Cdbgtricks: Strategies to update a compacted de Bruijn graph

Khodor Hannoush[1⋆], Camille Marchet[2], and Pierre Peterlongo[1]

[1] Univ. Rennes, Inria, CNRS, IRISA - UMR 6074, Rennes, F-35000 France
khodor.hannoush@inria.fr
pierre.peterlongo@inria.fr
[2] Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
camille.marchet@univ-lille.fr

Abstract. We propose `Cdbgtricks`, a new method for updating a compacted de Bruijn graph when adding novel sequences, such as full genomes. Our method indexes the graph, enabling to identify in constant time the location (unitig and offset) of any k-mer. The update operation that we propose also updates the index. Our results show that `Cdbgtricks` is faster than `Bifrost` and `GGCAT`. We benefit from the index of the graph to provide new functionalities, such as reporting the subgraph that shares a desired percentage of k-mers with a query sequence with the ability to query a set of reads. The open-source `Cdbgtricks` software is available at `https://github.com/khodor14/Cdbgtricks`.

Keywords: compacted de Bruijn graphs, k-mers, data structures, genomics, indexing

1 Introduction

The de Bruijn graph is one of the fundamental data structures that play crucial roles in computational biology. It is tremendously used in various applications, including but not limited to genome assembly [20,23], read error correction [16,13], read alignment [17,4] and read abundance queries [18]. The de Bruijn graph is a data structure in which the nodes represent the distinct substrings of length k of a set of strings, called k-mers, and the edges link the nodes that share an overlap of $(k-1)$. In a compacted de Bruijn graph, nodes along each maximal non-branching path of the de Bruijn graph are compacted into a single node representing a sequence of length $\geq k$, called a unitig. The de Bruijn graph can be constructed from a set of assembled genomes or a set of reads.

Several methods have been proposed in the literature for compacted de Bruijn graph construction, including `PanTools` [24], `Bcalm2` [6], `TwoPaCo` [21], `deGSM` [9], `Bifrost` [11], `Cuttlefish2` [14], `GGCAT` [7], and `FDBG` [8]. As genomic databases grow, there is a demand for dynamic de Bruijn graph data structures supporting sequence additions. `BufBoss` [1], `DynamicBoss` [2], and `FDBG` [8] support the addition and the deletion of nodes and edges in de Bruijn graphs.

Nevertheless, the performance of `BufBoss`, `DynamicBoss` and `FDBG` in sequence addition falls short compared to `Bifrost`. We refer the reader to the results of the `BufBoss` paper [1]. Despite being faster in adding new sequences, `Bifrost` builds a new graph from the sequences to be added, using its construction method, and then it merges the two graphs. The reliance on constructing a graph from the new sequences introduces computational overhead, potentially limiting the scalability and efficiency

⋆ Corresponding author

Khodor Hannoush, Camille Marchet, Pierre Peterlongo: Cdbgtricks: Strategies to update a compacted de Bruijn graph, pp. 86–103.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5 © Czech Technical University in Prague, Czech Republic

of `Bifrost`, particularly when dealing with large graphs, highlighting the need for a more efficient updating mechanism.

Recent advances have focused on developing memory- and time-efficient indexing structures for k-mers. Some notable methods, such as `BLight` [19], `SSHash` [22], `GGCAT` [7], and `Pufferfish` [3], are recognized for their efficiency in this regard. However, it is important to note that these methods are static, meaning that they cannot easily incorporate new data. This limitation becomes especially problematic with large datasets when construction time must be paid for every addition of new sequences.

Indexing methods are used as well for read queries, read mapping, and read alignment on de Bruijn graph. Read mapping against the de Bruijn graph has been studied extensively. `Bifrost` [11], `GGCAT` [7] and `SSHash` [22] proposed read query methods. The limitation of `Bifrost` and `SSHash` is the need to compare any queried k-mer with all k-mers having the same minimizer, the smallest substring of length m according to some order where $0 < m < k$. Although it is an efficient design to reduce memory usage, it slows down when negative queried k-mers.

In this paper, we propose "`Cdbgtricks`", a novel strategy, and a method to add sequences to an existing uncolored compacted de Bruijn graph. Our method takes advantage of `kmtricks` [15] that finds in a fast way what k-mers are to be added to the graph, and our indexing strategy enables us to determine the part of the graph to be modified while computing the unitigs from these k-mers. The index of `Cdbgtricks` is also able to report exact matches between query reads and the graph. We compared `Cdbgtricks` against `Bifrost` and `GGCAT`. Despite `GGCAT` lacking the update feature on the compacted de Bruijn graph, we included it in our comparison due to its potential efficiency in graph construction, which may outperform an update approach. `Cdbgtricks` is up to 2x faster than `Bifrost` on updating a compacted de Bruijn graph on 100 human genomes datasets, and it shows the competitiveness potential against `GGCAT` on larger human genome datasets for which `GGCAT` may not scale due to the high disk requirement. `Cdbgtricks` is up to 3x faster than `Bifrost` and `GGCAT` on updating a compacted de Bruijn graph on a large *E. coli* genomes dataset.

2 Methods

2.1 Preliminary definitions

A string s is a sequence of characters drawn from an alphabet Σ. In this paper, we use the DNA alphabet $\Sigma = \{A, C, G, T\}$ where every character has its complement in Σ. The complement pairs of Σ are (A, T) and (C, G). The reverse complement $\bar{s}$ of s is found by reversing s and then complementing the characters. The canonical form of a string s is the lexicographically smallest string between s and its reverse complement $\bar{s}$. We denote by $|s|$ the length of s. $s[i]$ denotes the i^{th} symbol of s, starting from zero ($s[0]$ represents the first character of s and $s[|s|-1]$ is its last character). Denote by $\mathrm{s}(i, j)$ the substring of s starting at i and ending at $j-1$. A k-mer is a string of length k. For a given k, in our specific context, we denote by $\mathit{pref}(s) = s(0, k-1)$ the $(k-1)$-prefix of a string s, and by $\mathit{suff}(s) = s(|s|-k+1, |s|)$ the $(k-1)$-suffix of s.

Definition 1. de Bruijn Graph: The de Bruijn graph constructed from a set of sequences S is a directed graph $G = (V, E)$ where V represents the set of distinct k-mers of S. When input sequences S are made of raw sequencing data, before constructing the graph, the k-mers of S are counted, and those whose abundance is smaller than a

fixed threshold are considered to contain sequencing errors, thus they are discarded. Note that a node u represents a k-mer x and its reverse complement $\bar{x}$. Two nodes v and w are connected by an edge $e \in E$ from v to w if one of the following holds:

1. $\mathit{suff}(v) = \overline{\mathit{suff}(w)}$
2. $\mathit{pref}(v) = \overline{\mathit{pref}(w)}$
3. $\mathit{suff}(v) = \mathit{pref}(w)$

In any of these three cases, v is an in-neighbor of w, and w is an out-neighbor of v. It is worth mentioning that within the scope of this paper, the edges are not stored explicitly; rather, they are deduced from the nodes.

Definition 2. Path: A path of a dBG is an ordered set of nodes where every two consecutive nodes are connected by an edge.

Definition 3. Unitig: A maximal non-branching path is a path $p = \{f, v_1, v_2, ..., v_{|p|-2}, l\}$ where every v_i has only one in-neighbor and one out-neighbor and f and l do not have this property. A maximal non branching path can be compacted to form a unitig u. The compaction of two nodes v and w can be achieved as follows:

1. If $\mathit{suff}(v) = \mathit{pref}(w)$ then $\mathit{compaction}(v, w) = v \odot w[|w| - 1]$
2. If $\mathit{suff}(v) = \overline{\mathit{suff}(w)}$ then $\mathit{compaction}(v, w) = v \odot \overline{w[0]}$
3. If $\mathit{pref}(v) = \overline{\mathit{pref}(w)}$ then $\mathit{compaction}(v, w) = \overline{w[0]} \odot v$

Without loss of generality, we suppose in what follows that two nodes v and w are in forward-forward direction i.e., $\mathit{suff}(v) = \mathit{pref}(w)$. It should be noted that the rest of the cases remain valid within the scope of this definition. In what follows we will not emphasize if a k-mer x is canonical or not.

Definition 4. Compacted de Bruijn Graph: Replacing the maximal non-branching paths by their unitigs provides a compacted form of the de Bruijn graph. An illustration of a de Bruijn graph and its compacted version is shown in Figure 1.

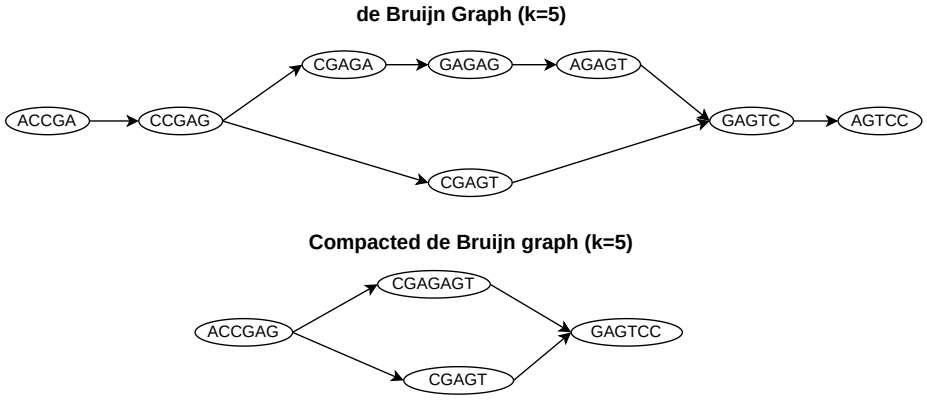

Figure 1. A de Bruijn graph and its compacted de Bruijn graph version.

Definition 5. Minimizer: A minimizer of a string s is a substring q of fixed length m where $m < |s|$ and q is the smallest m-mer of s with respect to some order. In this paper, the order is defined on the basis of a hash function.

Definition 6. Minimal perfect hash function MPHF: Given a set of keys K, a minimal perfect hash function (MPHF) is a function that bijectively maps the elements of K to the elements of the set $I = \{i | 0 \leq i < |K|\}$.

2.2 Overview of the Algorithm

`Cdbgtricks` enables to add a set of new sequences S to a compacted de Bruijn graph G. We denote by K_S the set of k-mers in S and by K_G the set of k-mers in G. The set of k-mers in K_S but not in K_G have to be added to G. We call N this set $K_S \backslash K_G$. To efficiently determine N we use the `kmtricks` [15] tool. To help understand the proposed algorithm, we first describe the process when adding k-mers from N one after another in the compacted de Bruijn graph G. We show later (Section 2.4), how to avoid these $|N|$ individual additions.

When adding a k-mer x from N to the compacted de Bruijn graph G, we distinguish the following cases, as represented in Figure 2:

1. **Add x as a new unitig**. Neither $\mathit{pref}(x)$ nor $\mathit{suff}(x)$ appears in any unitig of G. In this case, the existing unitigs of G are not modified and x is added as a new single unitig in G (Fig 2.a).
2. **Right extension of a unitig**. If $\mathit{pref}(x)$ equals $\mathit{suff}(u)$ for a unitig u of G, and u has no out-neighbor, then u is extended with the last character of x ($u = u \odot x[k-1]$)(Fig 2.b).
3. **Left extension of a unitig**. If $\mathit{suff}(x)$ equals $\mathit{pref}(u)$ for a unitig u of G and u has no in-neighbor, then the first character of x is added to the left of u ($u = x[0] \odot u$) (Fig 2.c).
4. **Merge two unitigs**. If the addition of x leads to the right extension of a unitig u_1 **and** to the left extension of a unitig u_2, then, after the extensions, $\mathit{suff}(u_1) = \mathit{pref}(u_2)$. In this case the two unitigs u_1 and u_2 are merged into a unique unitig $u = u_1 \odot u_2(k, |u_2|)$ (Fig 2.d).
5. **Splitting a unitig**. If $\mathit{pref}(x)$ exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $\mathit{suff}(u_1) = \mathit{pref}(u_2) = \mathit{pref}(x)$ and x is added as a single unitig. Respectively, if $\mathit{suff}(x)$ exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $\mathit{suff}(u_1) = \mathit{pref}(u_2) = \mathit{suff}(x)$ and x is added as a single unitig (Fig 2.e).

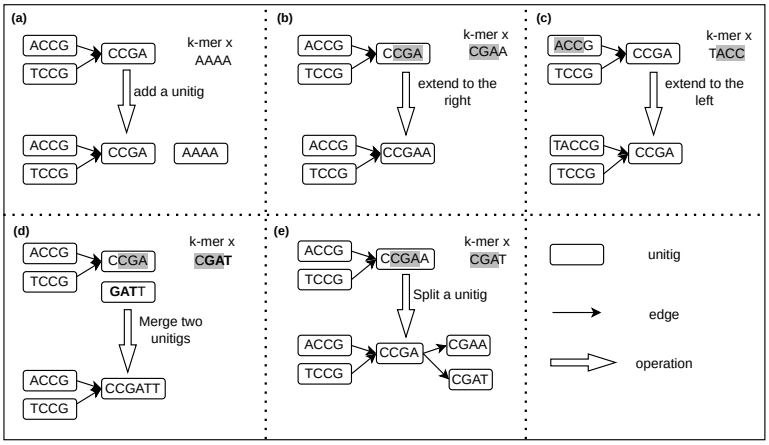

Figure 2. Possible operations when adding a k-mer to a compacted de Bruijn graph with $k = 4$. (a) Adding the k-mer as a new unitig. **(b)** Extending a unitig to the right. **(c)** Extending a unitig to the left. **(d)** Merging two unitigs. **(e)** Split a unitig into two unitigs. Gray and bold sequences represent overlap between the added k-mer and some unitigs of the graph.

These operations rely extensively on the pattern matching of $suff(x)$ and $pref(x)$ in the unitigs of G. In order to rapidly perform these operations, we propose to index the graph, as explained in the next section.

2.3 Indexing the graph

A core operation in `Cdbgtricks` consists in identifying if a $(k-1)$-mer occurs in any unitig of a compacted de Bruijn graph G, and, if this is the case, to determine the couple(s) (unitig id, offset) where it occurs.

This operation is performed twice for each k-mer x of N (for $pref(x)$ and $suff(x)$). Ideally it has to have a $O(1)$ time-complexity and to be fast in practice. This is a very common operation for which existing indexing solutions such as [19,22] are convenient. However, in the context of this work, the specificity is that, when adding sequences to the graph, the indexed data evolve as some unitigs G can be split, merged, extended, and some new unitigs can be added to G. Hence, those static methods are not adapted. We propose the following strategy to cope with this particular situation.

Indexing k-mers for querying $(k-1)$-mers Despite the fact that we query $(k-1)$-mers, we chose to index k-mers instead of $(k-1)$-mers. A $(k-1)$-mer may have up to eight occurrences in G because it can be the suffix of four possible k-mers and the prefix of four possible k-mers. Indexing from one to eight couples (unitig id, offset) per indexed element is not efficient as it requires a structure of undefined and variable size. This leads to heavy data-structures and cache-misses on construction and query times. To cope with this issue, we chose to index k-mers instead of $(k-1)$-mers. Indeed, each k-mer in a compacted de Bruijn graph of order k, occurs at exactly one couple (unitig id, offset).

Given this indexing scheme in which k-mers are indexed, when querying a $(k-1)$-mer x', the eight possible k-mers containing this $(k-1)$-mer (four k-mers in which x' is the prefix, and four k-mers in which x' is the suffix) are queried. If a match is found, the offset of the $(k-1)$-mer is deduced depending on the case (either x' is the prefix or the suffix of a queried k-mer for which a match is found).

As a matter of fact, we only index each k-mer in its canonical form. Then, a queried k-mer is searched in its canonical form.

Partitioning the k-mers of the graph Conceptually, we could use any associative table such as a hash table for mapping each k-mer of G to its couple (unitig id, offset). However, this would require explicitly storing the k-mers which is a waste of space as k-mers are already explicitly existing in unitigs. Alternatively, we use an MPHF f from the k-mers of the graph. Doing so, we need only to store the position of each indexed k-mer. Formally the position of a k-mer x is defined by $p_x =< u_{id}, u_{off}, orientation >$, where u_{id} is the identifier of the unitig u, $0 \leq u_{off} \leq |u| - k$ is the offset of x in u, and $orientation$ is a boolean variable that is true if x is in its canonical form in u, else it is false. The positions of the k-mers in the graph are stored in a vector V. Given a k-mer x, its position is $p_x = V[j]$ where $j = f(canonical(x))$. There are two observations to be made here:

- At query time, f can give valid hash values for alien k-mers, which are k-mers that are not present in the graph. To handle this, we compare the queried

k-mer to the actual k-mer in the graph, whose position is retrieved thanks to $V[f(canonical(x))]$.
- Adding new k-mers requires to recompute f.

This last point is problematic, since for every addition operation, f must be recomputed, which is a linear-time operation in terms of the number of indexed k-mers. To resolve this, we divide the set of k-mers in the graph into multiple subsets called "buckets". Each subset is indexed using its own MPHF. The key idea being that while adding sequences to a graph, only a subset of the buckets are modified, and so only a subset of the MPHFs have to be recomputed. At query time, the bucket of the queried k-mer x is retrieved, and the corresponding MPHF provides the position of x in the graph.

Formally, we define $\{b_0, b_1, ..., b_{n-1}\}$ buckets. For each of these buckets b_i, an MPHF f_i is computed on its k-mers. MPHFs are computed using `PHOBIC` [10] as it provides the fastest lookup compared to the state of the art tools that compute MPHFs.

The k-mers in the graph are separated into buckets based on their minimizers. The k-mers sharing the same minimizer cannot be distributed into different buckets. However, this strategy may result in a well-known problem of non-uniform distribution of the k-mers in the buckets [5]. Some buckets could be orders of magnitudes larger than some others. Also, the small buckets are problematic for the construction of an MPHF using `PHOBIC`, as higher number of $bits/\text{k}{-}mer$ is required for small buckets (see Figure 3).

All in all, we propose a strategy so that all the batches contain a minimum number of k-mers.

- The number of k-mers sharing the same minimizer should be at least equal to a parameter ρ for creating a bucket. Note that the size of a bucket does not have an upper-bound.
- For the remaining k-mers, we process them by groups of k-mers where the k-mers within a group share the same minimizer. From these groups, we create what so-called "super-buckets" which are buckets containing k-mers that share different minimizers. We start with an empty super-bucket S_0 to which we add the groups of k-mers one by one. Once the number of k-mers added to S_0 exceeds $\gamma \times \rho$ k-mers (with γ a user defined multiplicative factor), we create a new super-bucket. We keep creating and filling super-buckets until all groups of k-mers are processed. The rationale behind this strategy is to achieve a balanced distribution of k-mers on the super-buckets. It is important to note that the size of a super-bucket has an upper bound, which we address in section 2.5.

Finally the data-structure is composed of the following components, represented in Figure 4:

1. A hash table T that maps each minimizer to its bucket identifier. The hash table will be used to identify the bucket that may contain a given k-mer x. The identifier of the bucket is then $b_i = T[minimizer(x)]$ where $minimizer(x)$ is the minimizer of x.
2. A hash table F that maps each bucket identifier to its MPHFs. Hence, $F[b_i]$ is the MPHF computed from the set of k-mers in bucket b_i.

3. A hash table U that maps the identifier of each unitig of the graph to its sequence. Hence, $U[u_i]$ is the unitig sequence whose identifier is u_i.

4. The positions of the k-mers in the graph are stored in a 2-D vector P. $P[b_i]$ is the vector of positions for the k-mers in bucket b_i. $P[b_i][F[b_i](x)]$ is the tuple position $< u_{id}, u_{off}, orientation >$ for the k-mer x in the bucket b_i.

Overall given the position of a k-mer x, x can be retrieved by retrieving the unitig $u = U[u_{id}]$ from the hash table U, hence $x = u(u_{off}, u_{off} + k)$ (Figure 4.c). Note that if the minimizer of x is not present in T, then x does not belong to the graph (Figure 4.b).

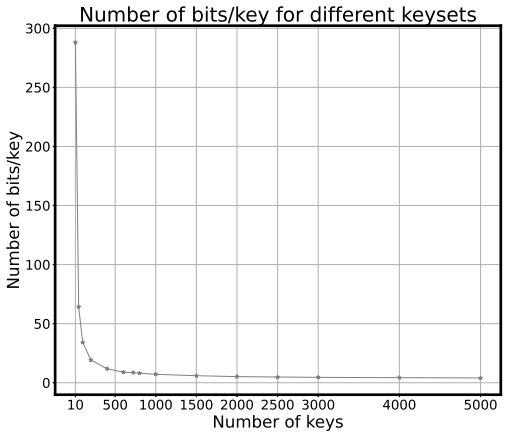

Figure 3. The number of bits/key required for building a MPHF with `PHOBIC`. MPHFs from different sets of keys of sizes ranging from 10 to 5000 random 64-bit keys were computed by `PHOBIC`, and bits/key were then measured.

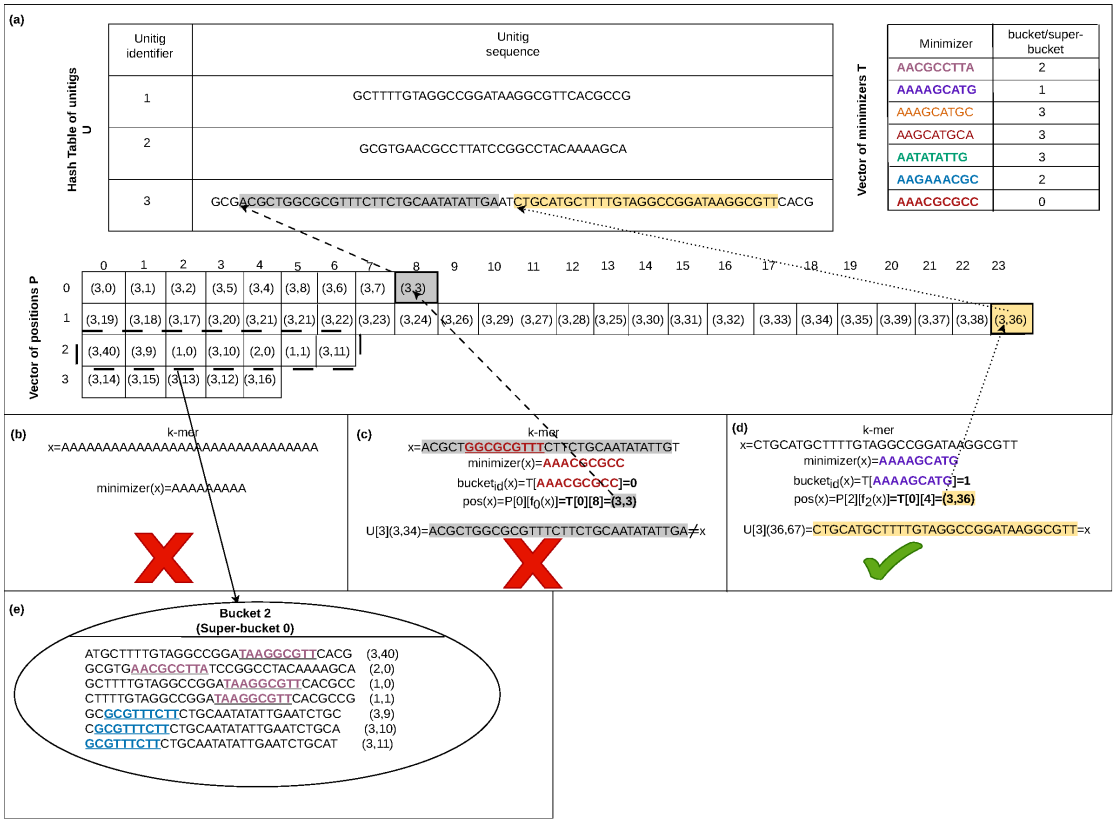

Figure 4. Overview of the data structure. **(a)** the hash table U of the unitigs of a compacted de Bruijn graph with $k = 31$; the hash table T of minimizers that maps a minimizer to its corresponding bucket or super-bucket; and the vector P of positions of the k-mers as a tuple (u_{id}, u_{off}) where u_{id} is the identifier of the unitig in which the k-mer occur at offset u_{off}. Note that the values on top of the vector P represent the hash value of the k-mers computed by the MPHF of their bucket or super-bucket, and the values to the left of P represent the bucket or super-bucket identifier. **(b)** querying an absent k-mer whose minimizer is not in the index. **(c)** querying an absent k-mer whose minimizer is in the index. **(d)** querying a present k-mer. **(e)** the dashed box is converted back to a super-bucket of the k-mers.

2.4 Computing the future unitigs and updating the graph

Recall that N denotes the set of k-mers to be added to a compacted de Bruijn graph G. In Section 2.2 we proposed an overview of algorithms in which k-mers from N are added one after another to G. In practice, for performance reasons, we first compact k-mers from N into what we call "funitigs" (for *future unitigs*).

The funitigs are not simply the unitigs of N as any $(k-1)$-mer of those funitigs that is already in G must be either a prefix or a suffix of a funitig. Doing so, the funitigs are not split latter when added to the graph. The details about the funitig construction are given in Algorithm 3 of supplementary materials.

Once the funitigs are constructed, each of them is added to the graph one after the other. The rules described section 2.2 for adding a k-mer to G exactly apply for adding a funitig to G. The `Cdbgtricks` tool exactly implements those rules, that we do not recall here.

2.5 Updating the index

`Cdbgtricks` enables to update the index of a compacted de Bruijn graph, after the addition of sequences. The updated index can serve for any future update on the graph and for k-mer queries against the graph.

While adding funitigs to the graph, we remember the identifiers of the modified unitigs and of the modified buckets. After all funitigs are added to the graph, the index of the corresponding unitigs and buckets are updated. The update of the index of the graph is divided into three stages:

1. Splitting a unitig or a joining two unitigs or a unitig with a funitig result in changing unitig identifier(s) and the offsets of some k-mers. When we split a unitig u into two unitigs u_1 and u_2, u_1 get the identifier of u, u_2 gets a new identifier and the offsets of its k-mers get recomputed. When we merge a funitig with one or two unitigs, the resultant sequence gets the identifier of one of these unitigs, and the offsets its k-mers get recomputed.
2. The addition of k-mers to super-buckets may lead to doubling their maximum size ($\gamma \times \rho$). In this case, for each concerned super-bucket, it is divided into two new super-buckets.
3. Recompute the MPHFs of the buckets and super-buckets to which new k-mers were added.

When dividing a super-bucket into two super-buckets (case 2), the objective is to balance the size of the two created super-buckets. To address case 2, a straightforward greedy strategy is employed to split the super-bucket into two smaller ones. We propose a simple greedy algorithm (see Algorithm 1) for performing this task.

Algorithm 1 Divide a super-bucket into two super-buckets

Require: A super-bucket of k-mers B
Ensure: Two balanced super-buckets B_1 and B_2
  Initialize four empty sets, B_1, B_2, M_1, M_2
  for each k-mer $x \in B$ **do**
    $m \leftarrow minimizer(x)$
    if $m \in M_1$ **then**
      $B_1 \leftarrow B_1 \cup \{x\}$
    else if $m \in M_2$ **then**
      $B_2 \leftarrow B_2 \cup \{x\}$
    else if $|B_1| < |B_2|$ **then**
      $B_1 \leftarrow B_1 \cup \{x\}$
      $M_1 \leftarrow M_1 \cup \{m\}$
    else
      $B_2 \leftarrow B_2 \cup \{x\}$
      $M_2 \leftarrow M_2 \cup \{m\}$

2.6 Read querying

A compacted de Bruijn graph constructed using `Cdbgtricks` supports sequence queries. In practice, while querying a sequence s on a graph G, `Cdbgtricks` determines if at least α% of the k-mers of s are in the graph (with α a user-defined parameter). If this is the case, `Cdbgtricks` indicates the uni-MEMs, as defined in `deBGA` [17]. Each uni-MEM is a tuple $< u_{id}, u_{start}, u_{end}, s_{start}, s_{end} >$ where u_{start} and u_{end} are the start and end positions of mapping on the unitig whose

identifier is u_{id}, and s_{start} and s_{end} are the start and end positions of mapping k-mers of the queried sequence s. In other words, the k-mers whose offsets in the read are between r_{start} and r_{end} are found in the unitig u_{id} between u_{start} and u_{end}. A uni-MEM is found through the extension of the first common k-mer between the read and a unitig. The extension ends in one of the following cases:

1. A mismatch is encountered.
2. Either the end of the read or the end of the unitig is encountered.

3 Results

All presented results are reproducible using command lines and versions of tested tools, that are given in this repository `https://github.com/khodor14/Cdbgtricks_experiments`. The executions were performed on the GenOuest platform on a node with 4×8 cores Xeon E5-2660 2,20 GHz with 128 GB of memory.

3.1 Genome datasets

We tested `Cdbgtricks` in two frameworks, corresponding to two input datasets of distinct size and complexity. The first one, called "*human*" is composed of 100 assembled human genomes that were used in the `GGCAT` experiments [7]. These genomes are available on zenodo (`10.5281/zenodo.7506049,10.5281/zenodo.7506425`). The second set, called "*coli*" is composed of 7055 *E. coli* genomes downloaded from NCBI (`https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/030/`).

While `Cdbgtricks` is capable of initially constructing a compacted de Bruijn graph from scratch, it is worth noting that there are faster alternatives for creating the initial graph. As such, for each dataset, we created an initial graph in fasta format from one genome (chosen as the first in the alphabetic order of the file names) using `Bifrost`. Once created `Cdbgtricks` can be used for indexing this initial graph. Subsequently, for each dataset, we added one by one the remaining genomes.

3.2 Used Parameters

The used parameters are the same for the two datasets. In all experiments we used $k = 31$ and minimizers of size $m = 11$. During the update experiments the tools were executed using 32 threads, while during the query experiments the tools were executed using a single thread.

For `Cdbgtricks`, the parameters controlling the bucket size were set to default. The bucket lower bound size is $\rho = 5000$ and a the super-bucket multiplicative factor γ is set to 4. This setting of parameters means that the size of a super-bucket is approximately 20000 k-mers, and once it reaches 40000 k-mers, it gets divided into two super-buckets. The values of the parameters were chosen to ensure satisfactory results that will be shown in the subsequent sections.

3.3 Percentage of modified buckets

As explained Section 2.3, one of the key ideas in `Cdbgtricks` is to distribute indexed k-mers into multiple buckets, each bucket being indexed with its own MPHF. Doing

so, we expect that, while adding k-mers from novel sequences, k-mers are added to only a fraction of the buckets, and then only, a fraction of the MPHFs have to be recomputed. More precisely, we expect that the percentage of modified buckets, i.e. $100 \times \frac{\text{number of modified buckets}}{\text{total number of buckets}}$ decreases as the number of genomes in the graph increases. Note that here we do not differentiate buckets and super-buckets and we regroup these two notions in the term "buckets".

In this section we test this expectation on the *human* and *E. coli* datasets. The results about the percentage of modified buckets are shown Figure 5. Results show that, as expected, the percentage of modified buckets decreases with respect to the number of genomes. The shaded cluster of points for the *E. coli* dataset shows that in the majority of cases, the percentage of modified buckets and super-buckets is less than 20%. More specifically, results with, say, more than 5000 *E. coli* genomes show that, except for some outliers, less than 10% of the buckets are modified when adding a new genome.

These results validate the chosen default parameters, and they confirm the expectation that lesser and lesser buckets are modified while increasing the number of genomes of the same species in a compacted de Bruijn graph.

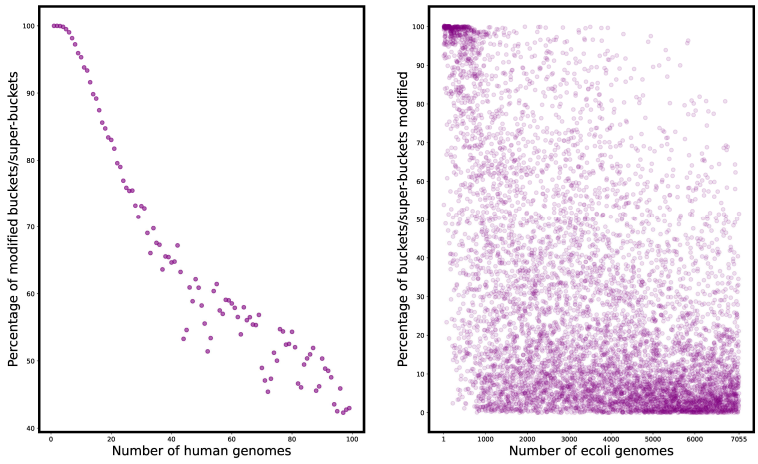

Figure 5. Percentage of modified buckets.

3.4 Scalability

One of the main objective of `Cdbgtricks` is the time performances when updating a compacted de Bruijn graph with new sequences. In this context, we compared the `Cdbgtricks` update time, with the update time obtained thanks to `Bifrost`, also able to update an already created compacted de Bruijn graph. Furthermore, although `GGCAT` does not provide graph updating capabilities, we included it in our comparison due to its efficiency. The memory and disk for the update with `Cdbgtricks` and `Bifrost` and for the construction with `GGCAT` are also reported.

Results for the *human* dataset are shown in Figure 6. Note that, on the *human* dataset, `GGCAT` reached a timeout we set at two days on more than 71 human genomes. Hence, only the results for the first 71 genomes were reported for `GGCAT`. Globally, the results on this dataset show that `Cdbgtricks` is at least 2x faster than `Bifrost` on

graphs composed of 50 genomes or more. Compared to `GGCAT`, `Cdbgtricks` is slower on this small number of genomes. Given that as the number of genomes in the graph increases, the `GGCAT` construction time naturally increases while the `Cdbgtricks` update time decreases, one can expect `Cdbgtricks` to be faster when dealing with more genomes than those tested here. However, given the observe `GGCAT` limitation after 71 genomes, we could not verify this fact in practice, at least for human genomes. The memory used by `Cdbgtricks` and `Bifrost` are slightly the same and are limited to a few dozen gigabytes. `GGCAT` uses much less memory, but needs up to order of magnitude more disk.

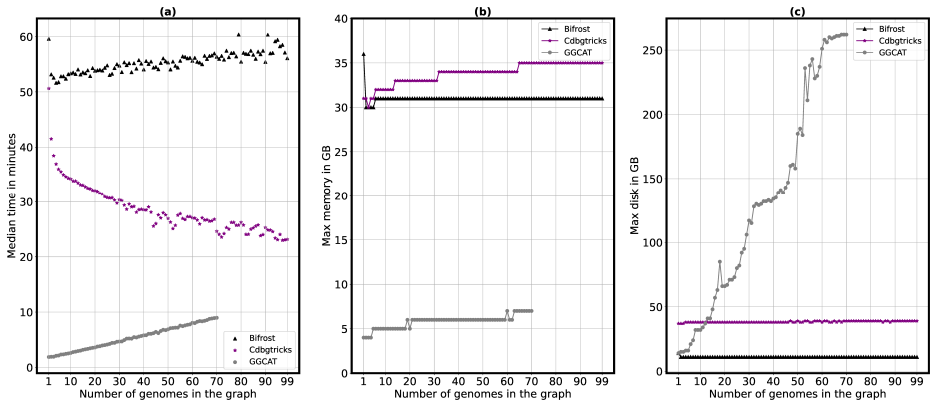

Figure 6. Results on human genomes dataset. Time (a), memory (b) and disk usage (c) are given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`.

Results for the *coli* dataset are shown in Figure 7. For the sake of clarity of presenting the results of the *E. coli*, we chose to report the median time over a window of 200 genomes. The detailed presentation of execution time is in supplementary materials. On this dataset, both `Bifrost` and `GGCAT` were faster than `Cdbgtricks` on graphs composed of less than a thousand genomes. However, in the vast majority of cases, when the number of genomes get higher than, say, 2000 genomes, `Cdbgtricks` is 2x to 3x faster than `GGCAT` and `Bifrost`. With `Cdbgtricks`, adding an *E. coli* genome to a compacted de Bruijn graph graph containing already few thousands genomes requires between 30 and 50 seconds. `Cdbgtricks` uses slightly the same amount of memory compared to `GGCAT` and roughly twice the amount of memory compared to `Bifrost`. `Cdbgtricks` uses up to 4x more disk compared to `Bifrost`, while it uses much less disk compared to `GGCAT`.

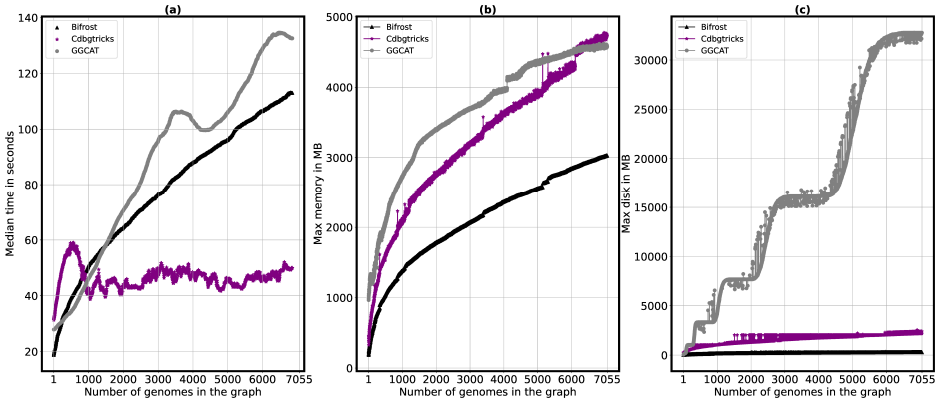

Figure 7. Results on *E. coli* genomes dataset. Time (a), memory (b) and disk usage (c) are given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`. The time is given as the median over a window of 200 consecutive points.

3.5 Results querying sequences

We propose some experiments for comparing the query performances of `Cdbgtricks` with those of `Bifrost`, `GGCAT` and `SSHash`. Note that these four tools do not offer the same query features. Although, these results must be considered as rough estimations showing the main tendencies.

Using `Bifrost`, we constructed a graph from 15,806 *E. coli* genomes, and a graph from 10 human genomes. Then we constructed an index for each graph using either `Cdbgtricks`, `Bifrost` or `SSHash`. We have differentiated between results obtained with positive queries (querying sequences present in the graph) and those obtained with negative queries (querying random sequences) with k-mers that are not in the two constructed graphs. The positive queries are a subset of unitigs from each graph. The negative queries are composed of one million random sequences of length between 500 and 1000 base pairs. The querying results are shown Table 1.

Table 1. Performances of sequence queries using a compacted de Bruijn graph for `Cdbgtricks`, `Bifrost`, `SSHash`, and `GGCAT`

Dataset	Query type	Tool	Memory (MB)	Disk (MB)	time (mm:ss)
E. coli	Negative	`Cdbgtricks`	4723	0	10:02
		`Bifrost`	4362	0	06:43
		`SSHash`	725	0	00:07
		`GGCAT`	560	3325	01:32
	Positive	`Cdbgtricks`	4724	0	02:15
		`Bifrost`	4362	0	01:43
		`SSHash`	725	0	01:10
		`GGCAT`	644	2978	01:26
human	Negative	`Cdbgtricks`	25520	0	12:25
		`Bifrost`	27376	0	11:37
		`SSHash`	6090	0	00:07
		`GGCAT`	615	6861	4:55
	Positive	`Cdbgtricks`	25520	0	04:23
		`Bifrost`	27376	0	06:37
		`SSHash`	6090	0	01:14
		`GGCAT`	746	7053	05:04

The results shows that `Cdbgtricks` and `Bifrost` obtained similar results in term of memory, while `Cdbgtricks` is slightly slower. While `GGCAT` is faster than `Bifrost` and `Cdbgtricks`, and it uses the smallest amount of memory in these query experiments, it uses few Gigabytes of disk. Despite `SSHash` being the fastest tool, it consumes an order of magnitude more memory than `GGCAT`. These results shows that `Cdbgtricks` offer queries in a reasonable amount of time, and the performance of `Cdbgtricks` is close to the performance of `Bifrost`.

4 Discussion and future work

In this paper, we presented `Cdbgtricks`, a novel method for updating a compacted de Bruijn graph when adding new sequences such as full genomes. `Cdbgtricks` also indexes the graphs, hence it enables to query sequences and detect the portions of the graph that share k-mers with the query. The dynamicity of the proposed index is achieved thanks to the distribution of k-mers into multiple buckets, each bucket being indexed using a minimal perfect hash function (MPHF). The addition of new k-mers affects only a fraction of the buckets, for which the MPHF has to be recomputed. In practice, when indexing a large number of genomes (dozens of human genomes or thousand of *E. coli* genomes) `Cdbgtricks` outperforms the computation time of state-of-the-art tools dedicated to the creation of the update of compacted de Bruijn graphs.

Of independent interest, exploiting `PTHash`, the `Cdbgtricks` indexing framework offers a theoretical way to bijectively associate each k-mer from a set composed of n distinct k-mers with a unique value in $[0, n[$. In that respect, despite the fact that some engineering work remains to be done to achieve this practical feature, our indexing strategy can be used as an MPHF for this kind of dataset. Furthermore, it will present two main additional advantages when compared to a classical MPHF:

- In essence, an MPHF is static. Adding an element to this kind of data structure requires one to recompute the entire MPHF from scratch to associate the $n + 1$ elements to a unique value in $[0, n + 1[$. As `Cdbgtricks` distributes the k-mer set over numerous "sub-MPHFs", adding an element requires only to recompute one of the sub-MPHFs. This offers a clear advantage when adding a few elements to large MPHFs, composed of, say, billions of elements.
- The MPHF definition does not impose that so-called "alien k-mers" (k-mers not belonging to the indexed set) are detected as aliens at query time. Actually, MPHFs that do not store the indexed set of elements (as this is the case for `BBhash` and `PTHash`), are not able to always discriminate an alien k-mer from an indexed one. In the context of this work, the presence of the indexed k-mers in the stored unitigs enables us to validate that a query k-mer actually belongs to the original set, and thus enables us to detect whether it is an alien k-mer or not.

A future research direction is to devise a smarter bucket clustering approach. One way could be to group the buckets whose minimizers appear in the same unitigs. Doing so, we could expect more data locality, limiting the cache-misses.

The compacted de Bruijn graph computed by `Cdbgtricks` is not colored. This means that the information is lost about the original genome(s) a k-mer belongs to. In recent years, significant attention has been given to the use of colored and compacted de Bruijn graphs in computational biology applications [12]. Hence, another research priorities for the future of this tool is to integrate the color information. This will

necessitate minor yet potentially expensive operations. To include the new colors associated with a set of new sequences S, the color information of all k-mers in S already present in the graph G will have to be updated, while those k-mers are not modified in the current uncolored `Cdbgtricks` version.

There exists no limitation for using `Cdbgtricks` for merging the information of two compacted de Bruijn graphs G_1 and G_2. We can simply consider the k-mers of, say, G_2 to be added into G_1, and apply the exact same algorithm as proposed here. Future work will include validation and scaling tests for this approach.

Finally, we believe that the number of common k-mers but also the number of splits and joins performed when adding a sequence to a graph could be used as metrics to estimate the distance between a sequence and a compacted de Bruijn graph, or even between two compacted de Bruijn graphs.

5 Funding

This project received funding from the European Union's Horizon 2020 research and innovation program 369 under the Marie Skłodowska-Curie grant agreement No 956229.

Acknowledgements

We acknowledge the GenOuest bioinformatics core facility
`https://www.genouest.org` for providing the computing infrastructure.

References

1. J. Alanko, B. Alipanahi, J. Settle, C. Boucher, and T. Gagie: *Buffering updates enables efficient dynamic de bruijn graphs.* Computational and Structural Biotechnology Journal, 19 2021, pp. 4067–4078.
2. B. Alipanahi, A. Kuhnle, S. J. Puglisi, L. Salmela, and C. Boucher: *Succinct dynamic de Bruijn graphs.* Bioinformatics, 37(14) 07 2021, pp. 1946–1952.
3. F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro: *A space and time-efficient index for the compacted colored de Bruijn graph.* Bioinformatics, 34(13) 06 2018, pp. i169–i177.
4. F. Almodaresi, M. Zakeri, and R. Patro: *PuffAligner: a fast, efficient and accurate aligner based on the Pufferfish index.* Bioinformatics, 37(22) 06 2021, pp. 4048–4055.
5. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev: *On the representation of de bruijn graphs.* Journal of Computational Biology, 22(5) 2015, pp. 336–352, PMID: 25629448.
6. R. Chikhi, A. Limasset, and P. Medvedev: *Compacting de bruijn graphs from sequencing data quickly and in low memory.* Bioinformatics, 32 06 2016, pp. i201–i208.
7. A. Cracco and A. Tomescu: *Extremely-fast construction and querying of compacted and colored de Bruijn graphs with GGCAT.* bioRxiv, 2022.
8. V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie: *Practical dynamic de Bruijn graphs.* Bioinformatics, 34(24) 06 2018, pp. 4189–4195.
9. H. Guo, Y. Fu, Y. Gao, J. Li, Y. Wang, and B. Liu: *degsm: Memory scalable construction of large scale de bruijn graph.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 18(6) 2021, pp. 2157–2166.
10. S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer: *Phobic: Perfect hashing with optimized bucket sizes and interleaved coding.* arXiv, 2024.
11. G. Holley and P. Melsted: *Bifrost - Highly parallel construction and indexing of colored and compacted de Bruijn graphs.* bioRxiv, 2019.

12. S. T. Horsfield, N. J. Croucher, and J. A. Lees: *Accurate and fast graph-based pangenome annotation and clustering with ggcaller*. Genome Research, 2023.
13. B. Hou, R. Wang, and J. Chen: *Long read error correction algorithm based on the de bruijn graph for the third-generation sequencing*, in 2021 4th International Conference on Information Communication and Signal Processing (ICICSP), 2021, pp. 616–620.
14. 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, 23(1) 2022, p. 190.
15. T. Lemane, P. Medvedev, R. Chikhi, and P. Peterlongo: *kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections*. Bioinformatics Advances, 2(1) 04 2022.
16. A. Limasset, J.-F. Flot, and P. Peterlongo: *Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs*. Bioinformatics, 36(5) 02 2019, pp. 1374–1381.
17. B. Liu, H. Guo, M. Brudno, and Y. Wang: *deBGA: read alignment with de Bruijn graph-based seed and extension*. Bioinformatics, 32(21) 07 2016, pp. 3224–3232.
18. C. Marchet, Z. Iqbal, D. Gautheret, M. Salson, and R. Chikhi: *REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets*. Bioinformatics, 36(Supplement_1) 07 2020, pp. i177–i185.
19. C. Marchet, M. Kerbiriou, and A. Limasset: *BLight: efficient exact associative structure for k-mers*. Bioinformatics, 37(18) 04 2021, pp. 2858–2865.
20. J. R. Miller, S. Koren, and G. Sutton: *Assembly algorithms for next-generation sequencing data*. Genomics, 95(6) 2010, pp. 315–327.
21. I. Minkin, S. Pham, and P. Medvedev: *TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes*. Bioinformatics, 33(24) 09 2016, pp. 4024–4032.
22. G. E. Pibiri: *Sparse and skew hashing of K-mers*. Bioinformatics, 38(Supplement_1) 06 2022, pp. i185–i194.
23. M. Schatz, A. Delcher, and S. Salzberg: *Assembly of large genomes using second-generation sequencing*. Genome research, 20 09 2010, pp. 1165–73.
24. S. Sheikhizadeh, M. E. Schranz, M. Akdel, D. de Ridder, and S. Smit: *PanTools: representation, storage and exploration of pan-genomic data*. Bioinformatics, 32(17) 08 2016, pp. i487–i493.

Supplementary Materials

Figures

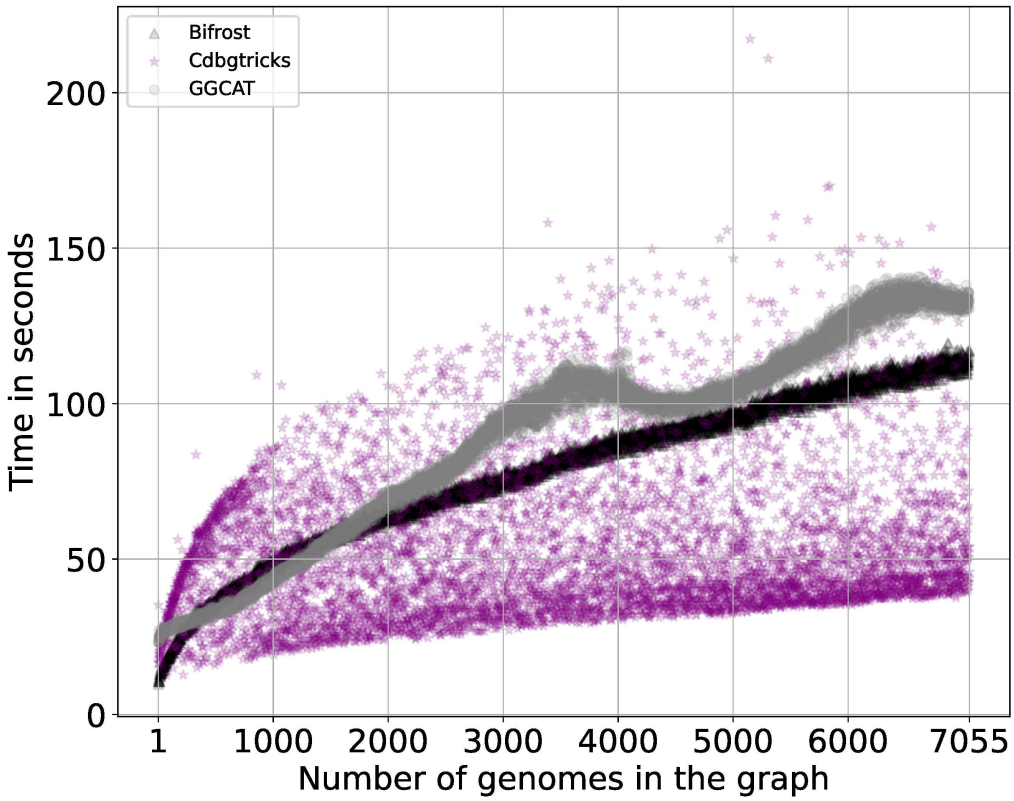

Figure 8. Results on *E. coli* genomes dataset. Time is given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`.

Algorithms

Algorithm 2 Test k-mer presence

Require: Hash Table of unitigs U, k-mer s, Table of buckets T, Table of mphfs F
Ensure: s belongs to a unitig

function KmerPresence(H,s,T,F)
- $mini_s \leftarrow canonicalminimizer(s)$ ▷ compute the canonical minimizer of s
- $i \leftarrow T[mini_s]$ ▷ retrieve the identifier of the bucket of $mini_s$
- **if** $i \neq NIL$ **then** ▷ the minimizer is in the index
  - $f_i \leftarrow F[i]$ ▷ retrieve the mphf of the bucket
  - $q \leftarrow canonical(s)$ ▷ compute the canonical form of s
  - $j \leftarrow f_i(q)$ ▷ compute the hash value of q
  - $p_j \leftarrow P[i][j]$ ▷ retrieve the position tuple of q
  - $id \leftarrow p_j.u_{id}$ ▷ retrieve the unitig identifier
  - $o \leftarrow p_j.u_{off}$ ▷ retrieve the offset of q in this unitig
  - $u \leftarrow U[id]$ ▷ retrieve the unitig
  - return $canonical(u(o, o+k)) = q$ ▷ compare the k-mers
- return false ▷ the minimizer is not in the index, so s is not in the graph

Algorithm 3 Construct funitigs

Require: Hash table of new k-mers K,Index of the graph I
Ensure: A set of funitigs X

```
function ConstructFunitig(K)
    X ← ∅
    for each k-mer s ∈ K do
        if s is not marked as used then
            mark s as used
            s_1 ← ExtendRight(s)
            s_2 ← ExtendRight(s̄)
            X ← X ∪ {s̄_2 ⊙ s_1(k − 1, |s_1|)}
    return X
function ExtendRight(s)
    q ← s
    g ← s(1, k)                                   ▷ the (k-1)-suffix of s
    while True do
        a ← FindRightExtensions(g)
        if a ≠ ϵ then
            b ← FindRightExtensions(ḡ)
            if b ≠ ϵ then
                q ← q ⊙ a
                g ← g(1, k − 1) ⊙ a
            else
                break
        else
            break
    return q
function FindRightExtensions(x)                   ▷ x is a (k-1)-mer
    extension ← ϵ
    count ← 0
    for each a ∈ ∑ do
        q ← x ⊙ a
        if q ∈ K then
            extensions ← a
            count ← count + 1
        else if KMERPRESENCE(I.U,s,I.T,I.F) then  ▷ x is found in the graph
            extension ← ϵ
            break
    if count ≠ 1 then
        extension ← ϵ
    return extension
```