicate and count the k-mers, by employing an optimized sorting algorithm specialized for fixed-length short sequences, which is well suited to k-mer sorting. While highly optimized, it has a very high intermediate disk usage, proportional to the uncompressed input size, making it sub-optimal for very repetitive datasets. Note that if a k-mer appears multiple times in the input data, it is guaranteed that all the occurrences are stored in the same bucket, thus ensuring correct k-mer counts.

1.4.4 Cuttlefish 1

Cuttlefish 1 [34] (Khan et al., 2021) introduced a fundamentally different approach for large-scale de Bruijn graph construction from genome collections. Rather than using hash tables to track k-mer extensions, it encodes the local state of each graph vertex as a compact automaton: each vertex tracks whether it has zero, one, or more than one distinct left and right neighbors in the k-mer set. By processing sorted k-mer lists and updating the automaton state, unitigs can be identified without storing all k-mers in memory simultaneously. Cuttlefish 1 uses KMC3 for k-mer counting and a minimal perfect hash function (MPHF) over k-mers to index the automaton states. Its main advantage is low and predictable memory usage; its main limitation is that the MPHF construction cost grows with k.

1.4.5 TwoPaCo

TwoPaCo [35] (Minkin & Medvedev, 2017) is designed specifically for constructing compacted de Bruijn graphs from collections of complete (assembled) genomes rather than reads. It uses a two-pass approach: in the first pass, a Bloom filter identifies

candidate junction k-mers (those with more than one distinct predecessor or successor); in the second pass exact junction k-mers are found using a hash map (on the reduced set of candidates) and the graph is built by tracing paths between confirmed junctions. This two-stage strategy reduces the required amount of memory since the Bloom filtering step keeps only a small fraction of original k-mers to be processed in the second pass. Because it assumes whole-genome inputs, it can exploit the absence of sequencing errors and does not need abundance thresholds, making it fast and memory-efficient for genome databases. It is not designed for read datasets, where errors and high redundancy make the Bloom filter approach less accurate.

1.4.6 Cuttlefish 2

Cuttlefish 2 [5] is a widely used tool for maximal unitig computation. It starts with an initial k-mer counting step using KMC3 [30]. It employs a minimal perfect hash function on the k-mers using BBHash [36]. The key insight relies on an automaton-based approach to compute the branching state of every graph vertex, using only the minimum amount of information, i.e., zero, one, or more than one left / right neighbors. Then, it builds the graph by looking at the automaton states, extending a unitig while the current vertex does not branch forwards and the following vertex does not branch backwards. Like Cuttlefish 1, it uses KMC3 as a preprocessing step to do k-mer counting, so it still has a very high disk usage for repetitive datasets.

1.4.7 Cuttlefish 3

Cuttlefish 3 [37] is a newer revision of the tool, which improves performance while keeping comparable memory and disk usage. It adopts a read-partitioning strategy very similar to the GGCAT one, with a superkmer-based approach. To construct the local unitigs in each partition, it uses a hash map storing all the vertices along with their state (branching or not). This allows for a very fast extension of non-branching paths, since only one query has to be performed on the hash map in the non-branching case. To join the unitigs built in the previous phase, it models the task as a list-ranking problem, allowing deterministic running time with a high level of parallelism. It also features many optimizations and an improved way of managing colors. Although the algorithm represents a substantial improvement over its predecessor, Cuttlefish 2, and the experiments confirm its strong performance, in our benchmarks it crashed or deadlocked on several larger runs; we report these as observed failures without attributing a definitive cause.

1.4.8 BiFrost

When the first version of GGCAT was being developed, the state-of-the-art tool for maximal unitig computation with associated color information was BiFrost [38]. It uses an in-memory only approach, with various blocked Bloom filters [39, 40] partially indexed by minimizers that approximate the k-mers present in the final graph, and does several passes on the original input to remove the false edges wrongly

created due to the use of Bloom filters. Then it internally stores the k-mers grouped by minimizers, allowing for relatively fast deletions and insertions of new k-mers. However, while blocked Bloom filters are very memory efficient, they are not very cache efficient when compared with sequential algorithms, even with the infra-block SSE2 optimizations done in BiFrost. Also the memory representation of the k-mers gives a trade-off between the ease of doing small updates to the graph and the speed of inserting batches of k-mers, thus the build time of the graph is still considerably high. The main advantage of BiFrost over other tools is the ability to perform real-time queries and modifications on the graph, since it is all indexed and loaded in memory. For that reason, it is still widely used in contexts where online queries and updates are needed. To (optionally) build a colored graph, it uses various types of compressed bitmaps (roaring bitmaps [41], or simple bitsets) to store the set of colors of each k-mer. While this preserves the fast updates and queries, it stores redundant colorset information since k-mers that share the same set of colors are still encoded as two separate sets.

1.5 Code availability

All the code of GGCAT is open source, licensed with the MIT license and freely available at: https://github.com/algbio/ggcat/.

The tool used for benchmarks as well as the datasets are available at: https://github.com/Guilucand/ggcat-test-benchmarks/.

2 GGCAT 1

2.1 Overview

GGCAT is a tool that produces compacted (and optionally colored) de Bruijn graphs from reads or genomes in .fasta or .fastq file formats. The aim of this tool is to handle very large quantities of input data while retaining relatively low memory usage and computation time. To achieve that, GGCAT uses different algorithms that split the de Bruijn construction problem into many different subproblems that can be solved independently, thus using far less working memory than processing the whole dataset at once. It uses a mix of known and novel techniques that dramatically improve performance and make the tool scalable to datasets that competing tools did not process in our experiments, especially for colored graphs, as shown in Section 2.6.

2.2 Algorithm phases

2.2.1 Superkmers partitioning

Each read R_j is split into substrings $S_1, ..., S_{\ell_j}$ that overlap on $k-2$ characters, such that all $(k-1)$-mers of S_i have the same minimizer, for all $i \in \{1, ..., \ell_j\}$. For the minimizer hash function $mini$ we use the ntHash [31] function, because it can give fast computation while ensuring good randomness in its value. Note that we can have multiple minimizer locations in the same substring S_i as long as they have the same hash value. We can compute $S_1, ..., S_{\ell_j}$ in linear time in the size of R_j as follows. First, for every m-mer x of R_j, we compute $mini(x)$ in a rolling manner. Then, in a sliding window manner, we compute the minimum of each window of $k-m$ consecutive m-mers (which correspond to a $(k-1)$-mer). Finally, we group consecutive $(k-1)$ -mers that share the same minimum in their corresponding window. For efficiency, we perform these three steps in a single pass over R_j.

For every S_i obtained in this manner, let a and b be the characters of R_j immediately preceding and succeeding S_i in R_j, respectively, or \$ if they do not exist. We call a and b *linking characters*. Consider the string $S'_i := a \cdot S_i \cdot b$ and observe that S'_{i-1} and S'_i have a suffix-prefix overlap of k characters, since S_{i-1} and S_i have a suffix-prefix overlap of $k-2$ characters and we added b at the end of S_{i-1} and a at the beginning of S_i. See Figure 11 for an illustration. Recall that all $(k-1)$-mers of S_i have the same minimizer, say h; we assign each extended string S'_i to a *group* G_h associated to such unique minimizer h. We say that a k-mer x appears in a group G_h if x is a substring of some S'_i in G_h.

The above grouping strategy is similar to the one in [30], applied to k-mers instead of $(k-1)$-mers (our strings $S_1, ..., S_{\ell_j}$ are called *super* $(k-1)$*-mers* in [18]), with the exception that when we group we add the linking characters. The following simple properties are key to ensuring the correctness of our approach.

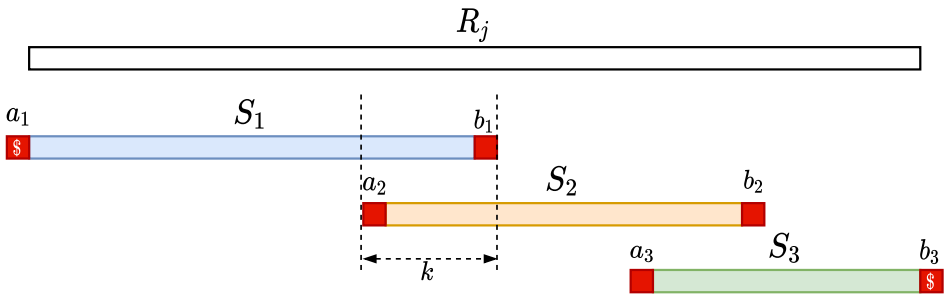

Figure 11: Illustration of the read splitting step. Read R_j is split into substrings S_1, S_2, S_3 such that all $(k-1)$-mers of each S_i have the same minimizer, and extra linking characters (in red) are added to each S_i. The overlap between two such consecutive extended S_i's is of exactly k characters.

Lemma 2.1:

Let x be a k-mer appearing in the reads R, and in a group G_h. The following properties hold:

(a) There is at most one other group in which x appears, and moreover, x appears in two distinct groups if and only if $mini(pre_{k-1}(x)) \neq mini(suf_{k-1}(x))$

(b) $occ(x, G_h) = occ(x, R)$;

(c) If $mini(x) = h$ (i.e., x does not contain a linking character), and it has a $(k-1)$-suffix-prefix overlap with some k-mer y (in either order), then also y appears in group G_h.

Proof:

(a) Let $h_1 := mini(pre_{k-1}(x))$ and $h_2 := mini(suf_{k-1}(x))$. The k-mer x appears only in groups G_{h_1} and G_{h_2}, which are two distinct groups if $h_1 \neq h_2$.
(b) This follows by construction of the group, since all the k-mer occurrences that have the same minimizer are put in the same group.
(c) If the minimizer of y is also h (i.e., it does not contain a linking character), then y also appears in G_h. If not, recall that we added an extra character at the

beginning and end of every string assigned to G_h, thus y is a k-mer containing a linking character and therefore appears in G_h. ■

Since we want to write the groups to disk, and their number is the number of distinct minimizers, we merge the groups into a smaller number of buckets that are written to disk. The assignment of each group to a bucket is determined by truncating its minimizer hash value to a fixed number of bits (i.e., by taking the hash modulo the number of buckets), so the bucket of each group is fully determined by its minimizer hash.

2.2.2 Partial unitigs construction

Lemma 2.1 ensures that extending any k-mer x can be correctly performed just by querying the group of x.

For each group, we perform:

- A k-mer counting step of the strings in the group, using a hash map, while also keeping track of whether a k-mer contains a linking character. More precisely, we scan each string in a group, and for each k-mer that we encounter we increase by one its abundance in the hash map, and add a single extra bit flag to the hash map entry if it contains a linking character. To avoid increasing the number of bytes needed for the hash map values, the flags are stored in the most significant bits of the abundance counter.
- We then define a list K containing all the unique k-mers of the group that satisfy the required abundance threshold. This list can be computed in two different ways:
  - If hashes are invertible, the list K can be implicitly iterated by traversing the hash map and inverting each hash value
  - Otherwise (for $k > 64$) we create a new list during the k-mer counting step, which explicitly stores all the unique k-mers and is filtered by abundance when traversed.
- We traverse the list K of k-mers, and for each unused k-mer x, we initialize a string $z := x$, which will be extended right and left as long as it is a unitig (see Figure 12, Figure 13). We try to extend z to the right by querying the hash map for $\mathsf{suf}_{k-1}(z) \cdot c$, for all $c \in \{A, C, G, T\}$. If there is a unique extension y such that $\mathsf{suf}_{k-1}(z) = \mathsf{pre}_{k-1}(y)$, then we query the hash map for $c \cdot \mathsf{pre}_{k-1}(y)$, for all $c \in \{A, C, G, T\}$. If exactly one match is found (i.e., $\mathsf{suf}_k(z)$), then we replace z with $z \odot^{k-1} y$, and we mark k-mer y as used in the hash map. If y is not marked in the hash map as having a linking character, then we repeat this right extension with the new string z. The queries to the hash map are correct thanks to Lemma 2.1 (c). When we stop the right extension, we perform a symmetric left-extension of z. After both extensions are completed, the resulting unitig z is

given a unique index id_z. If the extension of z was stopped because of a linking character in the first or last k-mer y of z, we add (y, id_z) to a list L.

Notice that, after all groups have been processed, for any (y, id_z) in L, there exists exactly one other $(y, id_{z'})$ in L, added from a different group, by Lemma 2.1 (a). These two tuples indicate (non-maximal) unitigs that have to be iteratively merged to obtain the maximal unitigs.

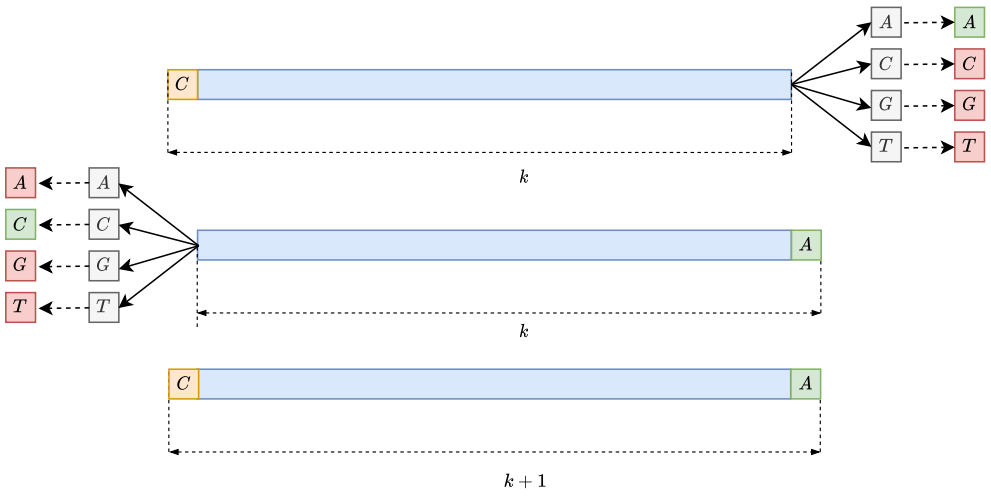

Figure 12: The extension step of the intermediate unitig construction happens inside each group. For each k-mer (top) it looks for a possible extension by checking all the 4 possible neighbor k-mers in both directions, and extends the k-mer (bottom) only if there is exactly one match both forwards and backwards (depicted in green in the first two figures from the top). Then it repeats the same process until no more extensions can be performed.

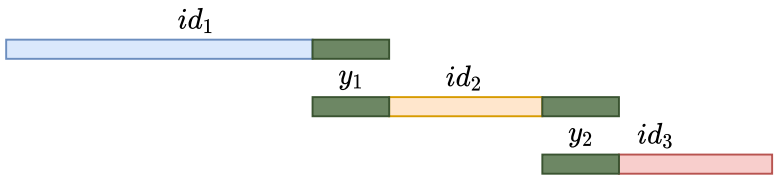

Figure 13: The result of the intermediate unitig construction. Each intermediate unitig that has a possible extension shares an ending with another intermediate unitig.

2.2.3 Unitigs merging

The tuples (y, id_z) in L are sorted by y, such that the two entries (y, id_z) and $(y, id_{z'})$ appear consecutively. Moreover, for any unitig z, there are at most two entries (x, id_z) and (y, id_z) in L (corresponding to its two endpoints). From these, we construct a list $(id_z, id_{z'})$ that is put in another list P of pairs of unitigs that must be merged into maximal unitigs. This is one of the hardest steps to parallelize, since no partitioning can be done ahead of time that puts all the unitigs that are contained in a maximal unitig in the same partition. In other tools, e.g., BCALM2 [17], this step is done using a union-find data structure that can be difficult to be used with concurrency. Our solution uses a randomized approach (i.e., with guaranteed correctness and only *expected* running time) to put in the same partition the unitigs that should be merged, repeating the process until all the unitigs are merged into the final maximal unitigs.

We proceed as follows (see Figure 14). We allocate a fixed number of buckets. Initially, for each list in P, we mark both its ends as *unsealed*. We repeat the following procedure until P is empty:

- For each list in P, we choose at random one of its unsealed ends. Without loss of generality, let this end be l. We put the list in the bucket corresponding to l, while in the bucket corresponding to the other ending r we put a placeholder.
- Inside each bucket, we sort the lists by the ending that caused the list to be placed in the bucket. Then, we merge all the endings that are equal to produce longer lists. If an ending in a bucket is not merged, and it has no corresponding placeholder (of another list) in the bucket, then it is marked as *sealed*.
- Finally, we remove from P each list having both ends sealed.

In the above process, given two lists in P that must be merged, there is at least a probability of $\frac{1}{4}$ that they are assigned to the same bucket to be merged (in the worst case, both ends are unsealed). Thus, in expectation, this desired outcome happens only after 4 tries.

Both L and P are also stored in buckets, to allow more concurrency while processing them.

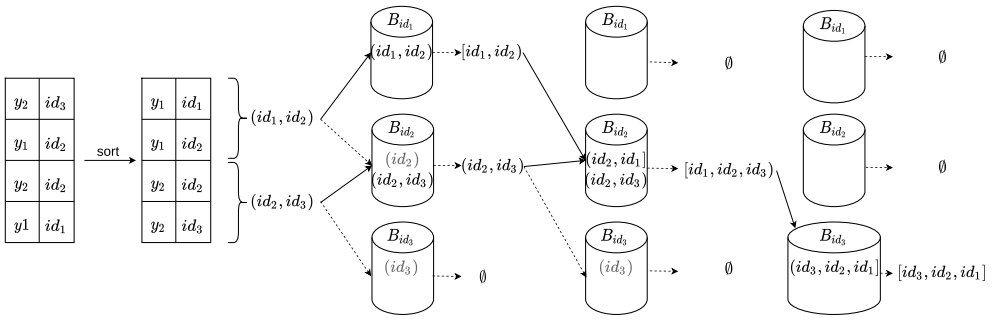

Figure 14: The unitig merging step on the unitigs from Figure 13. Each pair (ending, unitig index) is sorted by ending, and indexes of unitigs that share the same ending are joined in a tuple. Each such tuple is assigned to a bucket corresponding to one of its unsealed endpoint indices chosen at random (solid arrows in the figure). For the other endpoint index of the tuple (dashed arrow), we put a *placeholder* in its corresponding bucket (in gray). Then, inside each bucket, pairs sharing the same unitig index are joined to form larger tuples. If an ending cannot be joined and does not have a corresponding placeholder, then it is marked as sealed and is not selected anymore for bucket assignment. For example, in the first step, in B_{id_1} the pair (id_1, id_2) is sealed at id_1, because there is no placeholder for id_1; however, in B_{id_2} the pair (id_2, id_3) is not sealed at id_2 because id_2 has a placeholder. The steps are repeated until no more tuples can be joined. For the non-canonical case, we can merge the pairs if one extremity is at the end and the other one at the beginning of the pairs. For canonical k-mers, we also have to keep track of the direction of the k-mer before joining them, see Section 2.3 for more details.

2.3 Construction correctness

We start by proving the correctness of the algorithm (without reverse complements).

Theorem 2:

Given a multiset R of strings and an abundance threshold a, the strings U obtained at the end of our algorithm are the maximal unitigs of R under threshold a.

Proof:

We prove that U satisfies the conditions of Definition 1.1 (for non-canonical unitigs).

For condition 0 of Definition 1.1, since we start from k-mers, all strings in U have length at least k. To see that U is also a *set* (i.e., contains no duplicates), in our algorithm every k-mer is considered only once, and is assigned to a unique unitig.

For condition 1 of Definition 1.1, observe that the algorithm does not introduce any k-mer that is not also in R, and it keeps exactly the k-mers whose abundance reaches the threshold. Lemma 2.1 (b) guarantees that $\mathtt{occ}(q, R)$, the number of occurrences of k-mer q, is the same as the number of occurrences in its group, and thus the operation from Section 2.2.2 performed inside its group respects its abundance in R. Therefore, $\forall q \in \Sigma^k \ \mathtt{occ}(q, R) \geq a$ iff $\mathtt{occ}(q, U) \geq 1$.

Next, we prove conditions 3 and 2 of Definition 1.1. Given an intermediate unitig z, we check for the eight k-mers that contain $\mathtt{suf}_{k-1}(z)$; we extend z iff only two k-mers appear (one outgoing from, and one incoming to $\mathtt{suf}_{k-1}(z)$), thus condition 3 of Definition 1.1 is satisfied.

After merging z with this outgoing k-mer, there are no other k-mers (and thus no other unitigs in U, since each k-mer appears exactly once in U, because in Section 2.2.2 we mark the used k-mers) that contain $\mathtt{suf}_{k-1}(z)$ so condition 2 of Definition 1.1 holds for single groups. This condition is still not satisfied globally, due to the repetition of all the k-mers containing a linking character.

We now prove condition 2 of Definition 1.1 after the unitig merging step. Note that the unitigs fed to the unitig merging step are the ones that start or end with a linking character, so they always overlap on k characters. After merging all the repeated k-mers, we satisfy condition 2 of Definition 1.1 globally. ■

All the steps described so far can be easily adapted to work with canonical k-mers, to obtain canonical maximal unitigs (Definition 1.1). This can be done by changing two steps. First, the hash functions are replaced by their *canonical* version, such that the hash of a k-mer is always equal to the one of its reverse complement. Second, in the unitig merging steps, relative orientations of the unitigs are tracked, to allow joining unitigs that can be present in opposite orientations in the input dataset, by reverse-complementing one of them.

2.4 Colors handling

To handle colors, GGCAT uses a *colormap* based approach. A colormap is a data structure that allows efficient storage and retrieval of color subsets (defined as sorted lists of integers), by storing each subset indexed by a single consecutive integer. To check for repeated color subsets, each subset is hashed with a strong hash function,

and the corresponding color index is put in a hash map indexed with the subset strong hash. This allows $O(1)$ checking of duplicate colorsets without keeping the whole colormap in memory and with a negligible risk of collisions (by default 128-bit collision-resistant hashes are used).

The colormap has a custom format where each color subset has its own index and is compressed with multiple algorithms. The format is chosen to be highly compressed and efficient to read in parallel, by allowing almost (parallel) random access to the colormap.

Each colorset is encoded in index order and the colors are sorted, deduplicated, stored as differences and then run-length encoded.

As an example, assume we want to compact the following color multiset S, where each element is a color index (i.e., the index of an input file). Repeated colors can appear when the same input file contains the same k-mer multiple times:

$$S = 8, 1, 3, 9, 4, 5, 8, 2, 6$$

The first step is to sort and deduplicate them, producing the color set S_c:

$$S_c = 1, 2, 3, 4, 5, 6, 8, 9$$

Then the compression algorithm represents the elements following the first one as differences:

$$S_{\text{diffs}} = 1, 1, 1, 1, 1, 1, 2, 1$$

Finally, the differences are optionally run-length encoded (the first element remains fixed):

$$S_{\text{diffs\_rle}} = 1, (1, 5), (2), (1)$$

Every difference is RLE-encoded only if doing that reduces the amount of needed space; otherwise, the difference is directly written. To indicate whether a difference is RLE-encoded, a single bit flag is stored as shown in Section 3.4.2.

After that, the resulting encoding is further compressed using LZ4, to increase the compression ratio in case of similar consecutive colorsets. Only a small range (10MB) of colorset data is LZ4-compressed in each block, to allow accessing a color in the middle of the colormap by only having to decode the block where it is contained.

2.5 Handling of k values > 64

While dealing with large k values, the memory needed to store each k-mer could be too high, as to store explicitly all k-mers of a sequence of length n we need to use $O(nk)$ bases. To mitigate this problem and keep both the memory used and the running time low, when $k > 64$ we use a 128-bit hash to represent each k-mer,

ensuring memory usage linear in the input size and independent of k. This approach introduces a very small probability of collisions, which is further reduced by splitting the k-mers according to their minimizers, thus limiting the number of k-mers that can potentially collide. The collision probability is so small that in practice no collisions were ever detected in the tests performed, even on very large datasets.

2.6 Results

We tested GGCAT on multiple datasets, comparing it with state-of-the-art tools for both colored and uncolored de Bruijn graph construction. The datasets were chosen to cover different types and sizes of inputs, as described in the next section.

2.6.1 Datasets and experimental setup

Multiple datasets were used to benchmark GGCAT, trying to cover different types of inputs and sizes. The datasets used are the following:

- Human reads from the HG002 sample of the AshkenazimTrio by the Genome in a Bottle consortium [42], sequenced using Illumina technology, with paired-end sequences having an average length of 250 bp
- Gut microbiome reads, from project PRJEB33098 [43]. This dataset is quite difficult to process because there is a very large group of distinct k-mers sharing the same minimizer, and it was useful for testing the *resplit* capability of GGCAT: when a bucket grows too large (far beyond the average bucket size), GGCAT repartitions it using a longer minimizer to keep each subproblem manageable (see Section 3.10).
- 100 Human genomes, from the set of 2505 Human genomes generated by [7]

using GRCh37 and the variant files from the 1000 Genomes Project [44]. For convenience we uploaded the Human genomes used for the benchmark to Zenodo.

- Salmonella genomes, downloaded in February 2022 from the Enterobase database [45]. Depending on the benchmark, either a 100K subset or the full 309K collection was used.
- 649K Bacterial genomes, downloaded from [46].

The main statistics of these datasets are summarized in Table 1.

Dataset	k	#k-mers	#unique k-mers	#max. unitigs	Avg len	Max len	N50
Human reads	27	193386M	3143M	271.71M	37.57	4257	28
	63	162119M	3442M	85.85M	102.09	6428	71
Gut microbiome reads	27	71999M	2583M	210.92M	38.24	5633	30
	63	51096M	3100M	154.78M	82.03	3730	68
	119	18580M	1692M	60.74M	145.86	3196	140
Human genomes (100)	27	288091M	2531M	76.52M	59.07	7397	33
	63	288091M	3251M	41.09M	141.13	12638	125
Salmonella genomes (100K)	27	480109M	372M	25.82M	40.41	98331	32
	63	479815M	689M	24.02M	90.67	329835	73
Bacterial genomes (649K)	27	2527666M	36035M	1356.47M	52.57	68201	34
	63	2523967M	46791M	632.47M	135.98	798154	83
	119	2518215M	56390M	448.62M	243.70	2185866	151
	255	2504523M	71553M	318.53M	478.64	3427805	306

Table 1: Dataset statistics with various values of k. The first column considers the whole input dataset, while the rest of the columns refer to the graph built from the dataset with the specified k value. The Avg, Max and N50 columns are computed on the lengths of the maximal unitigs of the graph. N50 is defined as the length L such that 50% of the total nucleotide content across all maximal unitigs is contained in unitigs of length at least L. $k = 27$ was chosen as it is the same smallest value used for tests in [5] while $k = 63$ represents the largest odd value for which GGCAT uses invertible hashes. Two datasets have extended experiments ($k \geq 119$) to demonstrate GGCAT performance with very large k values.

Experiments were run on three servers of increasing power:

- a *small* server with an AMD Ryzen 5 3600 6-core CPU, 64GB RAM and a RAID 0 with two 7200RPM HDDs;
- a *medium* server with an AMD Ryzen Threadripper PRO 3975WX 32-core CPU, 512GB RAM and a RAID 5 7200RPM HDD;
- a *large* server with two AMD EPYC 7H12 64-core CPUs, 2TB RAM, HDD storage, and a SATA SSD.

2.6.2 Uncolored results

The results for uncolored de Bruijn graph construction are shown in Table 2. GGCAT matches or outperforms Cuttlefish 2 in almost all configurations. The speedup is most pronounced for large values of k and repetitive datasets: on 309K Salmonella genomes at $k = 255$, GGCAT is approximately 21x faster than Cuttlefish 2 (3h:44m vs 77h:58m). For Human reads at $k = 27$, the two tools perform similarly, while at $k = 63$ GGCAT is 2x faster. Cuttlefish 2 failed to complete on Bacterial genomes at $k = 119$ and $k = 255$, while GGCAT handled all configurations. In terms of memory,

GGCAT uses slightly more RAM than Cuttlefish 2 on small datasets, but scales much better on large ones: on Bacterial genomes at $k = 63$, GGCAT uses 10 GB vs 54 GB for Cuttlefish 2. The disk usage of GGCAT is also substantially lower in most configurations, especially for repetitive genome datasets.

Dataset	Server	k	Cuttlefish 2	GGCAT
Human reads	*small*	27	1h:15m (3.95GB) [209GB]	1h:16m (4.54GB) [220GB]
		63	2h:07m (4.23GB) [140GB]	1h:03m (7.11GB) [156GB]
Gut microbiome reads	*small*	27	0h:30m (3.35GB) [78GB]	0h:22m (6.09GB) [78GB]
		63	1h:08m (3.86GB) [107GB]	0h:19m (5.42GB) [51GB]
		119	1h:04m (3.13GB) [97GB]	0h:12m (5.33GB) [32GB]
Salmonella genomes (309K)	*small*	27	6h:59m (4.38GB) [1515GB]	3h:38m (3.46GB) [378GB]
		63	12h:02m (3.88GB) [1145GB]	3h:31m (3.96GB) [274GB]
		119	17h:07m (3.95GB) [1088GB]	3h:39m (4.12GB) [279GB]
		255	77h:58m (4.82GB) [1056GB]	3h:44m (4.33GB) [325GB]
Bacterial genomes (649K)	*large*	27	13h:24m (42.60GB) [2604GB]	7h:7m (8.49GB) [1186GB]
		63	25h:29m (53.84GB) [1962GB]	5h:56m (10.06GB) [810GB]
		119	–	6h:14m (9.46GB) [793GB]
		255	–	6h:30m (9.42GB) [849GB]

Table 2: Uncolored construction, wall clock time, memory usage (in parentheses) and maximum disk usage including the size of the output files [in square brackets]. The Bacterial test was performed on the *large* server with 16 threads while the other tests on the *small* server with 12 threads.

2.6.3 Colored results

The colored construction results are shown in Table 3. GGCAT is dramatically faster than BiFrost colored in all configurations. On 100K Salmonella genomes at $k = 27$, GGCAT completes in 1h:31m compared to 50h:29m for BiFrost, a 33x speedup, and also uses 12x less memory. On Bacterial genomes (649K), BiFrost crashed at $k = 27$

and took more than 10 days at $k = 63$, while GGCAT completed in 13h:28m and 11h:33m respectively.

Dataset	Server	k	BiFrost colored	GGCAT colored
Human genomes (100)	*large*	27	6h:39m (30.68GB) [13GB]	1h:17m (8.29GB) [290GB]
		63	5h:16m (38.65GB) [9GB]	1h:08m (8.80GB) [224GB]
Salmonella genomes (100K)	*medium*	27	50h:29m (79.32GB) [54GB]	1h:31m (6.54GB) [201GB]
		63	46h:34m (85.21GB) [61GB]	1h:11m (6.18GB) [128GB]
Bacterial genomes (649K)	*large*	27	*crashed*	13h:28m (21.64GB) [1296GB]
		63	> 10 *days*	11h:33m (21.48GB) [838GB]

Table 3: Colored construction, wall clock time, memory usage (in parentheses) and the total maximum disk space including the size of the output files [in square brackets], using 16 threads. The Human and Bacterial benchmarks were executed on the *large* server and the Salmonella benchmark on the *medium* server

3 GGCAT 2

3.1 Overview

The second version of GGCAT introduces many improvements in the construction algorithm, increasing performance and decreasing the required computational resources. The optimizations involved almost all parts of the pipeline, and the underlying algorithms were redesigned to mitigate the main drawbacks of the previous version.

Starting from improvements in DEFLATE decompression, which is a bottleneck when processing a few large gzip input files, we also introduced a novel superkmer compaction phase that drastically reduces the temporary disk space needed for the first part of the pipeline. A redesigned unitig extension algorithm ensures a lower memory footprint and increased efficiency, by avoiding the use of a large hash table to store all the k-mers and by keeping them compressed as superkmers for as long as possible. The final unitig merging algorithm was also redesigned to avoid storing a disk-resident list of extremity hashes and to join partial unitigs directly. A further subsplitting step bounds the size of its in-memory hash maps and exposes enough independent work for parallel processing. A specialized hash map and a custom allocator for variable-sized vectors allow much faster handling of colorsets, achieving a global speedup of more than 2x for pangenome colored de Bruijn graph construction.

An improved simplitig algorithm was implemented. It no longer relies on the original matchtigs library, but is instead fully integrated in the unitig extension phase, introducing essentially zero overhead. Finally, a novel eulertig algorithm focused on merging circular simplitigs together was designed, lowering the memory and time requirements needed to compute eulertigs. General improvements in the parallel architecture guarantee better control over both memory and thread usage, fixing rare deadlocks and performance issues that could occur in some cases while using the first version of GGCAT.

3.2 DEFLATE optimization

3.2.1 Background: DEFLATE and its wrapper (gzip)

Input genomic files (FASTA/FASTQ) are commonly distributed in gzip format, which is a thin wrapper around DEFLATE. Since a bottleneck was found in decompressing large gzip input files, the focus of this section is on optimizing DEFLATE decompression.

DEFLATE [47] is a compression format that combines an LZ77 sliding window (up to 32KiB of history) with Huffman coding. The core idea of LZ77 is to find parts of input that are repeated and compress them by encoding the next occurrences as pointers to the previous ones, while Huffman coding allows common symbols to be encoded using fewer bits than the uncompressed representation at the cost of encoding less frequent symbols using more bits. A DEFLATE stream is a sequence of blocks, each beginning with a 3-bit header that indicates whether this is the final block and which coding is used. There are three kinds of blocks:

- Uncompressed: used for data that does not compress well
- Fixed Huffman: used for blocks that do not benefit from the overhead of storing custom Huffman trees. The code lengths are defined by the DEFLATE specification, so no tree description has to be stored in the block.
- Dynamic Huffman: used for typical compressible data, as they can achieve a better compression ratio by creating custom Huffman trees based on the symbol frequencies of the current block encoding.

Since uncompressed and fixed Huffman blocks are not common in compressed biological data, we focus on dynamic Huffman blocks. A dynamic Huffman block begins with the definition of the Huffman litlen (literal/length) and distance (offset) trees, encoded using a compact representation that involves another temporary precode Huffman tree.

Each symbol of the litlen tree then encodes either a literal byte (0-255) or a length for a *copyback* operation. A copyback operation copies a sequence of already-decoded output bytes into the current output position: it reads `length` bytes starting `distance` bytes before the current output position (i.e., from within the sliding history window) and appends them to the output. This mechanism is the core of LZ77 compression: repeated patterns in the input are encoded as back-references rather than being stored again. The offset tree symbols encode the distance for the copyback operations. Every length or offset symbol can have an extra number of bits that encode the final part of the length or offset value, as defined in Table 4 and Table 5 below. These tables are a key part of the DEFLATE specification that a correct implementation must handle in full.

Code	Extra bits	Length range
257	0	3
258	0	4
259	0	5
260	0	6
261	0	7
262	0	8
263	0	9
264	0	10
265	1	11-12
266	1	13-14
267	1	15-16
268	1	17-18
269	2	19-22
270	2	23-26
271	2	27-30
272	2	31-34
273	3	35-42
274	3	43-50
275	3	51-58
276	3	59-66
277	4	67-82
278	4	83-98
279	4	99-114
280	4	115-130
281	5	131-162
282	5	163-194
283	5	195-226
284	5	227-257
285	0	258

Table 4: Length symbols table

Code	Extra bits	Distance	Notes
0	0	1	(distance = 1)
1	0	2	(distance = 2)
2	0	3	(distance = 3)
3	0	4	(distance = 4)
4	1	5-6	(1 extra bit)
5	1	7-8	(1 extra bit)
6	2	9-12	(2 extra bits)
7	2	13-16	(2 extra bits)
8	3	17-24	(3 extra bits)
9	3	25-32	(3 extra bits)
10	4	33-48	(4 extra bits)
11	4	49-64	(4 extra bits)
12	5	65-96	(5 extra bits)
13	5	97-128	(5 extra bits)
14	6	129-192	(6 extra bits)
15	6	193-256	(6 extra bits)
16	7	257-384	(7 extra bits)
17	7	385-512	(7 extra bits)
18	8	513-768	(8 extra bits)
19	8	769-1024	(8 extra bits)
20	9	1025-1536	(9 extra bits)
21	9	1537-2048	(9 extra bits)
22	10	2049-3072	(10 extra bits)
23	10	3073-4096	(10 extra bits)
24	11	4097-6144	(11 extra bits)
25	11	6145-8192	(11 extra bits)
26	12	8193-12288	(12 extra bits)
27	12	12289-16384	(12 extra bits)
28	13	16385-24576	(13 extra bits)
29	13	24577-32768	(13 extra bits)
30	-	-	(reserved/unused)
31	-	-	(reserved/unused)

Table 5: Distance symbol table

After the definition of the Huffman trees, the Huffman symbols encoding the data follow. The end of block symbol (256) indicates the end of the current block. A litlen

symbol encoding for a length is always followed by an offset symbol, to indicate the distance of the copyback operation.

DEFLATE compressed files are often in gzip format, a simple wrapper around a DEFLATE stream, that adds a header (with optional file name, comment, and extra fields) and a trailer containing CRC-32 and ISIZE (the original uncompressed size modulo 2^{32}).

A naive implementation of a DEFLATE decompressor would read the compressed bitstream bit by bit, traversing the Huffman trees to decode each symbol as shown in Figure 15.

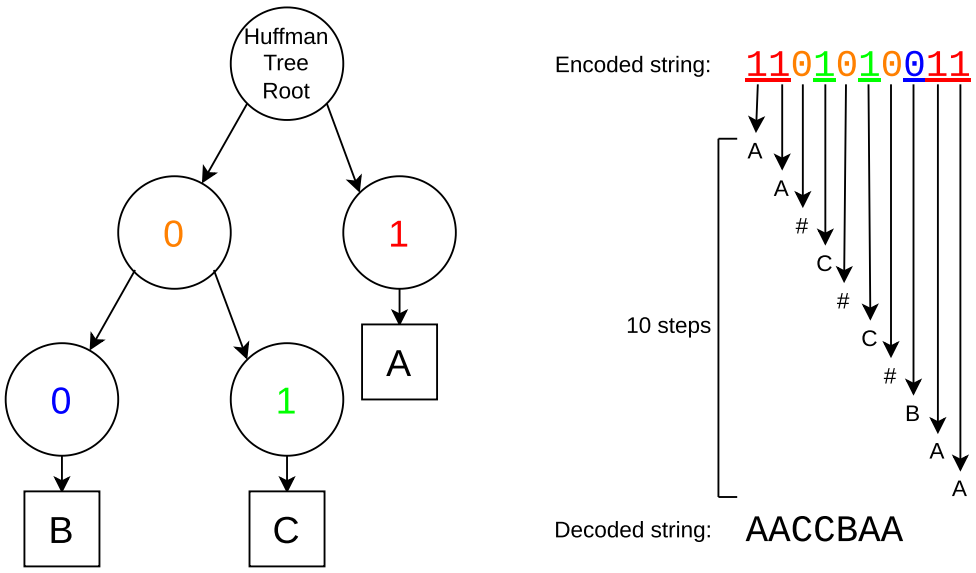

Figure 15: A visual representation of a naive Huffman decoding algorithm: each bit in the input sequence is processed individually, traversing the Huffman tree until a leaf node is reached. The bits in the input string are colored depending on which node of the tree they correspond to, and are underlined if they correspond to a leaf node.

While intuitive, this approach is quite inefficient, as it processes a single bit at a time, underutilizing the capabilities of modern processors. A much faster approach is described in the next section.

3.2.2 How libdeflate works

Libdeflate [48] is a C library that implements DEFLATE compression and decompression with an aggressive focus on speed. It targets the two common container formats

built around DEFLATE, zlib and gzip, and it is intended for whole-buffer operation rather than batch/stream processing. In practice this design, together with careful micro-architectural tuning, makes it substantially faster than zlib on modern x86 and ARM CPUs, and it is currently one of the fastest gzip decompression libraries. Since in GGCAT we only need to decompress gzip files, we will focus on the DEFLATE decompression part of libdeflate.

One of the central speed techniques used by libdeflate in decompression is the use of precomputed lookup tables to decode Huffman symbols. Table-driven decoding is a standard technique for prefix-free codes in general, including integer codes such as Elias gamma/delta, Golomb, Fibonacci, and variable-byte encodings [49]; in this setting it is applied to the canonical Huffman codes used by DEFLATE. That is, for each Huffman tree, libdeflate builds a lookup table with n bits (for some $n > 0$), mapping all 2^n possible n-bit combinations to the first Huffman symbol whose code is a prefix of those n bits. If the symbol is longer than n bits, a secondary fallback table is used to finish the decoding. The fallback case is quite rare, since longer Huffman symbols are by construction the least frequent ones, so the introduced overhead is negligible. This two-level decoding scheme ensures that most symbols can be decoded with a single table lookup, and compared to a naive bit-by-bit traversal of a Huffman tree, this reduces the number of operations and the branching dramatically and is critical for performance. An example of such a lookup table is shown in Figure 16.

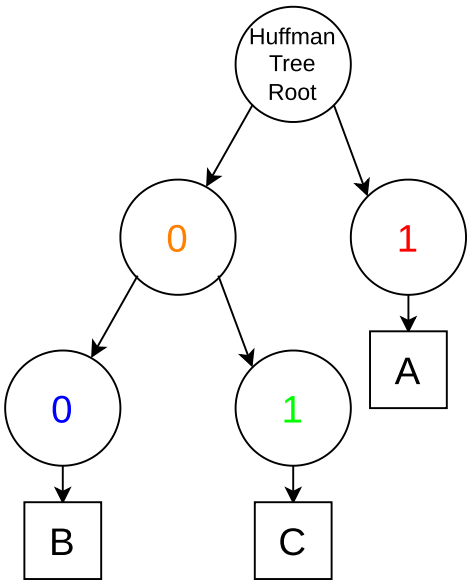

Table bits	Decoded symbol	Bits processed
000	B	2
001	B	2
010	C	2
011	C	2
100	A	1
101	A	1
110	A	1
111	A	1

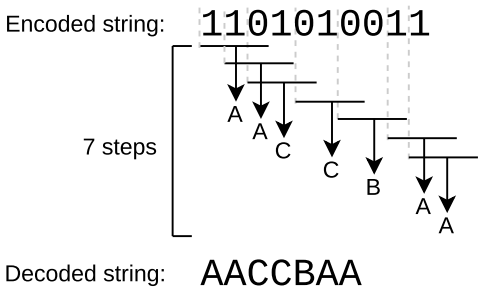

Figure 16: A visual representation of the optimized lookup table (with $n = 3$ bits) used by libdeflate. Every lookup can decode a full symbol by reading 3 bits at once, and the reading advances by the number of bits of the decoded symbol. In this example the decoding is completed in only 7 steps, compared to the 10 steps needed by the naive approach shown in Figure 15. Note that multiple entries in the lookup table can map to the same symbol, as all the combinations of bits after the symbol are valid continuations.

In practice, for each litlen dynamic Huffman tree, libdeflate creates one such lookup table with $n = 11$ that decodes a single litlen symbol. Then, for copyback operations, another lookup table is created with $n = 8$ to decode the offset of the operation.

Libdeflate also does many heuristic optimizations to improve branch predictions, such as defining a hot path to decode multiple literal symbols, fetching the data for the next table lookup while processing the current symbol (leveraging the ILP capability of modern processors) and using an almost branchless bit algorithm to update the index for subsequent data fetching.

3.2.3 Main libdeflate limitation

When processing a single large gzip member (>4GB), it is not possible to know the exact decompressed size, since the gzip header only contains the size modulo 2^{32}, so it is not possible to pre-allocate the exact output buffer size. With libdeflate's whole-buffer API this typically means guess -> allocate -> try -> reallocate until the decompressor succeeds. For multi-gigabyte genomic files, holding the entire decompressed result in memory is often unacceptable and repeated reallocations are expensive.

3.2.4 Streaming libdeflate optimizations

To overcome the limitations described in the previous subsection, a Rust rewrite of the decompression part of the library was performed, adding streaming support. In the rewrite, buffering was implemented with minimal performance impact and other optimizations were done to further improve performance. The buffering is implemented by tweaking the main decompression loop to check if there is enough input and output space available to continue decoding a symbol, and if not flush the output buffer and/or read more input data. To further improve performance, a new lookup table supporting decoding multiple symbols simultaneously was designed and implemented. An example of such a lookup table is shown in Figure 17.

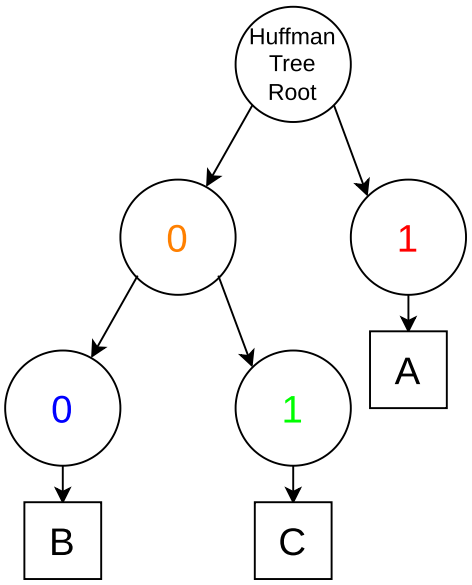

Table bits	Decoded symbol(s)	Bits processed
000	B	2
001	BA	3
010	C	2
011	CA	3
100	AB	3
101	AC	3
110	AA	2
111	AA	2

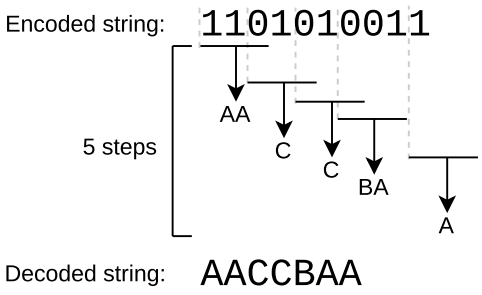

Figure 17: A visual representation of the optimized lookup table (with $n = 3$ bits) used by GGCAT. Every lookup can decode up to two full symbols, and the reading advances by the number of bits of the decoded symbols. In this example the decoding is completed in only 5 steps, as in many cases the table width is enough to decode more than one symbol with a single lookup.

3.3 Superkmers compaction

One of the main limitations of GGCAT 1 was the large amount of disk space required during the initial phase, where k-mers extracted from input files were partitioned into buckets. Although the intermediate bucket data was compressed, the compression algorithm used (LZ4) was not well suited for this type of data. In particular, LZ4's relatively short compression window prevented it from capturing similarities

between sequences that were added to buckets at different times. As a result, many redundant patterns remained uncompressed, especially in highly repetitive datasets such as bacterial pangenomes.

In such datasets, superkmers often appear multiple times, and can occur in different input files. Storing these repeated superkmers separately in the intermediate processing files leads to unnecessary duplication of data, which increases the required disk space and slows down the subsequent phases. To optimize disk usage, GGCAT 2 introduces a specialized compaction step that performs compression by recognizing and merging identical superkmers. That is, instead of storing the same sequence multiple times, each distinct superkmer is stored only once, along with a multiplicity counter indicating how many times it occurs in the dataset. This approach effectively captures superkmer redundancies that traditional compression algorithms like LZ4 cannot exploit (since equal superkmers can appear at a distance greater than the compression window), resulting in a substantial reduction in intermediate disk usage. This compaction approach is also applied in the colored case, where each superkmer now holds only a sorted list of the deduplicated colors instead of repeating the same superkmer for each repeated color.

The compaction process operates in parallel and is integrated directly into the bucketing algorithm. When a bucket grows beyond a specified size threshold, it is queued for compaction. Each queued bucket is processed independently: an in-memory hash map is built to store unique superkmers as keys along with their multiplicities. This enables efficient detection and merging of repeated sequences within the same bucket.

To further improve compression across related buckets, GGCAT 2 can selectively reprocess previously compacted bucket parts, as in many datasets (especially pangenomes) the sequences are very redundant even across different input files. If a newly compacted bucket contains superkmers that overlap with those already stored in earlier compacted data, these older bucket parts can be read again and integrated into the same hash map.

Since we need to read all the buckets to compact them, the compaction step takes advantage of that to further partition each bucket into sub-buckets, which can be processed independently in later phases of the pipeline. The sub-buckets are stored sequentially in each compacted bucket, so when a sub-bucket needs to be processed it can be found at specific file ranges of the compacted buckets. This hierarchical organization of buckets and sub-buckets enables both efficient disk usage and parallel computation, further reducing the working set of each subproblem. A schematic illustration of the structure of a compacted bucket is shown in Figure 18.

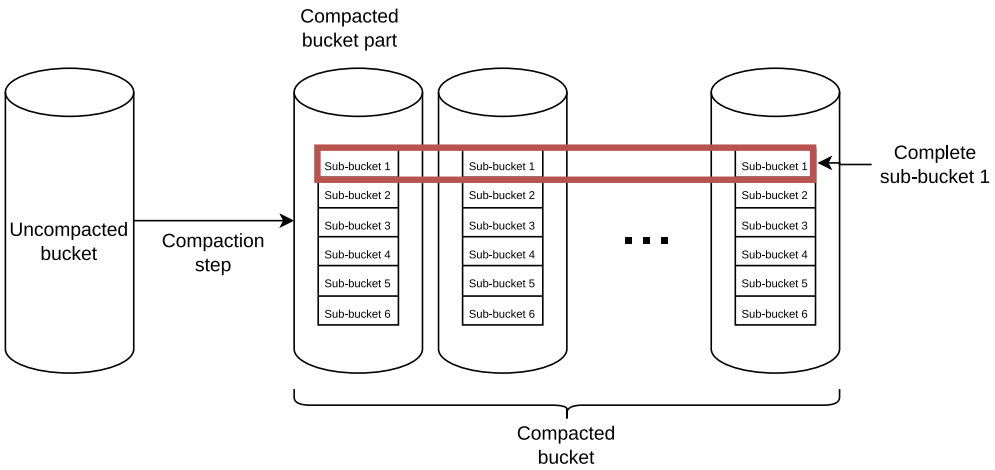

Figure 18: A compacted bucket structure. Each compacted bucket is composed of multiple files, each containing multiple sub-bucket parts. To completely read a sub-bucket, all the corresponding parts from each compacted bucket must be read and merged together, as shown in red in the figure.

Overall, this novel compaction strategy enables GGCAT 2 to handle much larger and more repetitive datasets using a fraction of the disk space previously required.

3.4 Integer storage format

A custom serialization format was developed to store intermediate data efficiently during processing. The design goals were to minimize disk usage while sustaining high I/O throughput. Because several fields, notably sequence lengths, can span wide numeric ranges, the format is built on a modified version of Google's unsigned varint scheme.

3.4.1 Background: Varint encoding

The Google Varint encoding [50] is a variable-length encoding for unsigned integers that can store arbitrary 64-bit integers using a variable (from 1 to 10) number of bytes. In this format, each byte uses the 7 least significant bits to store bits of the integer, while the most significant bit is used as a continuation flag: If it is set to 1, it indicates that there are more bytes to read, while if it is set to 0, it indicates that this is the last byte of the encoded integer. This encoding is very efficient since $\frac{7}{8}$ of the space is used for storing useful data and only $\frac{1}{8}$ is used to store serialization information in all but the last byte. In particular, if the number to encode is less than 128 (so it uses at most 7 bits), it fits in a single byte. The following table shows how many bytes are used to encode integers in different ranges:

Integer Range	Needed bytes
$0..2^7-1$	1
$2^7..2^{14}-1$	2
$2^{14}..2^{21}-1$	3
$2^{21}..2^{28}-1$	4
$2^{28}..2^{35}-1$	5
$2^{35}..2^{42}-1$	6
$2^{42}..2^{49}-1$	7
$2^{49}..2^{56}-1$	8
$2^{56}..2^{63}-1$	9
$2^{63}..2^{64}-1$	10

3.4.2 How GGCAT's integer storage format works

GGCAT extends the Google Varint integer encoding format by adding a custom set of boolean flags (depending on the type to serialize) as the most significant bits of the first byte of each serialized integer. These flags encode per-element metadata specific to the data type: for example, in the first phase they are used to encode the presence of extra bases at the superkmer endings, while for color differences they indicate whether a given difference value is run-length encoded. This allows better space usage since small integers can be stored along with boolean flags while occupying a single byte. The number of boolean flags that can be stored is chosen at compile time depending on the type to serialize, so no extra bits are wasted to store the number of flags. An example of such encoding is shown in Figure 19.

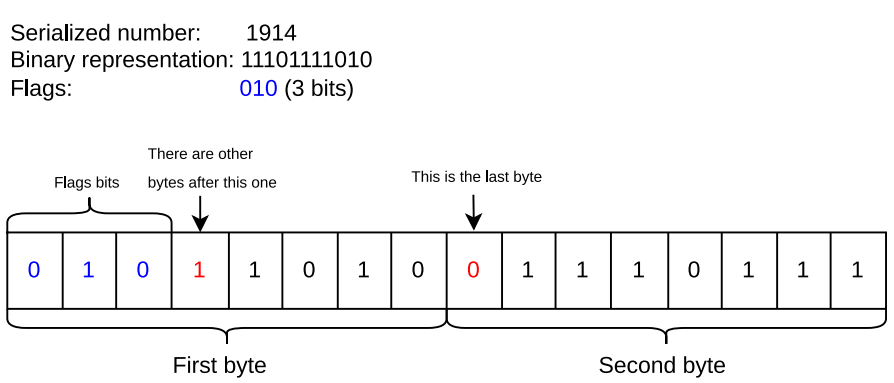

Figure 19: An example of the integer serialization format used in GGCAT, with three boolean flags. Since the flags are stored in the most significant bits of the first byte, in this case only the first 4 least significant bits are stored in the first serialized byte. The multibyte sequence stores successive payload groups in little-endian base-128 order.

3.5 Dynamic resizable vector optimization

During the compaction step and in color handling, it is necessary to store many vectors of variable size, to keep track of either the found superkmers or the current colorsets. Allocating many standard library vectors is very inefficient, as for each vector a call to the global allocator is required, and the allocator would have to deal with many very small allocations leading to memory fragmentation and an unacceptable overhead. To improve performance, a specialized vector was developed, using a custom allocator implementation that is more suited to handle such workflows allowing for a much faster allocation without increasing the memory usage overhead. The basic idea for the allocator is to have a single allocated area that is partitioned into ranges that are powers of two. Using this strategy, memory can be reused easily, since there is only a logarithmic number of possible allocation sizes. On reallocation or freeing, the old memory is added to a free list, indexed by the $\log_2$ of the size of the allocation, and on allocation the free list is checked first for a suitable memory range. While rounding up the allocation size to the next power of 2 can use up to double the memory, in practice this is an acceptable tradeoff. A visual representation of the allocator structure is shown in Figure 20. Another optimization done in this vector type is a small buffer optimization (SBO): if the data type to store is small enough, its bytes are stored directly inside the vector structure fields (e.g., repurposing the pointer and length fields), avoiding any allocator call for small vectors while keeping the size of the structure unchanged.

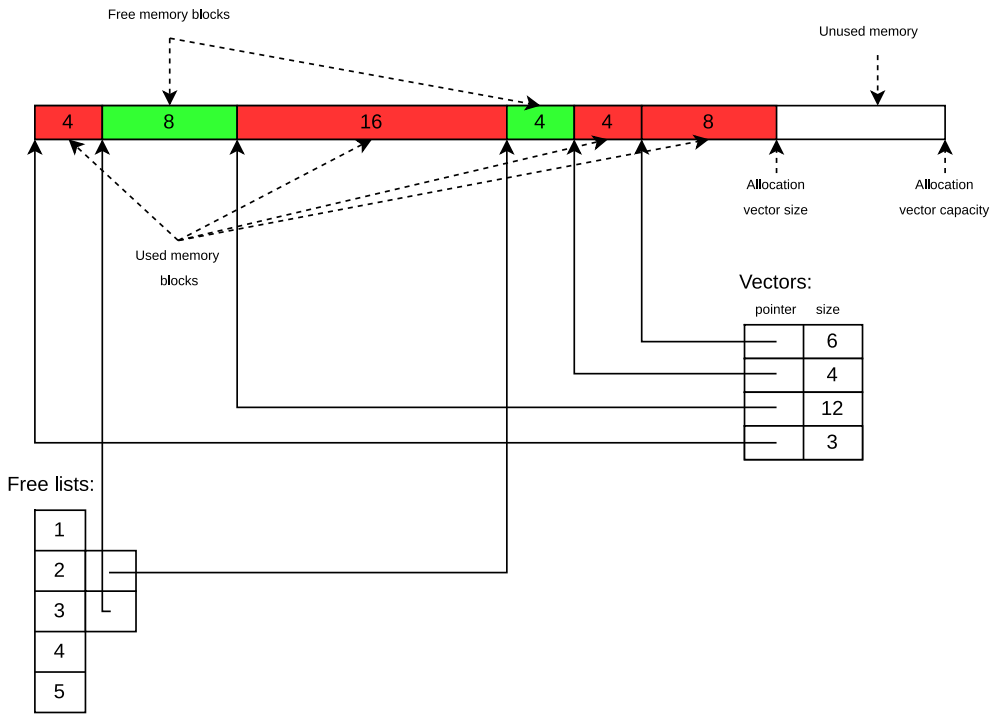

Figure 20: The structure of the custom allocator used for dynamic resizable vectors in GGCAT. The allocator pre-allocates a large memory area that is then reserved for the vectors. When a memory area is freed, it is added to a free list. As vector pointers are relative (i.e. array indices), when the memory area is exhausted it can be simply reallocated to a larger area using the system allocator.

3.6 Hash map optimization

In GGCAT hash maps are heavily used, since they are efficient and allow for $O(1)$ access to their elements. To further reduce the size of the subproblems to solve during the partial unitig extension algorithm, the superkmers are grouped according to their minimizer, and the algorithm is tolerant to minimizer collisions; therefore, using a standard hash map adds a performance penalty by performing collision resolution that can be avoided. For this reason, GGCAT uses a custom hash map that is indexed not by elements but just by their hash value, with each entry of the hash map containing a dynamic vector as described in the previous section. This enables grouping superkmers faster, by accepting collisions and keeping an optimized dynamically-sized vector of superkmers sharing the same minimizer.

3.7 Parallel architecture

To achieve high levels of parallelism and scalability, GGCAT 2 employs a custom library (*parallel-processor-rs*) that provides an in-memory filesystem with asynchronous writing and reading capabilities, along with a message-based parallel processing framework. This architecture greatly simplifies the development of parallel algorithms, by abstracting away the synchronization and communication details, allowing fast and safe parallelism.

3.7.1 In-memory filesystem

The developed in-memory filesystem features an asynchronous flushing queue to write files to disk when the memory usage exceeds the available memory, while allowing full in-memory processing when enough memory is available. This process is more efficient than the default OS filesystem caching, since it lets the library user define which files should be kept in memory, potentially optimizing for files that will be read soon instead of files that have just been written. Another advantage is that if a file is kept in memory, it is never copied around, as the memory chunks are directly read and written by producers and consumers of the files. The filesystem memory consists of a fixed size pre-allocated memory pool that is divided into chunks of a fixed size. When a file is created, a new chunk of this pool is reserved for the file, and if the file grows more chunks are allocated accordingly. When the memory pool is full or the file is marked as flushable, the filled chunks are sent to a flushing queue and written to disk asynchronously. When a file is read, it is either read directly from memory or by using an intermediate buffer if it was flushed to disk, allowing maximum performance in both cases. A simplified representation of the in-memory file structure is shown in Figure 21.

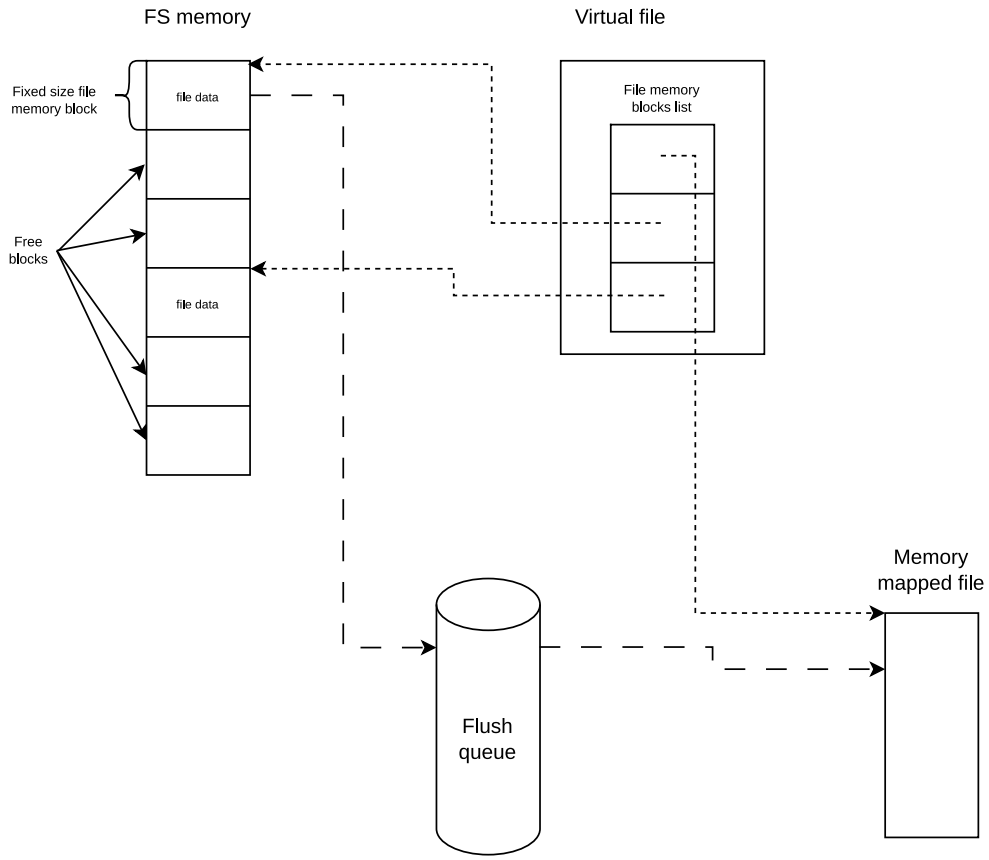

Figure 21: The structure of a file handled by the in-memory filesystem. Each file is composed of multiple fixed size chunks, that can be either referencing the global FS memory pool or the memory from a memory-mapped file on disk. To ensure compatibility with the OS filesystem (i.e. to make sure that it can be read normally after it is completely flushed), each file block is written to disk in the same order as it is in the in-memory representation. A flushing queue is used to asynchronously write filled chunks to disk.

3.7.2 Parallel message framework

The *parallel-processor-rs* crate also implements a message-based parallel processing framework, where jobs can be defined as having a type of input message and a type of output message. The scheduling is based on the concept of addresses and packets: An address is a serialized stream of messages that should be processed and can be added asynchronously by other threads. The execution pipeline is composed of multiple workers that at first wait for addresses to be available, then for each address they read and process all the messages in it (until the address sender indicates that no more

messages will be sent) and then wait again for new addresses. The workflow of the asynchronous message communication is shown in Figure 22.

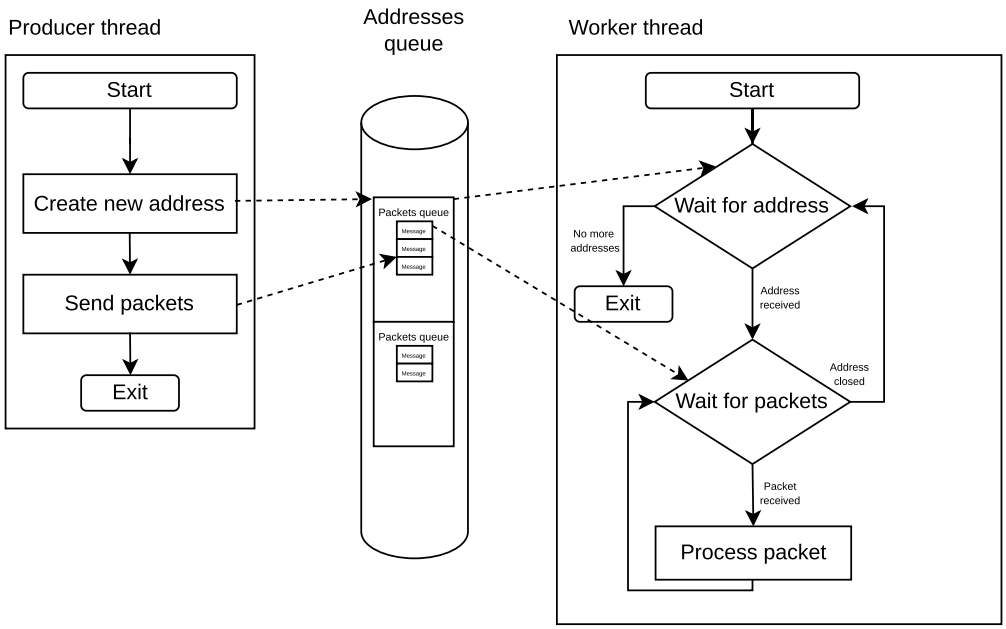

Figure 22: The schema of the message-based processing. A producer can generate an address (with its own queue) and send messages to it asynchronously. A worker can then dequeue the address and process sequentially all the messages in it.

3.8 Improved unitigs extension algorithm

In the first version of GGCAT, the hash-map-based method used to extend unitigs (described in Section 2.2.2) required a large amount of memory to store all the k-mers of a group, as every k-mer had its corresponding entry in the hash map. Furthermore, to perform a single base extension of a unitig, up to 8 lookups had to be performed in the hash map (one for each possible forward and backward branching). This turned out to be a bottleneck in both memory usage and runtime for some datasets, so a new algorithm was designed to mitigate these issues and avoid storing all k-mers explicitly.

The new algorithm is a multi-phase process that aims at keeping the k-mers encoded as superkmers for as long as possible. It starts in the first partitioning phase by ensuring that all the superkmers are given a unique orientation and alignment, then in the extension phase uses this additional guarantee to find regions that share

the same set of superkmers, knowing that all the k-mers in that region share the exact same properties.

3.8.1 Superkmers orientation and alignment

The first step of this new algorithm is to ensure that, whenever possible, the superkmers are oriented and aligned in a unique way. This is achieved by normalizing the superkmers while they are written to disk in the first phase (Section 2.2.1). Normalizing a superkmer means giving it a unique orientation and alignment such that all the equal k-mers are always written and aligned in the same way. Given a superkmer, it should first be determined whether it can be written in a unique orientation and alignment, and this is achieved by checking two properties:

- It should have a unique minimizer.
- The minimizer should have distinct forward and backward hashes.

The first property is needed to ensure that the superkmer can be properly aligned, as the minimizer is used as a reference point to offset the superkmers so that equal k-mers are always aligned. A visual example can be seen in Figure 23.

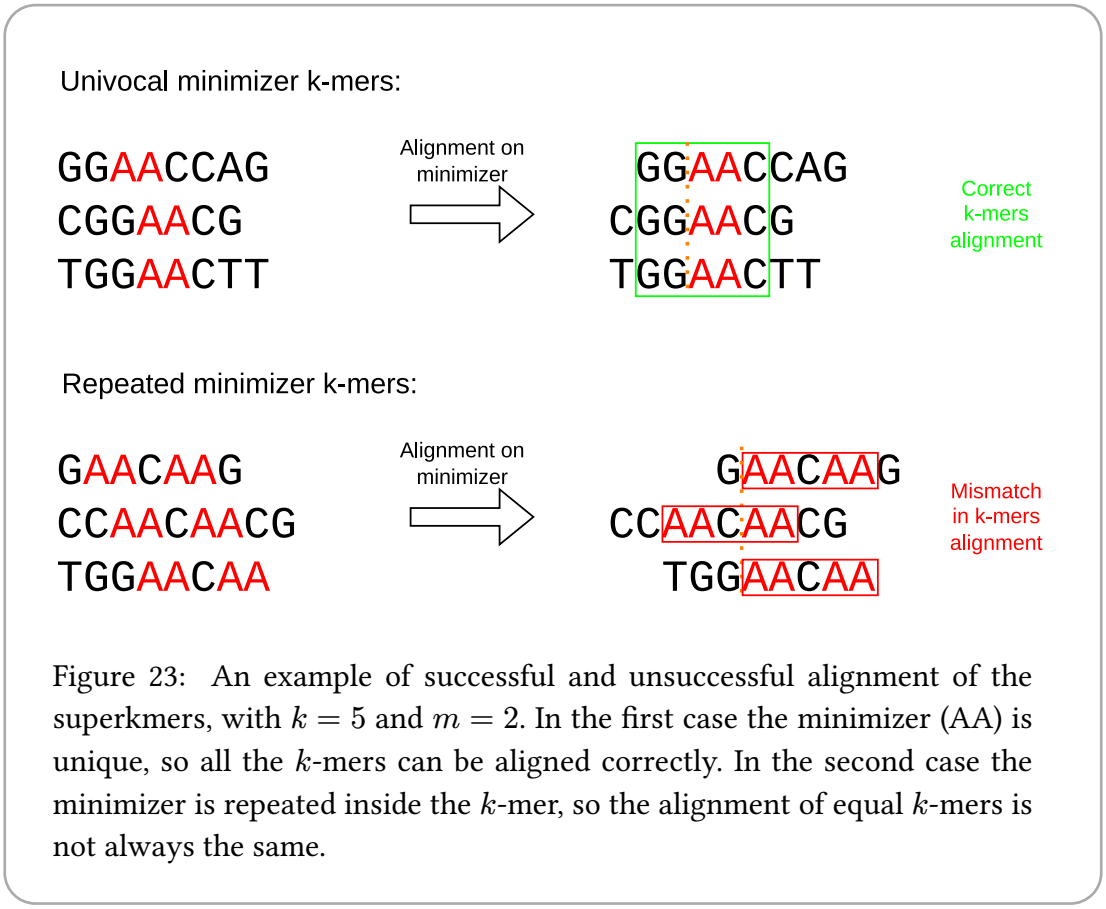

Figure 23: An example of successful and unsuccessful alignment of the superkmers, with $k = 5$ and $m = 2$. In the first case the minimizer (AA) is unique, so all the k-mers can be aligned correctly. In the second case the minimizer is repeated inside the k-mer, so the alignment of equal k-mers is not always the same.

The second property is needed in order to give a unique orientation to canonical k-mers (Section 1.3.2), by choosing the direction that minimizes the hash value. Fixing a direction of the minimizer guarantees that the whole k-mer is always represented in a unique way. An example of this is shown in Figure 24.

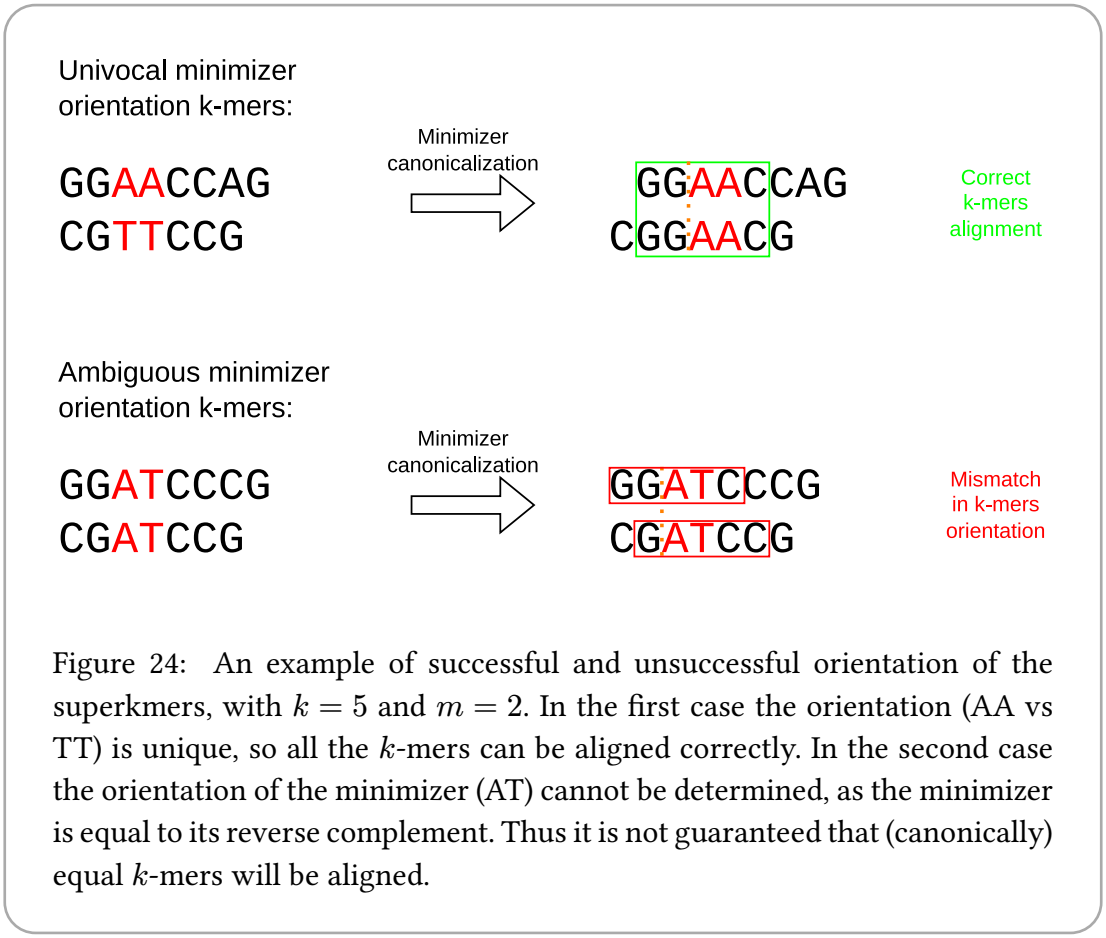

Figure 24: An example of successful and unsuccessful orientation of the superkmers, with $k = 5$ and $m = 2$. In the first case the orientation (AA vs TT) is unique, so all the k-mers can be aligned correctly. In the second case the orientation of the minimizer (AT) cannot be determined, as the minimizer is equal to its reverse complement. Thus it is not guaranteed that (canonically) equal k-mers will be aligned.

After normalizing a superkmer, it is written to disk in the correct orientation and with an alignment such that its minimizer is always in the forward orientation and its start is aligned to byte boundaries.

When it is not possible to normalize a superkmer, it is put in a special bucket that will be processed with the previous algorithm, which does not require alignment.

3.8.2 Extension phase

When all the superkmers sharing a minimizer are correctly aligned, a new algorithm that exploits this alignment can be used. The objective of this algorithm is to find maximal *stabletigs*, which are ranges of k-mers that share the exact same source set of superkmers, such that their properties (i.e. multiplicity and colors) are equal. Another

nice property of stabletigs is that they are non-branching, so they are also parts of unitigs.

To ease the algorithm explanation, we will now define two different substrings of a superkmer: the *reverse aligned prefix* and the *aligned suffix*. Given a superkmer S of length n with its minimizer starting at a 1-based position p, we define the aligned suffix $a_s = \mathit{suf}_{n-p}(S)$ and the reverse aligned prefix as $a_{rp} = \mathrm{rev}\big(\mathit{pre}_{p-1}(S)\big)$, where rev reverses the character order of the input string.

To compute stabletigs, the superkmers are first aligned on their minimizer, then they are sorted by their aligned suffix, as shown in Figure 25.

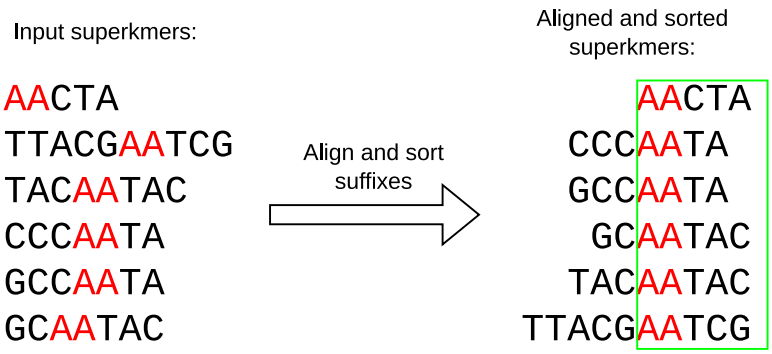

Figure 25: The preliminary step of the extension phase, all the superkmers are aligned on their minimizer and sorted lexicographically according to their aligned suffix.

After this step, the algorithm is organized in two functions, the first function creates groups of superkmers with equal aligned suffixes, while the second function processes these groups by trimming the aligned suffixes and creating stabletigs.

The first function groups superkmers by their matching aligned suffix using a stack-based algorithm, that creates a group for each aligned suffix match. Since the actual algorithm is a bit involved, an example of the groups produced with an explanation is shown in Figure 26.

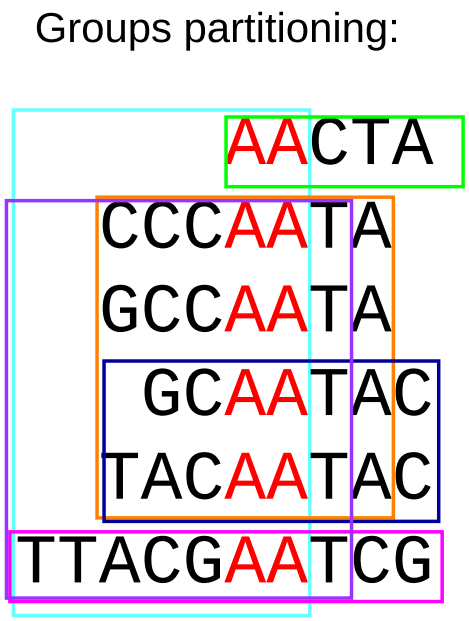

Figure 26: In this example, the algorithm starts by finding the green group (with aligned suffix AACTA). Then it tries to extend the aligned suffix AATA but the extension is suspended because the fourth superkmer has a longer matching aligned suffix (AATAC). Thus this aligned suffix is processed first to produce the blue group. Then the extension of the AATA aligned suffix can continue and the orange group is produced. After that the purple group is created, with AATCG as aligned suffix. Finally the last two groups (purple and light blue) are created, with aligned suffixes AAT and AA respectively.

The task of the second function is to process a single group, so that its superkmers get their aligned suffix trimmed on the right by a required amount. In other words, this means that we need to process all the k-mers that end in positions we want to trim. The level at which one group should be processed is found by looking at the maximum match between the group aligned suffix and the aligned suffixes of the superkmers right above and below the current group. An example of how the trimming level is chosen is presented in Figure 27.

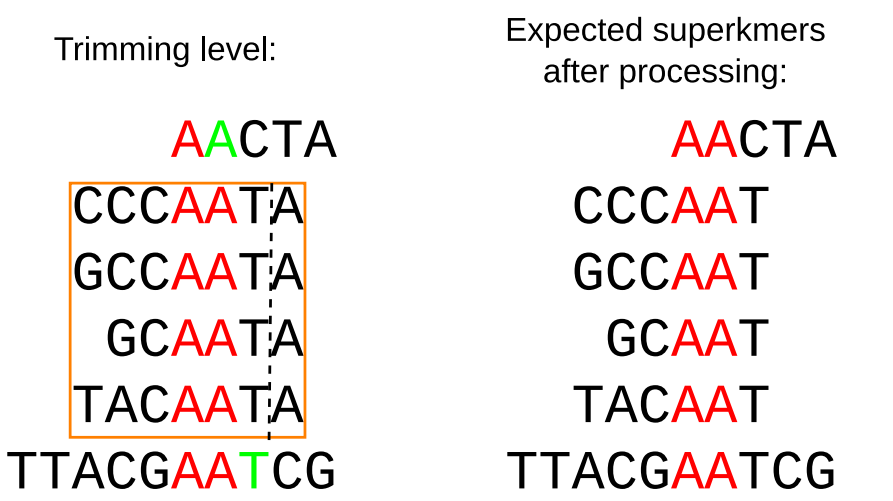

Figure 27: An example of trimming level and the remaining superkmers after the processing of the orange group. The last matching bases of the two superkmers above and below the group are highlighted in green. Since the rightmost of the two bases corresponds to an aligned suffix of three bases, the processing of the orange group should trim the superkmers so that they have an aligned suffix that is exactly three bases long.

The second function first sorts the superkmers lexicographically by their reverse aligned prefix. Then for each k-mer q it needs to process, it finds all the superkmers containing it, along with the maximum common reversed aligned prefix (starting from the center going left). If it is found that more than k total characters match between all the superkmers sharing the target k-mer q, it means that we found a stabletig larger than one k-mer. An example is shown in Figure 28.

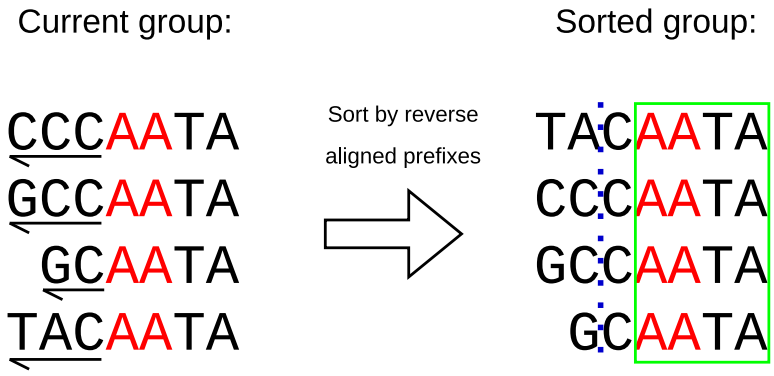

Figure 28: The first step for the second function is to sort the superkmers by their reverse aligned prefix. After that, considering $k = 4$, the first k-mer to be processed is AATA. The algorithm then finds all the superkmers that share this particular k-mer (since they are sorted they will be all consecutive), then finds the longest extension possible that is shared among all the superkmers containing that k-mer. In this case the candidate stabletig will be CAATA. Finally, a check is performed to see if the stabletig is allowed to be this long: the algorithm checks whether its leftmost k-mer ends after the required trimming level; if not, the stabletig is shrunk. In this case the trimming level shrinks the stabletig, as the leftmost k-mer CAAT ends at a position not allowed by it. Thus, the final stabletig produced is just AATA, and every superkmer is trimmed by one base, as shown in Figure 27.

After finding all the stabletigs, links between them are followed by tracking the stabletigs that were produced before by the same superkmers. The stabletig generation is the same for unitigs and simplitigs, while the joining rule changes according to the requested output. The algorithm accumulates old and new stabletigs while consecutive processed ranges still share a $(k-1)$-mer boundary, then applies the selected joining rule when this boundary chain ends. For unitigs, a boundary chain is crossed only when exactly one stabletig ends and exactly one stabletig starts there, because any other case would be branching. For simplitigs, branching is allowed, so the algorithm greedily pairs old and new stabletigs in their current order.

The implementation first computes the longest common centered suffix (LCCS) of every adjacent pair in aligned suffix order. A stack then enumerates suffix-sharing ranges from the longest suffix to the shortest one. Within each range, the superkmers are put in reverse-aligned-prefix order and the corresponding adjacent longest

common centered prefix (LCCP) values are used to identify all superkmers containing the same k-mer. The pseudocode below mirrors the implemented algorithm, but omits the merge-sorting details used to reuse already sorted chunks, which is an implementation optimization that does not change the produced stabletigs. Ranges are half-open: R.start is the first contained index and R.end is the first index after the range.

```
Sorting-based unitig extension(superkmers, k, a, mode):
sort superkmers by aligned suffix
n ← |superkmers|
LCCS[0] ← 0; LCCS[n] ← 0
LCCS[i] ← common centered suffix length of superkmers i − 1 and i for 1 ≤ i < n
prefix_rank ← ranks obtained by sorting by reverse aligned prefix
remaining[S] ← length of the aligned suffix of each S
skip[i] ← 1 for 0 ≤ i < n // used to skip whole ranges sharing the same remaining suffix
stack ← empty stack
i ← 0
while not stack.is_empty() or i < n:
    if stack.is_empty():
        G ← new range with G.start ← i and G.end ← i + 1
        stack.push((G, 0)); i ← i + 1
    (G, p) ← stack.top(); ℓ ← remaining[G.start]
    // while the next superkmer suffix matches at least ℓ characters
    while G.end < n and LCCS[G.end] ≥ ℓ:
        if remaining[G.end] > ℓ:
            H ← new range with H.start ← G.end and H.end ← G.end +1
            // process longer endings first, trimming them so that they will share exactly ℓ bases
            stack.push((H, ℓ))
            continue the outer loop
        G.end ← G.end + skip[G.end]
        stack.set_top((G, p)) // update the stack top with the new range
    reorder superkmers in range G by prefix_rank
    t ← max(p, LCCS[G.end]) // compute the maximum allowed trimming
    ProcessPrefixRange(G, ℓ, t, k, a, mode)
    for j ← G.start to G.end −1:
        remaining[j] ← t
```

```
    skip[G.start] ← G.end − G.start
    i ← G.end
    if t = p: stack.pop() // reached the target suffix trimming level
for each stabletig u without a predecessor: // output unitigs/simplitigs
    w ← u
    while u has a successor:
        u ← successor of u
        append u to w overlapping by k − 1 bases
    output w as a partial unitig
```

```
ProcessPrefixRange(G, ell, t, k, a, mode):
    LCCP ← common reverse-aligned prefix lengths of adjacent superkmers in range G
    prefix_stack ← empty stack
    prefix_stack.push((G, ℓ))
    while not prefix_stack.is_empty():
        (H, s) ← prefix_stack.pop()
        R_l ← partitions of H into maximal consecutive ranges sharing a k-mer
        O ← empty list of previously created stabletigs
        N ← empty list of newly created stabletigs
        for each range R in R_l: // iterate each range and find the longest allowed stabletig
            // remember the links to the previous computed stabletigs from the current superkmers
            append previous stabletig mappings for every superkmer in range R to O
            p ← min value in LCCP[R] // find the maximum shared reverse aligned prefix match between all superkmers in R
            l ← max(k − p, t + 1) // never exceed the maximum trim level
            u ← substring from suffix position l through s
            // add up and join colors and multiplicity
            multiplicity ← sum of the multiplicities in range R
            colors ← union of the colors in range R
            if |u| ≥ k and multiplicity ≥ a:
                create stabletig u with multiplicity and colors
                map every superkmer in range R to the new stabletig
                append u to N
```

```
22      otherwise clear their stabletig mappings
23      if R.end = H.end or there is a (k − 1)-mer break at R.end:
24          JoinStabletigs(O, N, mode)
25          clear O and N
26      if l ≠ t + 1: prefix_stack.push((R, l − 1))
```

```
JoinStabletigs(O, N, mode):
1   remove missing entries and duplicates from O
2   if O is empty or N is empty: return
3   if the first new stabletig or the first old stabletig crosses an external extra
    base: return
4   if mode = unitigs:
5       if |O| = 1 and |N| = 1: link the only stabletig in O to the only stabletig
        in N
6   otherwise if mode = simplitigs: // greedily join stabletigs
7       for each pair (u, v) in zip(O, N): link u to v
```

3.9 Memory-bounded unitigs merging algorithm

In the previous algorithm, the extension and final unitigs building part was divided in four different steps:

- Hashes sorting, which joined the hashes corresponding to the paired unitigs endings to produce pairs of unitig indices to be merged together
- Links compaction, to merge unitig indices together
- Reads reorganization, to resolve the indices and retrieve the original partial unitigs
- Unitigs building, to write the final unitigs file

In practice, for large datasets the hashes ended up occupying more disk space than the partial unitigs. This is because (except for some rare cases) partial unitigs built in the k-mers merge phase can contain only a single minimizer, so each unitig is almost always less than $2k - m$ bases long. If a partial unitig needs to be joined on both sides, it will have two entries in the hashes, one for each ending k-mer, thus occupying an equivalent space of $2k$ bases. Furthermore, hashes are not compressed on disk so their space usage is even larger.

In GGCAT 2 all these steps are unified into a single *unitigs merging* phase. Instead of storing pairs of unitig indices and later retrieving their sequences, the algorithm joins the partial-unitig sequences directly and writes a sequence as soon as both

its extremities are closed. Consequently, the large intermediate list of uncompressed extremity hashes and the reads-reorganization phase are no longer needed.

A problem arising from this change is that now the carried sequences are larger (usually by a small multiplicative factor) than the indices used before, so peak memory usage from this change was higher when using datasets having a very large output graph. To reduce the peak memory usage, the algorithm was tweaked to allow two things:

- Resplit the sequences more effectively to lower the number of sequences kept in memory at the same time, while keeping the same parallelism level
- Use again indirect indices but only for the largest unitigs, to avoid loading them in memory and carrying them around in this phase

The core idea is to put a sequence in the bucket determined by the smaller of the indices produced by the unsealed right and left endings of the sequence. With this partitioning, the smallest-index bucket contains all sequences with an ending having that bucket index.

This allows the sequential processing of each bucket, from the smallest to the largest index, knowing that after a bucket is processed, all the endings having that bucket index are completely joined. If, after merging, a sequence still has an ending belonging to another bucket, the sequence is rewritten in that bucket so that it can be joined again later.

To process a bucket at a time while retaining parallelism, an additional *subsplit* step is performed. That is, each main bucket is divided into many smaller subpartitions according to further bits of the selected extremity hash. Thanks to checkpoints in the bucket, this step can be scheduled as independent tasks, and the number of subpartitions is chosen based on the bucket size. Each subpartition can then be loaded, joined, and written independently, so only a small fraction of the dataset is in memory at any time and many subpartitions can be processed concurrently.

The processing of the subpartitions closely resembles the original links compaction phase, with the difference that now most matching endings are already in the same subpartition, thus speeding up the process. The main exception happens when one of the sequences has both endings assigned to the same bucket number. With n buckets, this occurs on average for only $\frac{1}{n}$ of the sequences in a bucket, corresponding to $\frac{1}{n^2}$ of the total sequence count: at least one ending has the bucket index, and the probability that the other has it too is $\frac{1}{n}$. By carefully choosing the number of buckets based on the estimated output size, we can keep this number extremely low. For this reason, instead of putting the sequences in subpartitions randomly when we have multiple subpartition choices for the same sequence, the subsplit operation uses a shared hash map indexed by the two open extremity k-mers of the unitig. This hash map is consulted only for unitigs whose two open extremities both belong to the current main bucket. It stores, for each extremity, whether the extremity should be

preferred or avoided in this round. When a unitig is routed through one extremity, that extremity is marked as preferred, while the other extremity is marked as avoided. The avoided mark records that no matching unitig is expected in the corresponding subpartition during this round, and prevents later unitigs from being routed where the join is known to fail. The same subsplit procedure is used both for the initial bucket contents and for the residual unitigs produced by a joining round.

In the pseudocode, $\mathrm{SubCount}(n, t)$ denotes the number of subpartitions chosen for an input of size n and thread budget t, $\mathrm{Bucket}(u)$ is the minimum main bucket among the open endings of unitig u, and $\mathrm{Part}(q, p)$ is the subpartition index determined by extremity q among p subpartitions. The shared hash map A stores only routing hints: $+$ means that the extremity should be selected if possible, and $-$ means that the extremity should be avoided if the other extremity is available. The subpartition index is always computed deterministically as $\mathrm{Part}(q, p)$ from the selected extremity q.

```
Memory-bounded unitigs merging(B, k, t):
 1  D_b ← empty deferred file for every main bucket b
 2  for each main bucket b in increasing order:
 3     A ← empty shared hash map from extremity k-mers to {+, −}
 4     U ← B_b ∪ D_b
 5     P ← Subsplit(U, b, t, A)
 6     while P is not empty:
 7        S ← empty list of unitigs that remain in bucket b
 8        for each P_i ∈ P as an independent task:
 9           (C_i, R_i) ← ProcessSubpartition(P_i, k)
10           emit every unitig in C_i
11           for each u ∈ R_i:
12              b′ ← Bucket(u)
13              if b′ = b: append u to S
14              otherwise append u to D_b′
15        if S is empty: P ← empty list
16        otherwise:
17           P ← Subsplit(S, b, t, A)
```

Subsplit(S, b, t, A):

$p \leftarrow \mathrm{SubCount}(|S|, t)$
$P \leftarrow p$ empty subpartition files
for each partial unitig $u \in S$ as an independent task:
  $(q_l, q_r) \leftarrow$ the left **and** right open extremities of u
  if only q_l belongs **to** bucket b:
    $q \leftarrow q_l$; $j \leftarrow \mathrm{Part}(q, p)$
  otherwise **if** only q_r belongs **to** bucket b:
    $q \leftarrow q_r$; $j \leftarrow \mathrm{Part}(q, p)$
  otherwise **if** $A[q_l] = +$:
    $q \leftarrow q_l$
  otherwise **if** $A[q_r] = +$:
    $q \leftarrow q_r$
  otherwise **if** $A[q_l] = -$:
    $q \leftarrow q_r$
  otherwise **if** $A[q_r] = -$:
    $q \leftarrow q_l$
  otherwise:
    $q \leftarrow q_l$
  $j \leftarrow \mathrm{Part}(q, p)$
  if $q = q_l$ **and** q_r belongs **to** bucket b:
    $A[q_l] \leftarrow +$; $A[q_r] \leftarrow -$
  otherwise **if** $q = q_r$ **and** q_l belongs **to** bucket b:
    $A[q_r] \leftarrow +$; $A[q_l] \leftarrow -$
  append (u, q) **to** P_j
return P

ProcessSubpartition(P_i, k):

$J \leftarrow$ empty local hash map from extremity k-mers **to** partial unitigs
$C \leftarrow$ empty completed list; $R \leftarrow$ empty residual list
for each $(u, q) \in P_i$:
  if q is absent from J: store u **in** $J[q]$
  otherwise:
    $v \leftarrow$ remove $J[q]$
    $w \leftarrow$ Join(u, v, q)
    if Closed(w): append w **to** C
    otherwise choose a remaining open extremity of w **and** append w **to** R
for each unmatched u remaining **in** J:
  if Circular(u): remove the duplicated overlap **and** append u **to** C
  otherwise choose a remaining open extremity of u **and** append u **to** R
return (C, R)

3.9.1 Indirect representation of long unitigs

Joining the partial-unitig sequences directly creates another potential source of high memory usage: a single partial unitig can become very long before it is completed. Copying the whole sequence after every join would make both the memory consumption and the joining cost proportional to the length of the growing unitig. To avoid this, sequences shorter than a threshold $T = \max(4 * k, 4096)$ are stored *inline*, while the internal parts of longer sequences are moved to an append-only file containing all the oversize unitigs. This threshold is chosen so that the unitig parts can be read from disk in random order without large performance penalties and also to ensure that the indirect unitig part is greater than or equal to the part that remains inline (k bases for each ending).

An indirect partial unitig keeps only an inline prefix and suffix, together with an ordered list of references to the removed internal parts. Each reference stores the file offset, the encoded color-data length, the sequence length, and one bit indicating whether the part must be reverse-complemented when reconstructed. The value `indirection_start` separates the inline prefix from the inline suffix. Thus, the logical sequence is reconstructed by concatenating the prefix, all referenced parts in order, and the suffix. Color data is split and reconstructed in the same way.

When joining two inline unitigs, they are merged on their common k-mer. If the resulting inline sequence exceeds T, its internal range is written to the oversize file while the first and last k bases remain inline, as shown in Figure 29. This keeps both extremities immediately available for hashing and comparison in later joins.

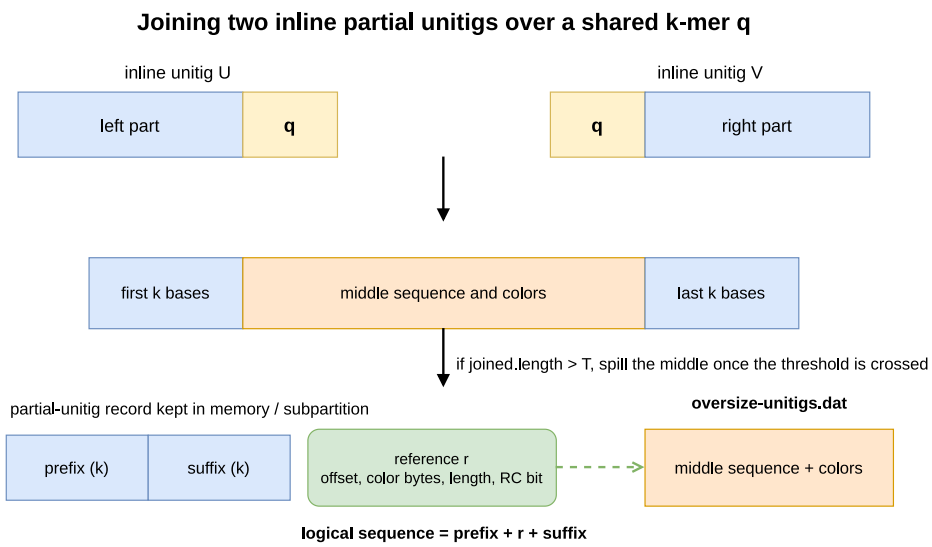

Figure 29: Conversion from an inline to an indirect partial unitig. After joining at the shared k-mer q, the middle sequence and its colors are written to the oversize file if the total sequence size exceeds the threshold T. The active record retains the two extremities (that are used for further joins) and a compact reference to the removed part.

If an inline unitig is joined to an indirect one, the existing references are copied without reading their sequence data. The new inline bases are appended to the appropriate extremity, and another internal part is spilled only if the enlarged inline region exceeds T. When two indirect unitigs are joined, their inner inline suffix and prefix form a bridge around the shared k-mer. This bridge is written as one new indirect part, placed between the two existing reference lists, while only the outer prefix and suffix remain inline. These cases are summarized in Figure 30. While this can allow the creation of very small indirect parts, they are always surrounded by indirect parts with size larger than T, so the average sequence length of the indirect parts is still larger than $\frac{2T}{3}$.

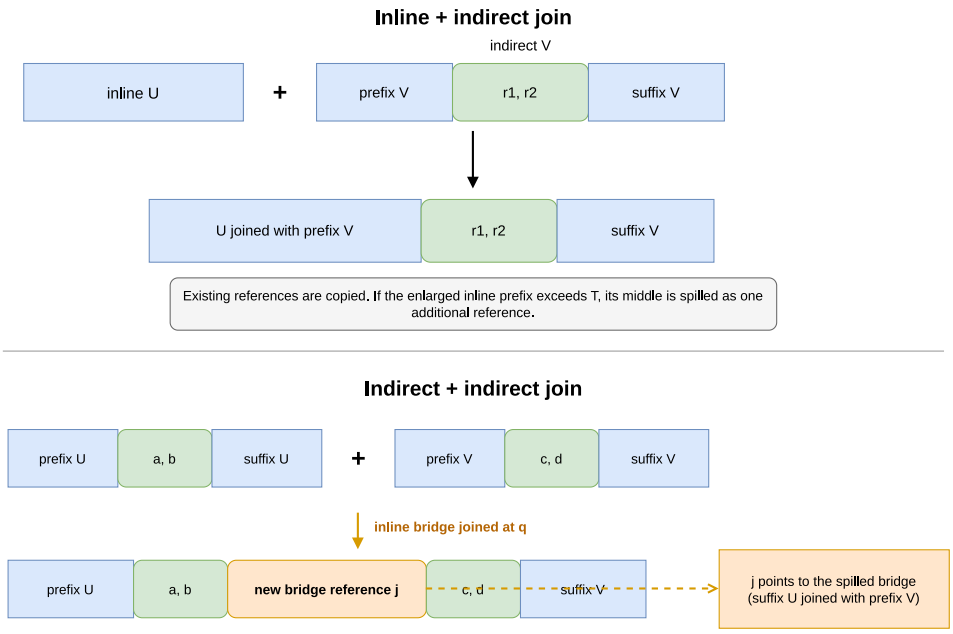

Figure 30: Joining unitigs that already contain indirections. In an inline–indirect join, existing references can be reused directly. In an indirect–indirect join, the two inner inline regions are joined and spilled as a new bridge reference between the original reference lists.

This representation makes joining depend mainly on the bounded inline extremities and the number of references, rather than on the total unitig length. Reverse complementation also does not require rewriting the spilled sequence: the reference order is reversed and the orientation bit of each reference is flipped. The full sequence and colors are read from the oversize file only when the final unitig is emitted.

3.10 Minimizers resplitting

In some datasets, it is possible that a particular minimizer is highly repeated, requiring a large amount of superkmers to be processed simultaneously. If not handled carefully, this can lead to a huge increase in memory and CPU usage, making the bucketing strategy much less efficient in splitting the problem into manageable subproblems. To account for this case, when a bucket has a size much larger than the average, a resplitting is performed, where the superkmers are partitioned again using a longer minimizer. To ensure that each new minimizer has a reasonably low amount of superkmers associated with it, it is chosen as: $m = k - 10$. In this way, each k-mer has exactly 10 bases not included in the minimizer, so each minimizer has at most $4^{10} * 11 = 11534336$ unique k-mers.

3.11 Fast Simplitigs

The first implementation of simplitigs used the matchtigs library [51], which employs a graph-based algorithm looking at the de Bruijn graph nodes corresponding to the unitig extremities and choosing which unitigs to merge. This library uses an in-memory algorithm not optimized for very large dataset sizes, so the simplitigs computation had a much lower practical limit for dataset size. In order to speed up the construction of simplitigs and support much larger datasets, the extension algorithm has been embedded in the k-mer extension phase, by extending a sequence as long as possible without checking for branching. A stylized example of the simplitig extension strategy is shown in Figure 31.

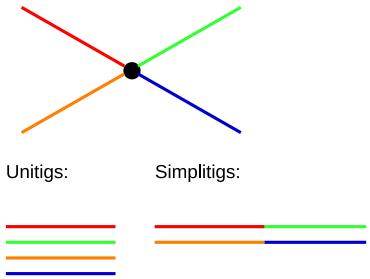

Figure 31: An example of unitigs vs simplitigs extension. In the unitigs case, the algorithm stops extending when it finds a branching node. When creating simplitigs, in case of a branching node, any non-visited outgoing edge is taken and the extension continues.

3.12 Fast Eulertigs

As the eulertigs computation used the same library used for computing simplitigs, it had the same performance limitations. Thus an improved algorithm for computing eulertigs was developed, as a post-processing step on simplitigs to convert a greedy simplitig output into a minimum-size eulertig representation. As explained in the eulertigs discussion in Section 1.3.8, eulertigs are an optimal SPSS representation that minimizes the number of sequences produced. In the greedy output produced by the embedded simplitig extension, the remaining opportunities to reduce the number of strings are circular simplitigs that can be inserted into another simplitig at a shared node. Since the number of circular simplitigs is usually much lower than the

number of linear ones, the new algorithm separates all the circular simplitigs from the linear ones, processes the circular simplitigs in memory and then searches for possible insertion points of the circular simplitigs into the linear ones. A diagram of the algorithm is shown in Figure 32.

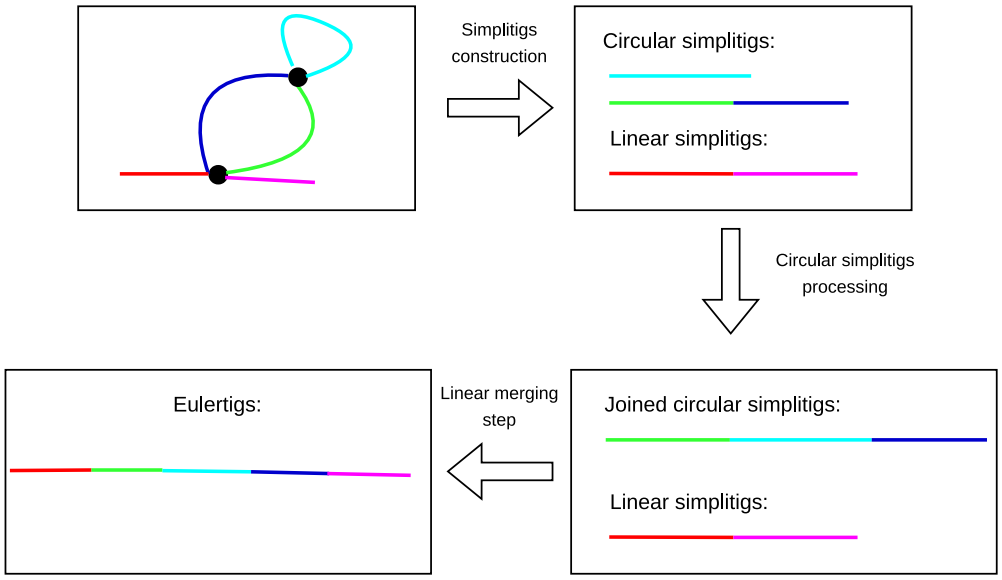

Figure 32: The steps of the new eulertigs building algorithm. After producing simplitigs and separating the circular from the linear ones, an intermediate step is performed to join all possible circular simplitigs. Finally, a merging phase of the circular over the linear simplitigs is performed, producing the final eulertigs.

3.13 Results

In this section we show the improvements of GGCAT 2 over GGCAT 1 on various dataset types and sizes for both uncolored and colored construction.

3.13.1 Experimental setup

All benchmarks in this chapter were performed on the same computational infrastructure described in Section 2.6 and use HDD storage. The Salmonella experiments were executed on the *small* server using 12 threads, while all other GGCAT 2 experiments were executed on the *large* server using 32 threads. The small server runs Ubuntu 22.04 LTS, while the large server runs Ubuntu 24.04 LTS. Because the large server has more cores than the configured thread count, the number of processors available to each run was constrained by binding the benchmark process and all of

its worker threads to a fixed CPU affinity mask containing the specified number of logical processors. This makes the configured thread count an effective upper bound on the hardware parallelism available to the tool, independently of the total number of cores on the machine. The benchmark harness also included deadlock detection: a run was stopped and marked as deadlocked if the average CPU utilization remained below 0.3 cores for more than one hour.

The datasets used in the following tables extend the naming introduced in Section 2.6. *Salmonella 100K* and *Salmonella 309K* are respectively a 100K subset and the full 309K EnteroBase Salmonella collection. *Human 100* and *Human 2505* are respectively the 100-genome subset and the full 2505-human-genome collection generated from GRCh37 and 1000 Genomes variants. *Gut* is the PRJEB33098 gut-microbiome read dataset, while *Tara50* is a larger Tara Oceans metagenomic read benchmark made of 50 input files [52]. *E. coli* is used as a small genome benchmark, and *AllTheBacteria* is an extension of the bacterial genomes dataset [46, 53] and is the largest bacterial-genome benchmark in this chapter.

3.13.2 Uncolored Salmonella genomes scaling

This benchmark analyzes resource consumption for uncolored construction on an increasing number of Salmonella genomes, ranging from 1K to 309K genomes. GGCAT 2 is consistently faster than GGCAT 1 across the scaling experiment, with an average runtime speedup of about 2.3x, while keeping the memory footprint in the same range. The largest improvement is in temporary disk usage, where the superkmer compaction step is especially effective because many sequences are repeated across pangenome samples. The scaling curves for time, memory and disk usage are shown respectively in Figure 33, Figure 34 and Figure 35.

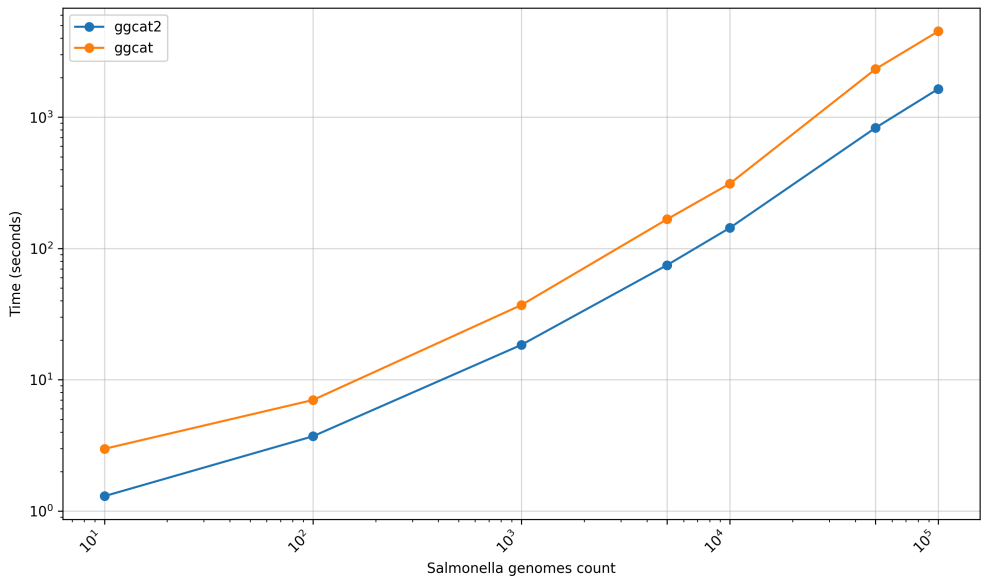

Figure 33: The required time to process an increasing amount of Salmonella genomes. GGCAT 2 outperforms GGCAT 1 in every dataset size, with an average 2.3x speedup.

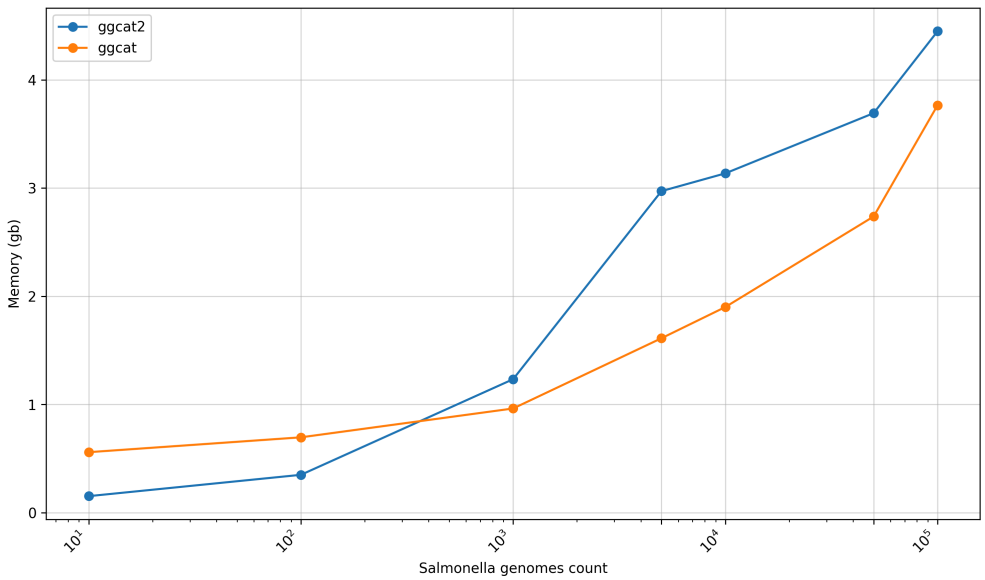

Figure 34: The required maximum working memory to process an increasing amount of Salmonella genomes. The GGCAT 2 memory footprint is similar to that of GGCAT 1, with modest increases in some cases due to the compaction step requiring a larger minimum amount of memory.

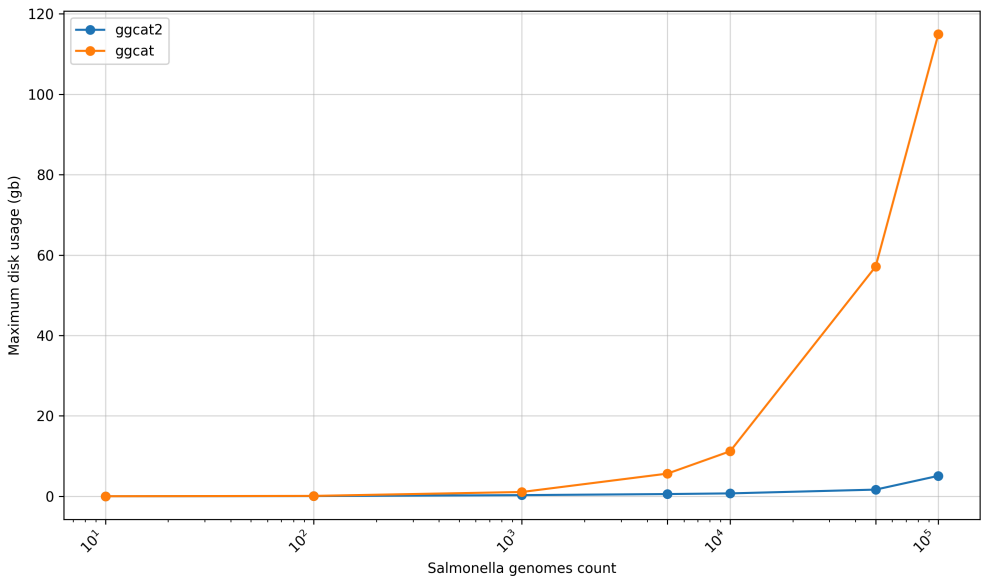

Figure 35: The required maximum disk utilization to process an increasing amount of Salmonella genomes. The GGCAT 2 disk usage is much lower than GGCAT 1, thanks to the compaction step that is particularly effective in case of pangenomes.

The final HDD Salmonella construction results are summarized in Table 6. In these runs, GGCAT 2 is between 1.7x and 2.4x faster than GGCAT 1 and uses between 22x and 39x less peak disk space. Cuttlefish 3 is much slower than GGCAT 2 on the 100K subset, and its runs on the full archive crashed.

Dataset	k	Cuttlefish 3	GGCAT	GGCAT 2
Salmonella archive (100K)	27	3h:15m (3.17GB) [518.02GB]	1h:18m (2.55GB) [167.59GB]	41m:24s (4.33GB) [4.34GB]
	63	1h:38m (3.23GB) [214.98GB]	1h:18m (2.91GB) [114.57GB]	32m:54s (4.08GB) [5.08GB]
Salmonella archive (309K)	27	*crashed*	4h:04m (3.57GB) [390.76GB]	2h:23m (5.73GB) [10.51GB]
	63	*crashed*	4h:12m (4.35GB) [279.45GB]	2h:04m (5.46GB) [11.66GB]

Table 6: Uncolored Salmonella construction, wall clock time, memory usage (in parentheses) and maximum disk usage including the output files [in square brackets], using 12 threads on the small server and HDD storage.

3.13.3 Uncolored construction

The uncolored construction results are shown in Table 7. On genome datasets, GGCAT 2 is between 1.5x and 2.8x faster than GGCAT 1, with an average speedup close to 2x. The disk reduction is even larger, ranging from about 5x on the smallest cases to more than 70x on the largest human dataset. Read collections are less dominated by repeated pangenome structure, so the runtime gains are smaller: at $k = 27$, GGCAT 2 is roughly comparable to GGCAT 1 on both read datasets; at $k = 63$, Tara50 reaches about 1.7x speedup, while Gut is slightly slower. In both read datasets, GGCAT 2 still reduces disk and memory usage. Among the completed Cuttlefish 2 runs, GGCAT 2 is usually several times faster, and Cuttlefish 2 fails on the largest bacterial benchmark at $k = 27$. Cuttlefish 3 improves over Cuttlefish 2 on some smaller cases, but it deadlocks or is killed on several of the larger human and read benchmarks.

Dataset	k	Cuttlefish 2	Cuttlefish 3	GGCAT	GGCAT 2
E. coli	27	3m:26s (2.95GB) [19.37GB]	4m:25s (4.51GB) [21.72GB]	1m:15s (2.57GB) [11.13GB]	0m:36s (3.81GB) [1.32GB]
	63	6m:39s (2.93GB) [14.60GB]	2m:13s (4.71GB) [9.17GB]	1m:18s (2.47GB) [7.48GB]	0m:28s (3.61GB) [1.48GB]
Human 100	27	48m:22s (4.59GB) [283.99GB]	*deadlock*	16m:07s (6.71GB) [197.55GB]	10m:44s (7.04GB) [11.97GB]
	63	1h:54m (4.39GB) [211.45GB]	*deadlock*	13m:13s (8.17GB) [126.54GB]	8m:04s (7.44GB) [12.82GB]
Human 2505	27	33h:16m (38.77GB) [7155.51GB]	*deadlock*	17h:47m (48.12GB) [5261.04GB]	9h:20m (12.60GB) [74.69GB]
	63	43h:01m (30.37GB) [5364.46GB]	*deadlock*	19h:15m (69.19GB) [4387.69GB]	8h:00m (12.02GB) [91.48GB]
Gut	27	12m:18s (4GB) [78GB]	*killed*	7m:20s (11GB) [80GB]	6m:30s (7GB) [30GB]
	63	33m:10s (4GB) [107GB]	8m:28s (106GB) [68GB]	5m:56s (10GB) [51GB]	6m:31s (6GB) [31GB]
Tara50	27	8h:18m (149GB) [1702GB]	*killed*	5h:31m (63GB) [1711GB]	5h:22m (16GB) [1190GB]
	63	7h:21m (128GB) [1018GB]	*killed*	3h:26m (63GB) [1073GB]	2h:04m (10GB) [836GB]
AllTheBacteria	27	*crashed*	*n/a*	25h:38m (24.67GB) [3933.07GB]	14h:23m (18.41GB) [283.85GB]
	63	76h:29m (174.97GB) [7730.59GB]	*n/a*	20h:30m (24.47GB) [2567.62GB]	11h:54m (14.73GB) [287.45GB]

Table 7: Uncolored construction, wall clock time, memory usage (in parentheses) and maximum disk usage including the output files [in square brackets], using 32 threads. The Gut and Tara50 values refer to unitig construction in the read benchmark set.

3.13.4 Simplitigs and eulertigs construction

The simplitig and eulertig construction results are shown in Table 8. In GGCAT 1, producing simplitigs and eulertigs was substantially more expensive than producing unitigs, because it used an independent library that employed a full in-memory algorithm. By embedding simplitig construction into the main extension phase, GGCAT 2 makes the additional runtime almost disappear: on the Gut benchmark it is between 1.4x and 3.0x faster than GGCAT 1, while using about an order of magnitude less memory. The larger Tara50 dataset highlights the scalability improvement: GGCAT 1 was killed when computing both simplitigs and eulertigs, while GGCAT 2 completed all configurations. For simplitigs, GGCAT 2 is also faster than Cuttlefish 2 on all reported read benchmarks, and additionally supports eulertig construction.

Dataset	k	Representation	Cuttlefish 2	GGCAT	GGCAT 2
Gut	27	Simplitigs	12m:19s (4GB) [78GB]	11m:52s (78GB) [80GB]	6m:28s (7GB) [20GB]
	27	Eulertigs	*n/a*	19m:32s (102GB) [80GB]	6m:33s (7GB) [20GB]
	63	Simplitigs	32m:44s (4GB) [107GB]	9m:16s (63GB) [51GB]	6m:31s (6GB) [31GB]
	63	Eulertigs	*n/a*	14m:33s (87GB) [51GB]	6m:35s (6GB) [31GB]
Tara50	27	Simplitigs	8h:17m (149GB) [1702GB]	*killed*	5h:16m (15GB) [1190GB]
	27	Eulertigs	*n/a*	*killed*	5h:27m (19GB) [1190GB]
	63	Simplitigs	7h:04m (128GB) [2267GB]	*killed*	2h:04m (10GB) [836GB]
	63	Eulertigs	*n/a*	*killed*	2h:12m (12GB) [836GB]

Table 8: Simplitig and eulertig construction, wall clock time, memory usage (in parentheses) and maximum disk usage including the output files [in square brackets], using 32 threads.

3.13.5 Colored construction

The colored construction results in Table 9 show the same trend. When compared to GGCAT 1, GGCAT 2 is between 1.2x and 2.5x faster, again averaging close to a 2x speedup. Disk reductions are less uniform than in the uncolored case because the output colorsets can dominate some runs, but they remain substantial on the human datasets, on the 100K Salmonella subset, and on Tara50. Memory usage also drops sharply on the largest human colored benchmark and on Tara50. Cuttlefish 3 colored construction completes on the 100K Salmonella subset, where it is more than 3x slower than GGCAT 2, but it crashes on the full Salmonella archive, deadlocks on the human colored benchmarks, and is killed by the memory limit on Tara50.

Dataset	k	Cuttlefish 3 colored	GGCAT colored	GGCAT 2 colored
Salmonella archive (100K)	27	4h:37m (6.16GB) [563.49GB]	2h:28m (5.17GB) [180.92GB]	1h:18m (5.85GB) [57.41GB]
	63	3h:04m (6.57GB) [257.79GB]	2h:16m (4.99GB) [119.74GB]	55m:42s (4.92GB) [52.04GB]
Salmonella archive (309K)	27	*crashed*	7h:58m (7.84GB) [421.60GB]	5h:45m (9.08GB) [192.63GB]
	63	*crashed*	7h:38m (7.25GB) [290.01GB]	3h:34m (8.49GB) [252.67GB]
Human 100	27	*deadlock*	31m:57s (12.09GB) [214.14GB]	17m:55s (8.67GB) [51.98GB]
	63	*deadlock*	27m:30s (14.82GB) [152.10GB]	11m:00s (8.23GB) [24.01GB]
Human 2505	27	*deadlock*	30h:18m (177.25GB) [5554.51GB]	18h:54m (41.61GB) [723.34GB]
	63	*deadlock*	28h:33m (204.16GB) [4536.62GB]	12h:25m (22.96GB) [322.58GB]
Tara50	27	*killed*	8h:35m (100GB) [1863GB]	7h:14m (55GB) [1335GB]
	63	*killed*	5h:27m (67GB) [1153GB]	2h:30m (22GB) [878GB]

Table 9: Colored construction, wall clock time, memory usage (in parentheses) and maximum disk usage including the output files [in square brackets]. The Salmonella rows use 12 threads on the small server; the Human and Tara50 rows use 32 threads on the large server.

4 Project Architecture, Testing, and Profiling

4.1 Overview

In this chapter we describe the details of the GGCAT implementation from a software engineering perspective, focusing on the code organization and the techniques used to ensure the correctness and performance of the tool.

4.2 Code organization

The organization of the codebase is an important aspect of software development, as it affects readability and maintainability, but also compile time and runtime performance. While Rust offers a much more streamlined code organization compared to C/C++, it still has some freedom in the way the code is divided into modules and/or crates. With a heavy use of generics, the division of the code into multiple crates is fundamental to keep compile times low, as with many generics the amount of generated code is multiplied by the Cartesian product of all the generic parameters. Especially while developing, it is better to recompile as little code as possible in case of small changes in order to have faster feedback cycles, and having multiple crates is very helpful in this regard.

4.2.1 Rust project layout

A Rust project is by default managed using Cargo [54], Rust's official package manager and build system. A Cargo project has a very simple structure, with a `Cargo.toml` file at the root of the project containing all the needed metadata and dependency definitions. All the project source code is contained in the `src` folder, with either a `main.rs` file (for binary projects) or a `lib.rs` file (for library projects) as the main entry point. Every other file inside the `src` folder is a module that can be declared with the `mod` keyword. All the dependencies of the project are automatically downloaded, with crates.io being the default package registry for Rust, where anyone can publish their own crates (Rust packages). Contrary to C and C++, the basic compilation unit (the smallest partition of code that needs to be recompiled when a change is made) in Rust is the whole crate, instead of single module files. This can increase compile times significantly if the crate contains many modules, as a change in a single module requires recompiling the whole crate. To specify compile-time options, Rust uses features, which are opt-in compile-time flags that can be used to enable or disable parts of code using conditional compilation. The careful planning of features is important as it can reduce compile times significantly, especially while

developing, by limiting the amount of code compiled. Another key point of Rust code is the possibility of running a `build.rs` script at compile time, which can be used to dynamically generate code or perform other operations needed before compiling the main code. A `build.rs` script can be instructed to run only on specific file/environment changes, and this is very important to avoid unnecessary runs of the script that in turn will often trigger a full recompilation of the crate.

To compile a basic Rust project, it is enough to run `cargo build` in the root folder of the project, and Cargo will automatically download the necessary dependencies and compile the code, producing the final artifacts in the `target` folder. Many different build options can be specified, such as the build profile (debug by default, or release using the `--release` flag that enables optimizations) and the desired feature flags (`--features` flag).

4.2.2 GGCAT Crates organization

To keep compile times low, GGCAT uses a project structure with multiple crates, with the most generic ones published on crates.io and the specific ones kept in the `crates` folder of the main GGCAT repository. The main crates developed for GGCAT and published on crates.io are:

- *dynamic-dispatch-rs*: A crate to implement dynamic generics dispatch to improve compile times. It is described in detail in Section 4.2.4.
- *mt-debug-counters-rs*: A crate that implements fast multi-threaded debug counters, where each thread has its own counter to avoid contention, and the final value is computed by summing all the per-thread counters. It is described in Section 4.4.3.
- *nightly-quirks-rs*: A crate that implements functions currently only available on nightly Rust, to allow using them on stable Rust and keep them even in case of deprecation.
- *parallel-processor-rs*: The backbone of GGCAT's parallel architecture and memory filesystem. It is described in Section 3.7.
- *streaming-libdeflate-rs*: The streaming version of libdeflate used in GGCAT, described in detail in Section 3.2.

The crates developed specifically for the GGCAT pipeline and not published on crates.io are:

- *api*: The main GGCAT API crate, which exposes all the main functions to use GGCAT as a library from a Rust project.
- *assembler*: The main assembler crate, which implements the whole pipeline to compute colored and uncolored compacted de Bruijn graphs.
- *assembler-kmers-merge*: The crate that implements the partial unitigs construction algorithm.
- *assembler-minimizer-bucketing*: The crate that implements the partitioning of reads into superkmers and their bucketing according to minimizers.

- *assembler-pipeline*: Downstream pipeline crate that implements the merging of partial unitigs and simplitigs/eulertigs construction.
- *capi*: The C API crate that exposes the GGCAT API to C/C++ projects.
- *cmdline*: The command line interface crate, which implements the `ggcat` binary. It internally calls the functions exposed by the *api* crate.
- *colors*: Structs and functions to handle colorsets in their various representations.
- *config*: Global configuration constants such as default buffer sizes and min/max values for various parameters.
- *dumper*: Crate implementing the dumping of a compacted de Bruijn graph, to allow integration with other tools. Its functions are exported in the *api* crate.
- *hashes*: Hash function implementations used in GGCAT. Currently provides implementations for canonical ntHash, identity hash (encoding the value of a k -mer directly as a hash value) and Rabin-Karp hashing, used for larger values of k.
- *io*: Various I/O helper functions, used to serialize and deserialize GGCAT internal data structures and handle compressed DNA sequences.
- *sequence-output*: Support for writing output sequences in FASTA, GFA1/2 and binary formats.
- *kmers-transform*: Base library used to perform a parallel transformation on k -mers stored in buckets. Currently used for the assembly partial unitigs construction and for querying the graph.
- *logging*: A crate that unifies GGCAT logging, exposing it to the *api* crate to allow interception of log messages by other tools using the API.
- *minimizer-bucketing*: Functions to handle sequences partitioning into superkmers. Used by both the assembler and querying pipeline.
- *querier*: Querying pipeline crate, implementing batch querying of k-mers and sequences from a colored or uncolored compacted de Bruijn graph.
- *structs*: Common internal structures used in GGCAT.
- *utils*: Common utility functions and data structures used in GGCAT.

4.2.3 Generics

To optimize runtime performance in both space and time, GGCAT contains numerous specializations for the various algorithms, choosing different hash functions, color representations and output formats depending on the input parameters. To avoid runtime overhead, these specializations are implemented using Rust generics, allowing the compiler to monomorphize the code for each specific combination of parameters at compile time. The disadvantage of this approach is that it increases compile times and generated code size significantly, but in the following section it is described how GGCAT mitigates these issues using dynamic dispatch. As an example of specialization, hash functions are decided based on k with the following logic:

- For $k \leq 16$ a 32-bit identity hash is used
- For $16 < k \leq 32$ a 64-bit identity hash is used

- For $32 < k \leq 64$ a 128-bit identity hash is used
- For $k > 64$ a 128-bit Rabin-Karp hash is used.

In this case the specialization ensures that the memory used is optimized for the selected k value. Another use of generics is to specialize between canonical and non-canonical processing, which requires the use of different hash functions, and the optional handling of colors.

4.2.4 Dynamic dispatch

An early implementation of GGCAT used the classical static generics monomorphization approach, where every time a function is called it triggers the generation of code for all the child generic functions, especially if most of them are inlined in the caller function. This can lead to very long compile times, which are very detrimental to development speed, as even small changes can lead to a recompilation lasting minutes. In this case splitting the code into multiple crates alone does not help, as a generic function is compiled only in the crate where it is called, similarly to header-only C++ templates. To overcome this issue, GGCAT uses a custom dynamic dispatch crate (*dynamic-dispatch-rs*) that precompiles all the possible combinations of generic parameters for a function, by having the function annotated with a custom attribute macro that declares all the desired combinations of generic parameters. This has two main advantages: first, in case of recompilation of an upstream crate, the downstream crates do not need to recompile the generic functions again, as they are no longer generic at the crate boundary. The second advantage is that using conditional compilation, only the needed combinations of generic parameters are compiled, reducing the amount of generated code significantly during development, where typically only one parameter combination is needed for testing.

As an example, consider the declaration of the main assembly function in the *assembler* crate:

```
#[dynamic_dispatch(MergingHash = [
    #[cfg(all(feature = "hash-forward", feature = "hash-64bit"))]
hashes::fw_seqhash::u64::ForwardSeqHashFactory,
    #[cfg(all(feature = "hash-forward", feature = "hash-128bit"))]
hashes::fw_seqhash::u128::ForwardSeqHashFactory,
    #[cfg(feature = "hash-16bit")]
hashes::cn_seqhash::u16::CanonicalSeqHashFactory,
    #[cfg(feature = "hash-32bit")]
hashes::cn_seqhash::u32::CanonicalSeqHashFactory,
    #[cfg(feature = "hash-64bit")]
hashes::cn_seqhash::u64::CanonicalSeqHashFactory,
    #[cfg(feature = "hash-128bit")]
hashes::cn_seqhash::u128::CanonicalSeqHashFactory,
    #[cfg(feature = "hash-rkarp")]
hashes::cn_rkhash::u128::CanonicalRabinKarpHashFactory,
```

```
    #[cfg(all(feature = "hash-forward", feature = "hash-rkarp"))]
hashes::fw_rkhash::u128::ForwardRabinKarpHashFactory,
], AssemblerColorsManager = [
    #[cfg(feature = "enable-colors")]
colors::bundles::multifile_building::ColorBundleMultifileBuilding,
    colors::non_colored::NonColoredManager,
], OutputMode = [
    FastaWriterWrapper,
    #[cfg(feature = "enable-gfa")] GFAWriterWrapperV1,
    #[cfg(feature = "enable-gfa")] GFAWriterWrapperV2
])]
pub fn run_assembler<
    MergingHash: HashFunctionFactory,
    AssemblerColorsManager: ColorsManager,
    OutputMode: StructuredSequenceBackendWrapper,
>(
  ... Parameters ...
){
  ... Function body ...
}
```

This function is annotated with the `dynamic_dispatch` attribute macro which declares all the desired combinations of generic parameters for the function. To call this function, the caller can use the `assembler::dynamic_dispatch::run_assembler` function, that takes an extra tuple as the first parameter indicating the identifier of the desired generic parameters. It can be called like this:

```
assembler::dynamic_dispatch::run_assembler(
            (merging_hash_dispatch, colors_hash, output_mode),
            ... Other parameters ...
);
```

An identifier for the generic parameter is automatically generated by annotating the generic type trait with the `#[dynamic_dispatch]` attribute. To retrieve the identifier of a specific generic parameter, the static `dynamic_dispatch_id` function of the desired type can be used.

4.3 Testing

Correctness testing for GGCAT combines checks inside the Rust codebase with an external testing suite, available at https://github.com/Guilucand/testing-ggcat. This external suite can check the correctness in two different ways:

- By comparing the output from two different ggcat versions, at either canonical sequence level or k-mer level.
- With an extended check of graph correctness by parsing the input files and computing the k-mer set using an in-memory algorithm. This method also allows checking the correctness of the color sets, by remembering the origin of each k

-mer. This can be useful to check for regressions that introduce or miss k-mers and to check the equivalence of unitig, simplitig, and eulertig representations.

This output checking is useful for testing behavior that spans multiple crates and pipeline phases, where unit tests for individual components are not sufficient to catch regressions in the complete executable. Keeping the suite separate from the main implementation also makes it possible to compare different GGCAT binaries, ensuring development builds and released versions have a consistent behavior.

4.4 Profiling

4.4.1 Intel VTune

When runtime performance is a critical aspect of a program, it is useful to profile the execution to find bottlenecks and possibly optimize them. One tool that turned out to be very useful in profiling GGCAT is Intel VTune Profiler [55], a profiling suite that can analyze different performance aspects, such as thread usage, CPU stalls and cache misses, memory I/O profiles, code hotspots and many others.

4.4.2 Flamegraphs

For profiling CPU usage and function call times, flamegraphs can be used to obtain a visual representation of the call stack and the time spent in each function. The main tool used to generate flamegraphs in Rust is `cargo-flamegraph`, which integrates seamlessly with cargo and allows generating flamegraphs with a single command. To generate a flamegraph, it is enough to run the command `cargo flamegraph --bin <binary-name>` in the root folder of the project, and it will generate an interactive SVG file `flamegraph.svg`.

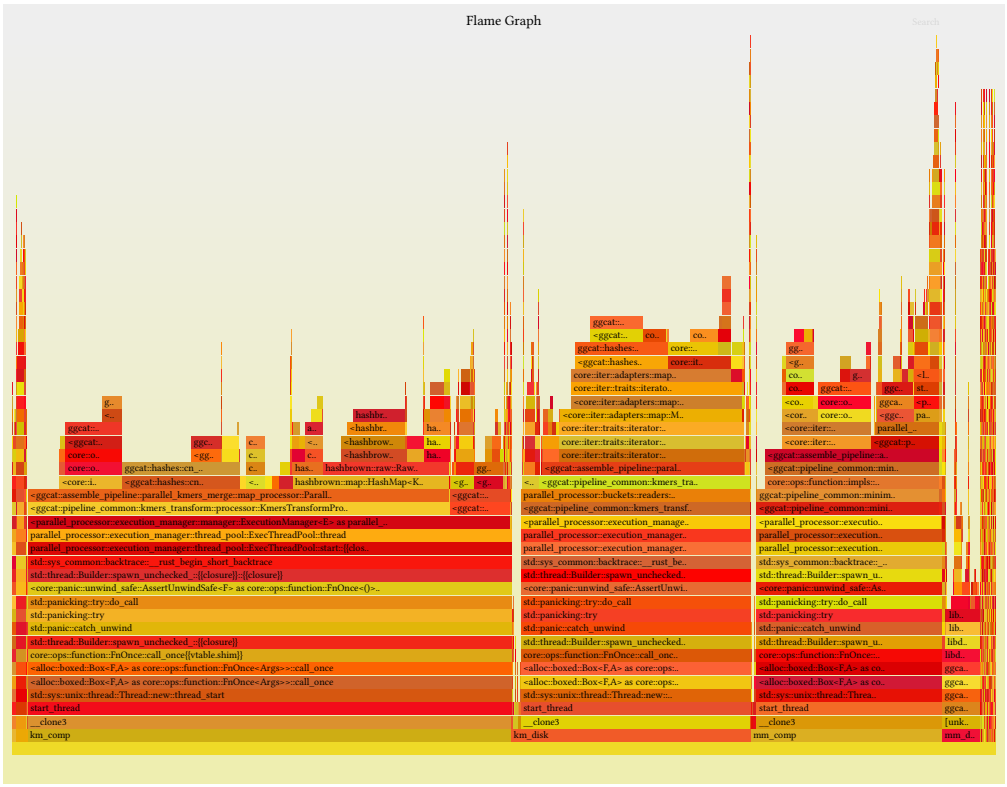

Figure 36: An example of a flamegraph generated using `cargo-flamegraph` while profiling GGCAT. The x-axis size represents the proportional time spent in each function, while the stacked bars represent nested function calls.

4.4.3 Optimized counters

To understand how many times a specific code path is executed, it is useful to have counters that can generate statistics about the execution. While incrementing a counter is a trivial operation, doing so in a multi-threaded context can introduce serious performance penalties: even with lock-free atomic operations, concurrent updates to the same counter must be serialized by the CPU, leading to stalls and slowdowns. To overcome this problem, a small library was developed that keeps a separate counter for each thread using thread-local storage, so there is no contention while updating the counters. When statistics are retrieved, another thread is responsible for summing all the local counters to produce the final result.

4.5 Debugging

A fundamental part of the development of any tool is the debugging phase, where bugs and unexpected behaviors are tracked and fixed. Debugging a complex multi-

threaded tool like GGCAT can be challenging, especially considering that often the bug can happen only in very specific conditions that are either probabilistic or require a long runtime before manifesting. To ease the debugging process, we used strategies and implemented command line options and tools that help in tracking down bugs.

4.5.1 Single phase replay

To help in tracking down bugs that happen only after a long runtime, GGCAT has the possibility of keeping the intermediate files of each phase on disk and replaying only a portion of the pipeline. This is achieved with the command line options `--step` and `--last-step`. To keep the intermediate files, the flag `--keep-intermediate-files` can be specified.

4.5.2 Dataset minimization

To find output bugs that happen only on specific datasets, it is useful to find a minimal example that triggers the bug, to reduce the amount of input data and allow a detailed analysis of the execution. By running GGCAT on portions of the input data (for example specific input files or specific regions of input files) it is possible to check if the bug still happens, and by iteratively reducing the input data a minimal example can be found.

4.5.3 Rust panic and assertions

Contrary to other system programming languages like C and C++, Rust has built-in checks for array bounds and other memory safety issues, which trigger a panic when an invalid operation is performed and print a detailed stack trace of the location where the error occurred. This is extremely useful in tracking down logical errors that would otherwise lead to undefined behavior in C/C++.

Another useful debugging tool to catch logical bugs is the use of assertions, to verify that certain conditions always hold true at specific points of the code. These conditions are true if the logic of the code is correct, so an assertion failure indicates exactly where the logic diverged from the expected one. Rust provides built-in support for assertions using the family of `assert!` macros, which are also checked when compiling in release mode (as opposed to the `debug_assert!` macros that are only checked in debug mode), so even in optimized builds these checks can help in tracking down bugs, with minor impact to code performance.

4.5.4 Runtime logging difference

While optimizing the Rust libdeflate rewrite, it was quite common to introduce differences in the decompressed output compared to the pre-optimization version. Finding the root cause of these differences can be difficult, as they often occur in the middle of medium-large files, and comparing executions using the debugger or by printing to console is not feasible due to the extremely large number of operations performed.

To ease the comparison between two executions, a Rust tool called `runtime-diff` was developed, which can automatically compare two different logs to find the first difference between them. The instrumenting part consists of a library that exposes two macros:

- `runtime_check`: a `println`-like macro that requires that the printed string is the same between two executions, otherwise the comparison fails and returns the difference.
- `breadcrumb`: another `println`-like macro that is useful in tracking data specific to the instrumented executable. It is not compared against the other tool's output, and it is only printed in case of mismatch of a `runtime_check` to help in tracking the context of the error.

Then a separate tool called `runtime-diff` is used to compare the output of the two instrumented executions, by defining a file `runtime-diff.run` that specifies both the build command and the test commands to compare. A flag `--max-breadcrumbs` specifies the maximum number of breadcrumbs to print before the first difference, to avoid bloating the tool output.

```
build:
    cargo build --release --bin test-binary
    cargo build --release --bin test-binary2

test:
    new: ./target/release/test-binary -k 12
    new2: ./target/release/test-binary -k 12
    old: ./target/release/test-binary2 -k 12
```

Listing 1: An example of a `runtime-diff.run` file, that specifies how to build and run two different versions of the same project for comparison.

4.5.5 Memory sanitizers

For the most efficiency-critical parts of GGCAT, unsafe Rust code is used to bypass some of Rust's safety checks and achieve maximum performance. This bypass of the Rust checks makes bug tracking more difficult, as memory-related issues no longer trigger panics but can just lead to crashes, silent data corruption or undefined behavior, without any hint on where the problem is located. To track down memory-related bugs, memory sanitizers are very useful, as they instrument the code by adding checks for reading and writing to invalid memory locations. Nightly Rust has built-in support for memory sanitizers, that can be enabled by building the code with the following command:

```
RUSTFLAGS=-Zsanitizer=address cargo build -Zbuild-std --target x86_64-
unknown-linux-gnu
```

Acknowledgments

I would first like to thank my advisor, Romeo Rizzi, with whom I have collaborated for many years and whom I consider both a mentor and a friend. I am also deeply grateful to Alexandru I. Tomescu, who suggested the idea behind this tool, helped turn it into reality, and supported me through some of the most difficult moments of this research.

I am grateful to the whole Helsinki Bioinformatics group for the discussions, feedback, and supportive working environment, but also for the many happy moments we shared and for the friendships I found there.

I also want to thank my friend Alberto, with whom I shared many days in the office, and my Italian group of friends for their companionship throughout these years.

Finally, I want to thank my family for supporting me throughout this journey and always encouraging me: my mother Daniela, my father Carlo, my cousin Giovanni, and my aunt Monica.

Bibliography

[1] E. Drezen *et al.*, "GATB: Genome Assembly & Analysis Tool Box," *Bioinformatics*, vol. 30, no. 20, pp. 2959–2961, 2014, doi: 10.1093/bioinformatics/btu406.

[2] J. Ruan and H. Li, "Fast and accurate long-read assembly with wtdbg2," *Nature Methods*, vol. 17, no. 2, pp. 155–158, 2020.

[3] A. Rahman and P. Medvedev, "Representation of k-mer Sets Using Spectrum-Preserving String Sets," in *Research in Computational Molecular Biology - 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10-13, 2020, Proceedings*, R. Schwartz, Ed., in Lecture Notes in Computer Science, vol. 12074. Springer, 2020, pp. 152–168. doi: 10.1007/978-3-030-45257-5_10.

[4] K. Břinda, M. Baym, and G. Kucherov, "Simplitigs as an efficient and scalable representation of de Bruijn graphs," *Genome Biology*, vol. 22, no. 1, p. 96, doi: 10.1186/s13059-021-02297-z.

[5] J. Khan, M. Kokot, S. Deorowicz, and R. Patro, "Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2," *Genome Biology*, vol. 23, no. 1, p. 190, 2022, doi: 10.1186/s13059-022-02743-6.

[6] S. S. Schmidt and J. N. Alanko, "Eulertigs: Minimum Plain Text Representation of k-mer Sets Without Repetitions in Linear Time," in *22nd International Workshop on Algorithms in Bioinformatics, WABI 2022, September 5-7, 2022, Potsdam, Germany*, C. Boucher and S. Rahmann, Eds., in LIPIcs, vol. 242. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 2:1–2:21. doi: 10.4230/LIPIcs.WABI.2022.2.

[7] S. Schmidt, S. Khan, J. N. Alanko, G. E. Pibiri, and A. I. Tomescu, "Matchtigs: minimum plain text representation of k-mer sets," *Genome Biology*, vol. 24, no. 1, p. 136, 2023, doi: 10.1186/s13059-023-02968-z.

[8] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, "De novo assembly and genotyping of variants using colored de Bruijn graphs," *Nature Genetics*, vol. 44, no. 2, pp. 226–232, 2012.

[9] T. Zekic, G. Holley, and J. Stoye, "Pan-genome storage and analysis techniques," *Comparative Genomics*, pp. 29–53, 2018.

[10] N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter, "Near-optimal probabilistic RNA-seq quantification," *Nature Biotechnology*, vol. 34, no. 5, pp. 525–527, 2016.

[11] N. Luhmann, G. Holley, and M. Achtman, "BlastFrost: fast querying of 100,000s of bacterial genomes in Bifrost graphs," *Genome Biology*, vol. 22, no. 1, pp. 1–15, 2021.

[12] R. Wittler, "Alignment-and reference-free phylogenomics with colored de Bruijn graphs," *Algorithms for Molecular Biology*, vol. 15, no. 1, pp. 1–12, 2020.

[13] K. Dufault-Thompson and X. Jiang, "Applications of de Bruijn graphs in microbiome research," *iMeta*, vol. 1, no. 1, p. e4, 2022.

[14] J. N. Alanko, J. Vuohtoniemi, T. Maklin, and S. J. Puglisi, "Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes," *bioRxiv*, 2023, doi: 10.1101/2023.02.24.529942.

[15] J. Fan, J. Khan, N. P. Singh, G. E. Pibiri, and R. Patro, "Fulgor: a fast and compact k-mer index for large-scale matching and color queries," *Algorithms for Molecular Biology*, vol. 19, no. 1, p. 3, 2024, doi: 10.1186/s13015-024-00251-9.

[16] M. Karasikov *et al.*, "Efficient and accurate search in petabase-scale sequence repositories," *Nature*, vol. 647, no. 8091, pp. 1036–1044, Nov. 2025, doi: 10.1038/s41586-025-09603-w.

[17] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de Bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016, doi: 10.1093/bioinformatics/btw279.

[18] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015, doi: 10.1093/bioinformatics/btv022.

[19] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[20] A. Cracco and A. I. Tomescu, "Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT," *Genome Research*, vol. 33, no. 7, pp. 1198–1207, July 2023, doi: 10.1101/gr.277615.122.

[21] N. De Bruijn, "A Combinatorial Problem, Nederl. Akad. Wetensch. Proc., 1946, v. 49," *Indagationes Math*, vol. 8, pp. 461–467, 1946.

[22] I. J. Good, "Normal recurring decimals," *Journal of the London Mathematical Society*, vol. 1, no. 3, pp. 167–169, 1946.

[23] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de Bruijn graphs to genome assembly," *Nature Biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[24] G. Robertson *et al.*, "De novo assembly and analysis of RNA-seq data," *Nature Methods*, vol. 7, no. 11, pp. 909–912, 2010.

[25] A. Limasset, J.-F. Flot, and P. Peterlongo, "Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs," *Bioinformatics*, vol. 36, no. 5, pp. 1374–1381, 2020.

[26] L. Salmela and E. Rivals, "LoRDEC: accurate and efficient long read error correction," *Bioinformatics*, vol. 30, no. 24, pp. 3506–3514, 2014.

[27] B. Liu, H. Guo, M. Brudno, and Y. Wang, "deBGA: read alignment with de Bruijn graph-based seed and extension," *Bioinformatics*, vol. 32, no. 21, pp. 3224–3232, 2016.

[28] G. Benoit *et al.*, "Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph," *BMC Bioinformatics*, vol. 16, no. 1, pp. 1–14, 2015.

[29] D. L. Cameron *et al.*, "GRIDSS: sensitive and specific genomic rearrangement detection using positional de Bruijn graph assembly," *Genome Research*, vol. 27, no. 12, pp. 2050–2060, 2017.

[30] M. Kokot, M. Długosz, and S. Deorowicz, "KMC 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017, doi: 10.1093/bioinformatics/btx304.

[31] H. Mohamadi, J. Chu, B. P. Vandervalk, and I. Birol, "ntHash: recursive nucleotide hashing," *Bioinformatics*, vol. 32, no. 22, pp. 3492–3494, 2016.

[32] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[33] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 22, 2013, doi: 10.1186/1748-7188-8-22.

[34] J. Khan and R. Patro, "Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections," *Bioinformatics*, vol. 37, no. Supplement_1, pp. i177–i186, 2021.

[35] I. Minkin and P. Medvedev, "TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes," *Bioinformatics*, vol. 33, no. 24, pp. 4024–4028, 2017, doi: 10.1093/bioinformatics/btw609.

[36] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo, "Fast and scalable minimal perfect hashing for massive key sets," *arXiv preprint arXiv:1702.03154*, 2017.

[37] J. Khan, L. Dhulipala, P. Pandey, and R. Patro, "Fast and Scalable Parallel External-Memory Construction of Colored Compacted de Bruijn Graphs with Cuttlefish 3," *bioRxiv*, 2025, doi: 10.1101/2025.02.02.636161.

[38] G. Holley and P. Melsted, "Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs," *Genome Biology*, vol. 21, no. 1, p. 249, 2020, doi: 10.1186/s13059-020-02135-8.

[39] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[40] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2010.

[41] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Software: Practice and Experience*, vol. 46, no. 5, pp. 709–719, 2016.

[42] J. M. Zook *et al.*, "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *Scientific Data*, vol. 3, p. 160025, 2016, doi: 10.1038/sdata.2016.25.

[43] J. Mas-Lloret *et al.*, "Gut microbiome diversity detected by high-coverage 16S and shotgun sequencing of paired stool and colon sample," *Scientific data*, vol. 7, no. 1, pp. 1–13, 2020.

[44] The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, Sept. 2015.

[45] Z. Zhou, N.-F. Alikhan, K. Mohamed, Y. Fan, and M. Achtman, "The EnteroBase user's guide, with case studies on Salmonella transmissions, Yersinia pestis phylogeny, and Escherichia core genomic diversity," *Genome Research*, vol. 30, pp. 138–152, 2020, doi: 10.1101/gr.251678.119.

[46] G. A. Blackwell *et al.*, "Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences," *PLoS biology*, vol. 19, no. 11, p. e3001421, 2021.

[47] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3." [Online]. Available: https://www.rfc-editor.org/rfc/rfc1951

[48] E. Biggers, "libdeflate: Heavily optimized library for DEFLATE/zlib/gzip compression and decompression." Accessed: Jan. 15, 2025. [Online]. Available: https://github.com/ebiggers/libdeflate

[49] G. E. Pibiri and R. Venturini, "Techniques for Inverted Index Compression," *ACM Computing Surveys*, vol. 53, no. 6, pp. 1–36, Nov. 2021, doi: 10.1145/3415148.

[50] Google LLC, "Encoding — Protocol Buffers Documentation." 2024.

[51] S. Schmidt, S. Khan, J. N. Alanko, G. E. Pibiri, and A. I. Tomescu, "matchtigs: Minimum plain text representation of k-mer sets." GitHub repository, 2023.

[52] S. Sunagawa *et al.*, "Structure and function of the global ocean microbiome," *Science*, vol. 348, no. 6237, p. 1261359, 2015, doi: 10.1126/science.1261359.

[53] M. Hunt *et al.*, "AllTheBacteria - all bacterial genomes assembled, available and searchable," *bioRxiv*, 2025, doi: 10.1101/2024.03.08.584059.

[54] Rust Project Developers, "The Cargo Book." 2025.

[55] Intel Corporation, "Intel® VTune™ Profiler." 2024.