Buffering Updates Enables Efficient Dynamic de Bruijn Graphs

Jarno Alanko[a,b,*], Bahar Alipanahi[c], Jonathen Settle[c], Christina Boucher[c,1], Travis Gagie[b,1]

[a]*Department of Computer Science, University of Helsinki, Helsinki, Finland*
[b]*Faculty of Computer Science, Dalhousie University, Halifax, Canada*
[c]*Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, USA*

Abstract

Motivation: The de Bruijn graph has become a ubiquitous graph model for biological data ever since its initial introduction in the later 1990s. It has been used for a variety of purposes including genome assembly (Zerbino and Birney, 2008; Bankevich et al., 2012b; Peng et al., 2012), variant detection (Alipanahi et al., 2020b; Iqbal et al., 2012), and storage of assembled genomes (Chikhi et al., 2016). For this reason, there have been over a dozen methods for building and representing the de Bruijn graph and its variants in a space- and time-efficient manner.
Results: Although there exists a plethora of space efficient data structures for storing the de Bruijn graph, the majority of them make a compression-mutability trade-off. In particular, with the exception of a few methods (Muggli et al., 2019; Holley and Melsted, 2020; Crawford et al., 2018), compressed and compact de Bruijn graphs do not allow for the graph to be efficiently updated, allowing for data to be added or deleted. The most recent compressed dynamic de Bruijn graph, (Alipanahi et al., 2020a), relies on dynamic bit vectors, which are slow in theory and practice. To address this shortcoming, we present BufBOSS which is a compressed dynamic de Bruijn graph that removes the necessity of dynamic bit vectors by buffering data that should be added or removed from the graph. We implement our method, which we refer to as BufBOSS, and compare its performance to Bifrost, DynamicBOSS, and FDBG. Our experiments demonstrate that BufBOSS achieves attractive trade-offs compared to other tools in terms of time, memory and disk, and has the best deletion performance by an order of magnitude out of all the tools that are able to perform deletions. Our implementation is available at `https://github.com/jnalanko/bufboss`
Contact: alanko.jarno@gmail.com

Keywords: de Bruijn graph, dynamic data structures, succinct data structures, Burrows-Wheeler transform

1. Introduction

Analyzing population level data has posed significant algorithmic challenges as the amount of the data has risen steadily in the past decade or more. For example, the 1,000 Genomes Project was concluded in 2012 (McVean et al., 2012), and more recently, the 100K Project concluded in 2018 (Turnbull et al., 2018). The MetaSub project (Danko et al., 2019), which collects metagenomic samples from subway systems across the world began in 2016 and is now in its fifth year. One of the main goals of these projects is to identify rare variants – including but not limited to single nucleotide polymorphisms (SNPs), insertions, and deletions – within the population, and to attribute them to physiological or disease outcome. One method of analysis that has been proposed and received a significant amount of attention, is the construction and analysis of the *colored de Bruijn graph*.

To define this concept correctly it is useful to first give a constructive definition of the de Bruijn graph as follows: given a set of sequence reads R and integer k, the first step is to identify all unique k-length substrings (k-mers) occurring in R, then create a directed edge for each unique k-mer with node labels being the $(k-1)$-length prefix and the $(k-1)$-length suffix of the k-mer, and lastly, after all directed edges have been created, glue all nodes that have the same label. The original purpose of the de Bruijn graph was to assemble single genomes (Pevzner et al., 2001), however, the definition and purpose has expanded to the linked de Bruijn graph (Turner et al., 2018), positional de Bruijn graph (Cameron et al., 2017; Ronen et al., 2012; Alipanahi

*Corresponding author: `alanko.jarno@gmail.com`; Department of Computer Science, P.O. Box 68 (Pietari Kalmin katu 5), FI-00014 University of Helsinki, Finland

[1]These authors contributed equally.

et al., 2017) and paired de Bruijn graph (Medvedev et al., 2011) to name a few. The colored variant of the de Bruijn graph is arguably the most well-studied. It is constructed using the sequence data of a population of genomes, rather than a single genome. Here, a set of d sequence reads are given $R_1, R_2, .., R_d$, rather than a single set of sequence reads, and a color is assigned for each set of sequence reads. Using this assignment of colors, each edge (and node) is colored with color c_i if and only if the associated k-mer ($(k-1)$-mer) is contained in R_i. In this way, each edge (or node) is assigned one or more colors, which signify the individuals in the population that contain the corresponding k-mer ($(k-1)$-mer). The colored de Bruijn graph can then be seen as a de Bruijn graph constructed from the union of all k-mers and a binary matrix C that stores the colors of each k-mers, i.e., $C[i, j] = 1$ if the i-th k-mer is contained in R_j. Thus, by traversing the graph and finding shared and divergent paths, rare genetic variants occurring in the population can be identified.

Iqbal et al. (2012) were the first to present this concept of the de Bruijn graph and gave the first implementation, which was in turn used to analyze the 1,000 Genomes Project data. Yet, this initial naive implementation was not space efficient and thus, requires over three terabytes to analyze 3,765 strains of E.coli (Muggli et al., 2017); however, it motivated the need for space-efficient implementations that could handle population-level datasets. Hence, in the past couple years there has been an explosion in the interest in space-efficient colored de Bruijn graphs. Vari (Muggli et al., 2017) and Rainbowfish (Almodaresi et al., 2017), were the first space-efficient colored de Bruijn graphs methods. Both recognize that the colored de Bruijn graph can be made space-efficient by storing a single de Bruijn graph that is built from the union of all sets of sequence reads (i.e., $R_1 \cup R_2 \cup \cdots \cup R_d$), and the color matrix C define above. This representation allows the matrix to be compressed and stored independently of the graph. Bloom Filter Trie (BFT) (Holley et al., 2015) was an early method for compactly storing the colored de Bruijn graph. These initial developments were followed by many other methods, including Mantis (Pandey et al., 2018), and Bifrost (Holley and Melsted, 2020).

Several public consortium projects involve the analysis of large heterogeneous datasets – such as GenomeTrakr (Allard et al., 2016) and MetaSub (Danko et al., 2019) – which are continually evolving. For this reason, there is growing interest in dynamic, space-efficient colored de Bruijn graphs, which allow addition and deletion of new data. Given that there are methods for efficiently storing the color matrix in manner that is dynamic, one of the remaining challenges is how to store the de Bruijn graph in a manner that is mutable but also space- and time-efficient. This is challenging since mutability and compression are contradictory in nature. Even allowing for the addition of data into a compressed version of the de Bruijn graph requires some algorithmic cleverness; most notably, VariMerge (Muggli et al., 2019) and Bifrost (Holley and Melsted, 2020) allow only addition of new data. Hence, there are relatively few compressed data structures for storing the de Bruijn graph that allow for the addition and deletion of data; these include FDBG (Crawford et al., 2018) and DynamicBOSS (Alipanahi et al., 2020a). In Table 1, we give an overview of the basic de Bruijn graph data structures and the operations that they support. Although DynamicBOSS is capable of handling large datasets and is fully dynamic, it is slow due to the reliance on dynamic bit vector libraries, i.e., the library of Prezza (2017). Although dynamic bit vectors are being improved, we think it would be better to avoid their use and implement dynamism using static data structures instead.

Our contribution is simply this: a dynamic, space-efficient de Bruijn graph representation that eliminates the need for dynamic bit vectors. We implement dynamism using the combination of a static, space-efficient representation of the de Bruijn graph, and two auxiliary data structures, which buffer data that is to be inserted and deleted. As requests for insertions or deletions are made, the corresponding buffers are updated, and after a prescribed number of insertions or deletions, the static de Bruijn graph is rebuilt. This amortizes the cost of dynamic operations over the cost of rebuilding from time to time. Traversal or use of the graph takes into account the static de Bruijn graph as well as the insertion and deletion buffer, which thus makes the static nature of the underlying data structure opaque. We describe how to: (a) buffer the data that is to be inserted or deleted, (b) support graph traversal taking into account the updates in the buffer (c) merge the updates to the graph efficiently.

We compare our method to existing dynamic de Bruijn graphs – namely Bifrost (Holley and Melsted, 2020), FDBG (Crawford et al., 2018), and DynamicBOSS (Alipanahi et al., 2020a). The evaluation criteria are the memory and time needed to construct the data structure, the size of the data structure on disk, the time required to add and delete data from the graph, and the time required to perform look-up queries of k-mers. In addition to these existing methods, we also implement and compare against a modified version of FDBG that we refer to as FDBG-RecSplit, where RecSplit (Es-

	BufBOSS	DynamicBOSS	FDBG	Bifrost	Pufferfish	BFT	Mantis
Addition	Yes	Yes	Yes	Yes	No	No	No
Deletion	Yes	Yes	Yes	No	No	No	No

Table 1: An overview of recent de Bruijn graph implementations and their attributes. In theory, BFT is capable of addition of data but it is no longer supported or functional (Holley, 2019).

posito et al., 2020b) is used for minimal perfect hashing. RecSplit was presented in 2020 and shows to have superior computational performance against competing methods.

BufBOSS is clearly was superior to all methods that support addition and deletion (i.e., DynamicBOSS and FDBG) in both memory and time on all large datasets. Bifrost was the closest competitor but does not support deletion and our method was more performant than Bifrost in construction on large metagenomic datasets. In particular, our results show that BufBOSS was up to five times faster and used up to 30% less memory to construct than the closest competitor (Bifrost). The peak memory for addition can be pushed to lowest in the competition, beating Bifrost by a small margin and the rest by a factor of 8 or more. The performance of deletion was the best out of all method that support deletion, i.e., by a factor of 26 in time, and 13 in memory.

2. Preliminaries

In this section, we briefly go over some of the basic terminology and definitions that will be used throughout this paper.

2.1. Basic definitions

A string X, is represented as a sequence of characters: $\mathsf{X}[1..n] = \mathsf{X}[1]\mathsf{X}[2]\ldots\mathsf{X}[n]$, where n is the length of the string. Each character will be drawn from an ordered alphabet Σ of size σ. $\mathsf{X}[i..j]$ refers to the substring of X given by $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$, where $1 \leq i \leq j \leq n$. Using this notation, a suffix is a substring with $j = n$, while a prefix is a substring with $i = 1$. An empty string is denoted with ε, and it is considered to be a prefix and a suffix of any string. A string of length k is called a k-mer. The lexicographic order of two strings X and Y is defined by the order of the characters at the first mismatching position from the left. If no mismatch exists, i.e. one string is a prefix of the other, the prefix is defined to be smaller. The *colexicographic* order of strings is the same, but characters are compared from right to left, and in case of no mismatch, X is smaller than Y iff X is a suffix of Y.

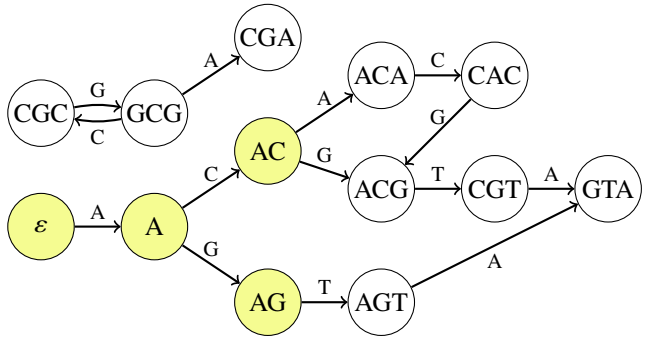
De Bruijn graphs are widely used in bioinformatics. The definitions used by bioinformaticians however differ from the graph theoretic roots of the concept in mathematics. In graph theory, a de Bruijn graph of order k over an alphabet Σ is a graph containing a node for all $|\Sigma|^k$ unique k-length strings. There is a directed edge from node u to node v if and only if the last $k-1$ characters of u are equal to the first $k-1$ characters of v.

In bioinformatics, de Bruijn graphs are subgraphs of the mathematical de Bruijn graph. Here, a de Bruijn graph is defined by the order k and a set of input strings $S_1, \ldots, S_m$, which correspond to individual genomes, sequence reads, or other DNA sequences. The literature splits into two different conventions. In one, the graph is fully defined by its set of nodes, and in the other, by its set of edges. In the *node-centric* definition, the nodes are all distinct k-mers of the input string. There is an edge from node u to node v if and only if the last $k-1$ characters of u are equal to the first $k-1$ characters of v. In other words, the node-centric de Bruijn graph is the subgraph of the mathematical order-k de Bruijn graph over the alphabet of the input strings, induced by taking the nodes corresponding all k-mers in the input strings, and adding all edges between them.

In the *edge-centric* definition, the de Bruijn graph is such that the vertices are all the distinct $(k-1)$-mers of the input strings. There is an edge from $(k-1)$-mer u to $(k-1)$-mer v if and only if there exists a k-mer in some input string that is prefixed by v and suffixed by u. This graph is a subgraph of the de Bruijn graph of order $k-1$ over the alphabet of the input strings. Contrary to the node-centric definition, not all edges in the subgraph induced by the nodes have to be included.

In this paper, we use the edge-centric definition. To mitigate confusion on whether k refers to the length of edges or nodes, we call the $(k-1)$-mers representing the nodes *nodemers* and the k-mers representing the edges *edmegers*. We denote the $(k-1)$-mer represented by a node v with $\ell(v)$ and the k-mer represented by an edge e with $\ell(e)$. A nodemer or an edgemer from the DNA alphabet ACGT is said to be *canonical* if it is equal to its reverse complement.

Figure 1: The Wheeler graph on the right corresponds to the edge centric de Bruijn graph with edgemers of length 4 from the set of strings ACGTA, ACACGT, AGTA and GCGCGCGA. The dummy node tree is highlighted in yellow. The colexicographic order of the nodes is the following: ε, A, ACA, CGA, GTA, AC, CAC, CGC, AG, ACG, GCG, AGT, CGT. The string EBWT is the concatenation of outgoing edge labels sorted by the order of the node at the origin of the edge. The EBWT is shown in the figure with vertical separators showing where the node at the origin changes. These separators are for illustration purposes only and are not included in the EBWT string. The sequence of out-degrees in colexicographic order is 1,2,1,0,0,2,1,1,1,1,2,1,1, whereas the sequence of in-degrees is 0,1,1,1,2,1,1,1,1,2,1,1,1. The Wheeler graph index represents these by encoding degree d with bit string $1{\cdot}0^d$, and concatenating the representations. The concatenated representation of out-degrees is denoted with O and the concatenated representation of in-degrees with I. The original BOSS takes another route and effectively adds an outgoing \$-edge from GTA to ε and from CGA to ε, which eliminates in-degrees and out-degrees of zero, and marks the first incoming and last outgoing edge to/from each node.



EBWT = A|CG|C| | |AG|G|G|T|T|AC|A|A

O = 1010010111001010101001010

I = 1101010100101010100101010

2.2. Overview of BOSS

The BOSS data structure (Bowe et al., 2012) is a generalization of the classic FM-index (Ferragina and Manzini, 2005) to de Bruijn graphs. It is a special case of the more general *Wheeler graph index* (Gagie et al., 2017). Here, we will describe the BOSS the data structure in terms of the more general Wheeler graph index, as our implementation is based on that representation.

The Wheeler graph index for an edge-centric de Bruijn graph G is based on a larger graph W that contains G as a subgraph. The edges of W are labeled by single characters, and the nodes are unlabeled. In the part of W that is the subgraph G, edges are labeled by the last characters of the corresponding edgemers in G. This construction guarantees that all incoming paths of length $(k-1)$ to a node v spell the same string. Thus one can walk back $(k-1)$ nodes in the graph to extract the $(k-1)$-mer represented by a node. However, in subgraph G alone, some nodes might not have an incoming path of length $(k-1)$. To amend this, we add for every node v with in-degree 0 a new path of k nodes, each representing a proper prefix of the $(k-1)$-mer represented by v, all the way to the empty prefix ε. The added edges are labeled by the last character of the corresponding prefix of the destination node. If the same prefix is added for two different nodes, the nodes representing those prefixes are glued together. These added nodes and edges are called *dummy nodes* and *dummy edges*. The notations $\ell(v)$ and $\ell(e)$ are extended for dummy nodes v and dummy edges e to denote the prefix represented by the node or the edge. The dummy nodes and edges form a tree starting from the node of the empty string, such that the leaves of the tree have outgoing edges to nodes of subgraph G.

The Wheeler index on W is based on a colexicographic order $\prec$ on the nodes and edges of W. For nodes u and v, it is defined that $u \prec v$ iff $\ell(u)$ is colexicographically smaller than $\ell(v)$. The edges are ordered colexicographically among themselves in the same way. The Wheeler index consists of three main components: (1) a string EBWT, which is the concatenation of all edge labels of W sorted by the order of the node at the origin of the edge, with ties broken arbitarily (2) a bit vector encoding the outdegrees of all nodes of W in the colexicographic order (3) a bit vector encoding the indegrees of all nodes of W in the colexicographic order. Figure 1 shows a worked example. Remarkably, these data structures define the graph completely.

The colexicographic ordering implies that if v and u are nodes such that $v \prec u$, and both have an outgoing edge with the same label c leading to destinations v' and u' respectively, then $v' \prec u'$. This means that if we have an interval $[i, j]$ of nodes in the order, and follow all outgoing edges from nodes in the interval with the same label c, we arrive at another contiguous interval $[i', j']$ of nodes. This, in turn, implies that it's enough compute just the endpoints i' and j' because all the other destinations will fall in between them. This can be done in $O(\log \sigma)$ time using rank queries on EBWT and rank/select queries on the in-degree and out-degree bit vectors (Gagie et al., 2017). Armed with this, we can locate the interval of nodes at the ends of paths labeled with any pattern P in $O(|P| \log \sigma)$ time by starting from the interval of all nodes, and updating the interval $|P|$ times following the characters of P. In the case of the de Bruijn graph, this gives us an algorithm to locate any nodemer in the graph and to traverse edges in the graph forward and backward.

The original BOSS representation of Bowe et al. (2012) uses the same ideas, but the implementation is slightly different. In this representation, nodes with in-degree or out-degree of zero are forbidden. This limitation allows us to represent the information of the in-degrees and out-degrees by marking the last outgoing edge from each node and the first incoming edge to each node. This comes at the cost of introducing a new character \$ to the alphabet and adding extra edges labeled with \$ to EBWT to ensure that every node has at least one outgoing and incoming edge, but it allows a shorter representation of the in-degrees and out-degrees.

3. Dynamizing Compact Data Structures

Bentley and Saxe (1980) proposed an approach to making static data structures semi-dynamic when queries are decomposable, where *semi-dynamic* means we can support insertions but not deletions and *decomposable* means we can quickly answer a query about the union of two disjoint sets when given the answers to the queries about those two sets separately. Decomposable queries include membership, minimum, maximum and mean, for example, but not mode.

To see how Bentley and Saxe's approach works, suppose we have a data structure D whose instances store static sets and support a decomposable queries, and a semi-dynamic set S. We partition S into $O(\log |S|)$ disjoint subsets whose sizes are distinct powers of 2, and building instances of D for those subsets. We maintain the invariant that there is a subset of size 2^t if and only if the $(t+1)$st bit from the right is a 1 in the binary representation of $|S|$.

To query S, we query the instances of D for all the subsets, and combine the answers to get the answer for all of S. To insert a new element x into S, we pool x with the elements from the subsets of size 1, 2, 4, etc, destroying the instances of D for those subsets as we go, until we find a size 2^t for which there is no subset. Since x is one element and $\sum_{i=0}^{t-1} 2^i = 2^t - 1$, we can now build an instance of D for the pooled subset of size 2^t, and maintain our invariant. The dynamic version of D for S we obtain this way is a constant factor larger than the static version of D for S, answers queries an $O(\log |S|)$ factor more slowly, and supports each insertion in amortized $O(P(S)\log(|S|)/|S|)$-time, where $P(S)$ is the time to build the static version of D for S.

Overmars and van Leeuwen (1981) extended Bentley and Saxe's approach to make the amortized complexity of insertions worst-case, using copies of the subsets and background processing, and to support deletions. Two techniques for supporting deletions are keeping a "ghost" instance of D that holds the deleted elements, and "lazily" deleting elements by marking them in the dynamic version of D for S and then collapsing a subset and rebuilding when more than half its elements have been lazily deleted.

Munro et al. (2015) adapted Bentley and Saxe's and Overmars and Van Leeuwen's results to dynamize compact but static data structures, focusing on indexes for document collections. They proposed keeping a fast but space-inefficient dynamic data structure for the elements that have been inserted or deleted most recently, as well as a series of compact static data structures for increasingly larger subsets, according to Bentley and Saxe's scheme; when the dynamic data structure grows too large, they empty it by rebuilding some of the static data structures and incorporating the buffer's contents. Munro et al. (2015) emphasized that previous approaches to dynamizing compact data structures generally relied on dynamic bitvectors, for which there are fairly strong lower bounds (Fredman and Saks, 1989). By maintaining a dynamic, pointer-based buffer and occasionally rebuilding a compact static data, they can avoid using dynamic bitvectors and thus side-step those lower bounds.

Munro et al. also explained how their ideas could be used to obtain dynamic compact representations of graphs, and Coimbra et al. (2020) recently applied those ideas to obtain an implementation of dynamic k^2-trees that is competitive with previous implementations, which were essentially just static k^2-trees but with dynamic instead of static bitvectors. Since Dynamic-BOSS (Alipanahi et al., 2020a) is essentially the data structure of Bowe et al. (2012) but with dynamic instead of static bit vectors, we are naturally curious how competitive a dynamic de Bruijn graph based on Munro et al.'s ideas will be.

For simplicity and practicality, in this paper we use only one compact static data structure in addition to the dynamic buffer. The main challenge is rebuilding the compact static data structure still in small space, since many data structures that are compact once built are not compact to build, and an implementation is not really compact if it is usually small but balloons every so often.

4. Buffering Insertions and Deletions

In this section we describe how to support insertions and deletions on a static de Bruijn graph by using a dynamic buffer. Here, we denote $G = (V, E)$ as the (original) graph before any updates. We denote A as the set

of edgemers we want to add, and D as the set of edgemers we want to delete. We assume that $A \cap D = \emptyset$, $A \cap E = \emptyset$ and $D \subseteq E$. The edge set of the modified graph G' is $E' = (E \cup A) \setminus D$ and the node set V' is derived from E' as the set of all nodemers that are prefixes or suffixes of edgemers in E'.

Since G is represented in a BOSS format, the edges and nodes of G are identified by their colexicographic ranks. We denote the colexicographic rank of a node $v \in V$ with $\mathsf{colex}(v) = |\{\ell(u) \preceq \ell(v) \ : \ u \in V\}|$, where $\preceq$ is denotes the colexicographic comparison and $\ell(v)$ is the label of v. Similarly for an edge $e \in E$, we denote $\mathsf{colex}(e) = |\{\ell(d) \preceq \ell(e) \ : \ d \in E\}|$. We abstract the BOSS structure behind the following interface.

- **Node search:** Given a nodemer $\ell(v)$, return $\mathsf{colex}(v)$ if $v \in V$ or report that $v \notin V$ in $O(k \log \sigma)$ time.
- **Edge search:** Given an edgemer $\ell(e)$, return $\mathsf{colex}(e)$ if $e \in E$ or report that $e \notin E$ in $O(k \log \sigma)$ time.
- **Out-edge label set:** Given $\mathsf{colex}(v)$, list all outgoing edge labels from v in $O(\mathsf{outdegree}(v))$ time.
- **In-edge label:** Given $\mathsf{colex}(v)$, return the incoming edge label to v, if exists, in $O(1)$ time (by construction, all incoming edges to a node always have the same label).
- **Out-edge rank:** Given $\mathsf{colex}(v)$ for a node v and a character c, return the colexicographic rank the edge with label $\ell(v) \cdot c$, if exists, in $O(\log \sigma)$ time.
- **Forward:** Given the representation of an edge e either as its rank $\mathsf{colex}(e)$, or the pair $(\mathsf{colex}(v), c)$, where v is the origin of the edge and c is the edge label, return $\mathsf{colex}(u)$, where u is the node at the destination of an edge, in $O(\log \sigma)$ time.
- **Backward:** Given $\mathsf{colex}(v)$ for a node v, return $\mathsf{colex}(u)$, where u is a node that has an outgoing edge to v, or report that no such edge exists, in $O(\log \sigma)$ time. If there are multiple candidates for u, we can return any.
- **In-edge interval:** Given $\mathsf{colex}(v)$, return the interval $[i, j]$ of the colexicographic ranks of the incoming edges to v in $O(1)$ time.

For implementation details of these operations on a Wheeler graph, see Gagie et al. (2017). We now describe how to use these operations and the buffers A and D to offer graph traversal functionality for G'.

We represent the addition buffer A with a hash table H_A, where the keys are node labels and the values are the sets of added incoming and outgoing edge labels. If an edge e is outgoing from the node, its label is the last character of $\ell(e)$, but if it is incoming, its label is defined here as the *first* character of $\ell(e)$. Note that the label in the incoming direction is usually not the same as the label of the edge in the Wheeler graph, but rather the label of an incoming edge to a node u that is at distance $k - 2$ from the origin of e backwards. The keys are packed into integers with $\lceil \log \sigma \rceil$ bits per character. This allows efficient hashing in $O(k \log(\sigma)/w)$ expected time, where w is the width of a machine word in the RAM-machine model. In practice, we limit $k \leq 32$, so that with $\sigma = 4$, all the keys can be represented with single 64-bit words. The values of H_A are encoded with two bit vectors using σ bits for the forward direction and $\sigma + 1$ bits for the backward direction as the special \$-symbol is also a possible incoming label. Table H_A provides node and edge membership queries to A in $O(k \log(\sigma)/w)$ expected time, given a node label $\ell(v)$ or an edge label $\ell(e)$.

The deletion buffer D is represented with just a single bit vector B_D of length $|E|$, such that $B_D[\mathsf{colex}(e)] = 1$ if and only if $e \in D$. This bit vector provides membership queries to D in constant time given $\mathsf{colex}(e)$, and in $O(k \log \sigma)$ time given $\ell(e)$ by computing $\mathsf{colex}(e)$ with the BOSS structure.

Adding an edgemer e to the addition set A is done by splitting e into the $(k - 1)$-length prefix and suffix nodemers v and u, and setting the bits corresponding to the last and first character of e in $H_A[\ell(v)]$ and $H_A[\ell(u)]$, respectively. Adding an edgemer e to the deletion set D is done by searching e using the BOSS to compute $\mathsf{colex}(e)$ and marking the position in B_D. We keep the table H_A and the vector B_D synchronized so that when we add an edgemer to H_A, we remove it from B_D if present, and vice versa. We do not support node deletions or additions explicitly because the node set is implicitly defined as the set of endpoints of all edges.

We now describe the query interface to the modified graph G'. Instead of operating on node identifiers $\mathsf{colex}(v)$, we operate on node *tokens*, which are pairs $(\mathsf{colex}(v), H_A[\ell(v)])$, where the first element of the pair is null if $v \notin V$ and the second element is null if $\ell(v)$ is not a key of H_A.

We can use a token to report whether the represented node exists in G'. Given the pair $(\mathsf{colex}(v), H_A[\ell(v)])$, the node v exists in G' if and only if one of the following holds (i) $H_A[\ell(v)]$ is not null (i.e. v is a prefix or a suffix of some edge in the addition set), or (ii) $\mathsf{colex}(v)$ is not null (i.e. v is in the BOSS structure) and there

is an incoming or outgoing edge to v in the BOSS that is not deleted. Checking case (ii) can be implemented by checking for the existence of a 0-bit in B_D in the in-edge range of v and at the colexicographic rank of every outgoing edge.

We can list all outgoing edge labels from a node represented by a token $(\mathsf{colex}(v), H_A[\ell(v)])$ by taking the union of the BOSS out-edge listing and the edges marked in $H_A[\ell(v)]$, and removing the edges marked in B_D.

Forward traversal from a node in G' is done by checking the existence of the edge in G' using the node token, and then updating $\mathsf{colex}(v)$ with a BOSS traversal step and $H_A[\ell(v)]$ with hash table lookup.

If the number of dynamic additions and deletions is small, we can optimize the time of graph traversal at the cost of a little space by marking in the BOSS those nodes which are affected by dynamic operations. These are the nodes whose labels are currently keys in H_A or that have at least one incoming or outgoing edge marked in B_D. With this, we do not have to to look up data from H_A and B_D on those nodes, and can rely on the static BOSS most of the time.

5. Batched Updates of Additions and Deletions

In this section, we describe a method to update a BOSS structure with a batch of additions and deletions. The input to the algorithm is a static BOSS structure $G = (V, E)$, a set A of edgemers we want to add, and a set D of edgemers we want to delete. The sets A and D are represented with the hash table H_A and the bit vector B_D described in the previous section. The output is a BOSS structure containing the set of edgemers of G with the set A added and D deleted.

5.1. Updating with Additions

We add the edgemers of A to G using a slight modification of the Vari-Merge algorithm (Muggli et al., 2019), which can be summarized by the following two steps: (1) a planning step that determines the set of common edgemers between the two graphs, and (2) an executing step that inserts the non-repetitive set of edgemers of one graph into another. The modification arises from the fact that we that we make additions from H_A directly rather than building a BOSS data structure from H_A and then making additions.

After an update with additions, it could be that some of the dummy nodes in the graph are no longer needed. These redundant dummy nodes might accumulate in the graph over the course of multiple updates. To address this potential issue, next we describe an algorithm to detect all redundant dummy nodes and edges in time proportional only to the total number of dummy nodes and edges.

We recall that the dummy nodes form a tree starting from the node of the empty string. We do a depth-first search in the tree of dummies from the root using the forward traversal operation in the BOSS interface. When we reach a leaf node the dummy tree, we follow all the outgoing edges to full nodemers. We check the in-degree of each such nodemer. If it is two or more, we can mark the incoming edge from the tree part for deletion, because the target node has another incoming edge which must come from another nodemer. If all out-edges of the dummy node were marked, we can mark the in-edge of the dummy node for deletion as well. Likewise, when we backtrack in the depth-first search, if all out-edges of a dummy tree node are marked for deletion, we mark its in-edge for deletion. In the end can we delete all marked edges using the update algorithm described in the next subsection.

5.2. Updating with Deletions

The approach of Vari-Merge can be readily applied for deletions as well, however, there is one complication arising from the dummy nodes of BOSS of G. In particular, we may need to add new dummy nodes and edges to G when we delete an edgemers in order to ensure that every node has an incoming path of length $(k-1)$. Hence, we need to identify when dummy nodes need to be added, and add them to the graph in an efficient manner.

The nodes that require insertion of an incoming dummy chain after deletions can be detected as a preprocessing step. For every edge that is marked for deletion in B_D, we first check whether all in-edges to its destination node v are marked for deletion. If this is the case, we check in the addition buffer if there are new incoming edges to v. If not, we need to add the incoming dummy chain. To do these checks, we need to know $H_A[\ell(v)]$ and the in-degree range of v. We can find both of these by using the static BOSS structure. We traverse the edge to v to find $\mathsf{colex}(v)$, which lets us compute the in-degree range, and then retrieve the label $\ell(v)$ by backward traversing $k-1$ steps, which enables us to look up $H_A[\ell(v]$).

Now to apply the deletions D, we first add all the dummy paths that will be required, by running the Vari-Merge algorithm for the dummies, and then run a version of Vari-Merge that deletes the edgemers of D.

It is possible to combine the addition of A, the addition of the required dummies after deletion of D, and the

deletion of D into just one adapted run of Vari-Merge. Moreover, since we have the edgemers of A and D already in memory, we don't have to build the BOSS of them, but can instead just sort them colexigraphically, which already enables us to simulate the Vari-Merge procedure. In fact, we do not even have to sort D, but can rather just ignore the edges marked in B_D in the executing step.

6. Results

In this section, we demonstrate the performance of BufBOSS against a variety of dynamic de Bruijn graph implementations from the literature. The experiments were run on a server with an Intel Xeon CPU E5-2640 v4 with 40 cores clocked at 2.40GHz, equipped with 755GiB of RAM. The timing and peak memory (RSS) were measured using the Unix utility `/usr/bin/time`.

6.1. Competing Tools

We compare the performance of BufBOSS to the following dynamic de Bruijn graph implementations: DynamicBOSS (Alipanahi et al., 2020a), FDBG (Crawford et al., 2018), FDBG-RecSplit, and Bifrost (Holley and Melsted, 2020). An overview of recent de Bruijn graph implementations and their attributes is given in Table 1. Hence, we did not compare against BFT (Holley et al., 2015), Vari (Muggli et al., 2017), Rainbowfish (Almodaresi et al., 2017), and Pufferfish (Almodaresi et al., 2018), and Mantis (Pandey et al., 2018) because they cannot perform addition or deletion of data.

6.2. Implementation of FDBG-RecSplit

As previously mentioned, we modified FDBG and compare against this modification in addition to original FDBG implementation. Here, we give some background on this modification. Alipanahi et al. (2020a) observed that FDBG fails for datasets approaching 2^{32} distinct nodemers. This is due to hash collisions. FDBG hashes nodemers first with the Karp-Rabin hash function (Karp and Rabin, 1987), and then hashes these hashes with the perfect hash function BBHash (Limasset et al., 2017). The problem is that the Karp-Rabin hash values are stored in 64-bit integers for best compatibility with BBHash, which means that collisions start to happen often at around $\sqrt{2^{64}} = 2^{32}$ keys due to the birthday paradox phenomenon in probability. In the interest of studying the scalability of FDBG for datasets that have more than 2^{32} nodemers, we edited the implementation to use 128-bit Karp-Rabin hashes, and changed the perfect hashing implementation from BBHash to a recently published new implementation called RecSplit (Esposito et al., 2020a). RecSplit is a good choice for this purpose because it is optimized to work for uniformly random 128-bit keys, and the Karp-Rabin hashes of the nodemers satisfy this model well. Arithmetic with 128-bit values is significantly slower than with 64-bit values, especially when modulo-operations are involved, so we also include results with the original FDBG.

6.3. Implementation of BufBOSS

We construct the BOSS structure by first using the KMC3 k-mer counter (Kokot et al., 2017) to list all distinct edgemer of the data. KMC3 is highly parallel, using both machine-level parallel vector instructions, and multi-threading to exploit the all cores available on the CPU.

We then write to disk triples (x, c, b), where x is a nodemer string, c is a character either to the left or to the right of x in the input data, and b is a bit that indicates whether c is on the left or on the right. We sort these triples on disk by the colexicographic order of the nodemers x using the stxxl library (Dementiev et al., 2008). We then scan the sorted pairs to count the number of characters to the left of x. If some nodemer x does not have characters on the left, we need to add the dummy nodes corresponding to all prefixes of x to the graph. We write these prefixes to another file on disk along with the characters of the right, and sort these colexicographically by the order of the prefixes. Finally, we merge the order of the dummy nodes into the order of the nodemers by streaming the two lists on disk and using logic of the merge phase of merge sort. While we are doing this merge we have, in the correct order, all the information to build the data structure: for every nodemer x, we have the count of characters on the left (the in-degree of x), and the outgoing edge labels (the out-degree and the labels of the outgoing edges from x), all in the colexicographic order of the nodemers.

When adding sequences to BufBOSS dynamically, we set a threshold $t \in [0, 1]$ such that if the addition buffer H_A contains fraction of more than t entries compared to the number of edgemers in the static BOSS structure, we flush the buffers H_A and B_D by running the update algorithm described in Section 5, and clearing the buffers afterwards. This amortizes the cost of individual updates over the buffer flushes, with different time-space trade-offs available for different values of t.

The rank and select structures required for queries are implemented with the SDSL-library. The queries are

implemented in our general-purpose Wheeler graph library. The dynamic buffer hash table H_A is implemented with the C++ standard library. Like FDBG, we store edgemers in 64-bit integers, so the maximum allowed edgemer length is 32.

6.4. Datasets

Our first set of datasets is short reads from the bacteria E. coli. We used 28,428,648 paired-end reads generated from whole genome sequencing of E.coli K-12 substr. MG1655 dataset (NCBI SRA accession ERX002508). We refer to this as 28M-e. Next, we split this sample to generate smaller datasets of sizes 20,000, 200,000, 2,000,000, and 14,000,000 reads, which we denote by 20K-e, 200K-e, 2M-e and 14M-e, respectively.

The second set of inputs is a series of metagenomic read sets of increasing size from a study on antimicrobial resistant determinants in commercial beef production (Noyes et al., 2016). The NCBI SRA accession number is PRJNA292471. We downloaded read sets that have 55,242,004 and 11,136,890 reads, which we refer to as 55M and 11M, respectively. We generated smaller samples by randomly selecting 20,000, 200,000 and 2,000,000 sequence reads from 11M sample, which we refer to these as 20K, 200K and 2M, respectively. Next, we concatenated three datasets, which consist of 55,242,004, 44,035,852, and 52,833,978 reads, to create a dataset with over 150M reads. We refer to this as dataset as 150M. We then added samples 4, 5, 9, 22, 25, 33, 34 and 51 to sample 150M to generate the sample 600M.

Since most of the tools we experiment with do not understand characters outside of the DNA-alphabet {A,C,G,T}, such as the invalid character N, we split the reads into pieces that contain only characters from the DNA alphabet. Since FDBG, FDBG-Recsplit and DynamicBOSS do not index reverse complements, but BufBOSS and Bifrost do, we concatenate the input read sets with their reverse complements for the former three tools to make the graphs the same for all tools.

6.5. Construction

For each E. coli dataset, we build the index of each tool using 50% of the reads, then add the next 25% of the reads and finally delete the remaining 25%. Addition and deletion are shown in the next subsection. Here, we discuss the construction time, space and memory. Table 3 shows the time and peak memory for construction as well as the size of the final data structure on disk. In addition, Figures 2a and 2b illustrate the time versus memory required for construction of the data structure using the largest E. coli dataset, and the time versus disk space required for construction of the data structure using the largest E. coli dataset.

The k-mer preprocessing steps of all tools are included in the time and peak memory numbers. BufBOSS was consistently the fastest index to construct, being almost 4 times faster than the closest competitor DynamicBOSS on the largest dataset. Our index construction takes a lot of memory on the small datasets, but this is due to KMC3 always requiring a large amount of memory even on small datasets. When the size of the data increases, our peak memory becomes more competitive. Bifrost had the smallest memory by a factor of 5 to the closest competitors DynamicBOSS and BufBOSS. DynamicBOSS had the smallest index on disk, being half the size of BufBOSS. This is due to the BOSS-implementation of DynamicBOSS being geared more toward small size rather than speed.

We test further scalability of construction using the metagenome datasets, omitting the original FDBG as it can not handle more than 2^{32} nodemers. The data for the rest of the tools is in Table 2. Figure 2c shows the time-memory plot for the 55M dataset, which is the largest FDBG-Recsplit was able to process before running out of memory, and Figure 2d shows the same plot for the 150M dataset without FDBG-Recsplit.

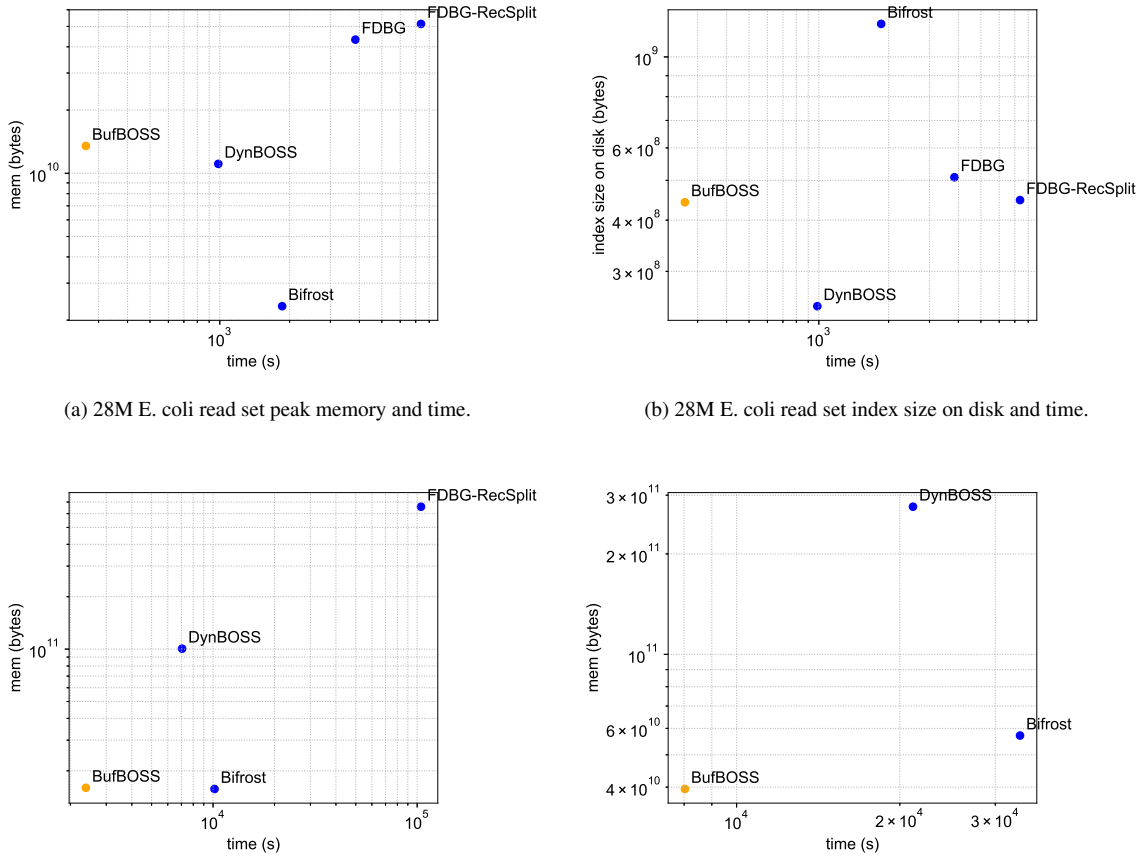
The external memory construction of BufBOSS now starts to show its scalability on the largest dataset, with the lowest peak memory out of all tools, while maintaining the fastest construction. DynamicBOSS again has the smallest index size on disk. The peak construction memory of FDBG-Recsplit is very large, and exceeds the 755GiB memory capacity of the machine already on the 150M dataset. This enormous peak memory is probably caused by the fact that FDBG-Recsplit holds all input edgemers in memory while building the index. The k-mer preprocessing step of dynamicBOSS also exceeded the 755GiB capacity of the machine on the 600M dataset.

6.6. Addition and Deletion

Since only two tools out of five are able to scale on construction of the metagenome datasets, we benchmark additions, deletions and queries only on the E. coli dataset. For the E. coli datasets, we evaluated the resources needed to add an additional 25% of the reads, and delete the remaining 25% of the reads. We added and deleted using the graphs that were constructed for the previous section. For deletion, we only compare to FDBG-Recsplit, FDBG and DynamicBOSS since

	BufBOSS			DynamicBOSS			FDBG-RecSplit			Bifrost		
	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time
20K	3.75	5381	0.14	2.04	381	0.52	4.47	535	0.69	1.82	22.65	0.13
200K	35.5	6658	0.38	19.6	2506	1.75	43.8	4885	7.06	18.8	108	1.14
2M	311	10312	2.98	172	7821	8.45	392	43050	75.3	198	1455	10.4
11M	1348	13129	13.4	744	34350	35.2	1723	188155	444	1158	6147	53.2
55M	3737	15234	39.7	2062	95908	118	4799	627340	1741	4875	14992	169
150M	10237	37653	134	5651	264221	353	OOM	OOM	OOM	14738	54454	555
600M	31406	115249	558	OOM	OOM	OOM	OOM	OOM	OOM	60500	165888	2730

Table 2: Index construction on the metagenome datasets. The units of disk and peak memory are MB (2^{20} bytes) and the unit of time is minutes. FDBG is not included because it runs into hash collisions for large datasets. FDBG-RecSplit ran out of memory (OOM) on the 150M dataset. The edgemer packing preprocessing step of DynamicBOSS ran out of memory on the 600M dataset. The number of distinct canonical edgemers in these datasets in increasing order of size are 1203852, 11839506, 107122222, 478210723, 1360576988, 3741452498 and 11676829812.



(a) 28M E. coli read set peak memory and time.

(b) 28M E. coli read set index size on disk and time.

(c) 55M metagenome read set time and peak memory.

(d) 150M metagenome read set time and peak memory

Figure 2: Index construction.

Bifrost does not perform deletion. Tables 4 and 5 illustrate the time, memory and disk required for addition and deletion, respectively. Also, Figures 3 and 4 illustrate the time versus space for additions and deletions, respectively.

To fairly evaluate the time for addition for Bifrost, we discount the time to load the index into memory. This is because Bifrost does not serialize its index to disk, but rather writes all maximal non-branching paths of the de Bruijn graph (unitigs) to disk in a graph for-

	BufBOSS			DynamicBOSS			FDBG-RecSplit			FDBG			Bifrost		
	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time
20K-e	2.04	5272	0.12	1.10	115	0.73	2.35	276	0.35	2.64	240	0.12	1.00	22.6	0.07
200K-e	11.9	5919	0.21	6.51	1144	0.91	14.5	1576	2.46	16.4	1328	0.86	9.63	54.1	0.42
2M-e	41.0	7135	0.43	22.7	2673	2.28	45.0	5265	9.62	51.0	4466	4.34	83.2	169	2.02
14M-e	210	10312	2.11	117	5221	9.31	212	24313	54.3	241	20462	29.2	549	1112	13.6
28M-e	422	12870	4.41	235	10573	16.4	427	49023	123	485	41271	64.1	1148	2221	31.0

Table 3: Construction on the E. coli dataset. The units of disk and peak memory are MB (2^{20} bytes) and the unit of time is minutes. The number of distinct canonical edgemers in these datasets in increasing order of size are 626875, 4087049, 13241253, 63704311 and 128431292.

	BufBOSS			DynamicBOSS			FDBG-RecSplit			FDBG			Bifrost		
	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time
20K-e	3.07	22.6	0.73	1.56	89.0	12.9	13.2	327	0.94	13.5	328	0.36	1.52	28.4	0.06
200K-e	15.6	96.3	2.78	7.69	651	59.4	48.7	1449	8.96	50.5	1451	3.22	13.7	74.6	0.30
2M-e	57.8	293	13.7	30.3	2532	262	202	10126	91.5	208	10132	30.0	124	391	1.36
14M-e	337	1364	97.8	168	11697	1628	1294	68744	699	1323	68773	245	828	1596	8.14
28M-e	569	2310	148	303	18217	2434	1923	122954	1389	1982	123013	449	1578	2698	14.6

Table 4: Additions after construction of 3 on the E. coli dataset. The units of disk and peak memory are MB (2^{20} bytes), and the unit of time is minutes. BufBOSS uses the buffer fraction parameter 0.01. The number of distinct canonical edgemers in these datasets in the addition set in increasing order of size are 329523, 2501881, 9047060, 38892710 and 56315888.

	BufBOSS			DynamicBOSS			FDBG-RecSplit			FDBG		
	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time	Disk	Memory	Time
20K-e	3.07	10.9	0.02	1.51	88.8	3.55	14.8	224	0.84	15.1	225	0.29
200K-e	15.6	40.1	0.12	6.15	658	105	103	1246	28.9	104	1248	12.8
2M-e	57.8	135	1.51	31.3	2477	292	345	8033	88.5	351	8039	39.4
14M-e	334	813	17.4	183	12925	767	1602	52932	349	1631	52961	204
28M-e	569	1259	32.6	323	16345	1044	2367	96480	875	2425	96538	536.9

Table 5: Deletions after construction of table 3 and additions of table 4 on the E. coli dataset. The units of disk and peak memory are MB (2^{20} bytes), and the unit of time is minutes. Bifrost is not included because it does not support deletions. The number of distinct canonical edgemers in these datasets in the deletion set in increasing order of size are 330563, 2526285, 8786286, 43705141 and 48377370.

mat. When Bifrost loads the index, it must re-index the whole dataset by computing and hashing k-mer minimizers for all unitigs in the data. This loading takes two minutes on the 28M E. coli read set. The loading time of other tools is negligible in our experiment. Hence we subtract Bifrost's loading time out to avoid skewing the addition performance results.

Figure 3 shows a few different time-space trade-offs BufBOSS can achieve by varying the buffer flushing threshold t. With the values of $t = 0.025$ or $t = 0.01$, BufBOSS achieves the lowest peak memory out of all tools, losing in time only to Bifrost.

FDBG and FDBG-Recsplit perform very poorly with additions, because they store new nodemers in a hash table as strings. This blows up both the peak memory and the size of the index on disk. The smallest index on disk is DynamicBOSS, being about half as small as BufBOSS in the end. Bifrost has the fastest additions even when BufBOSS does not flush the buffer at all ($t = 1.0$).

In deletion efficiency, BufBOSS is by far superior, being an order of magnitude better than the other tools in both time and peak space. The deletions of BufBOSS are very efficient because we only mark the deleted edgemers in a bit vector. No buffer flushes are required since the space of the deletion buffer bit vector B_D does not depend on the number of deletions. We note that the other tools could adopt a similar technique for deletions.

6.7. Edge Existence Queries

We test the existence query speeds of the tools on six different types of input data. The first three query types are single edgemers: we query edgemers sampled from the construction read set, the addition read set, and completely randomly generated edgemers which do not exist in the data with high probablity. The last three are the same, except that instead of single edgemers, we query all edgemers of whole read sequences in the same

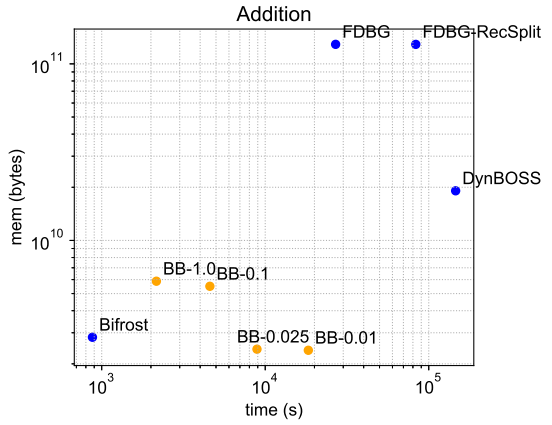


Figure 3: Addition performance on the 28M E. coli dataset. The data points labeled with BB-t are runs on BufBOSS with different buffer fraction parameters t.

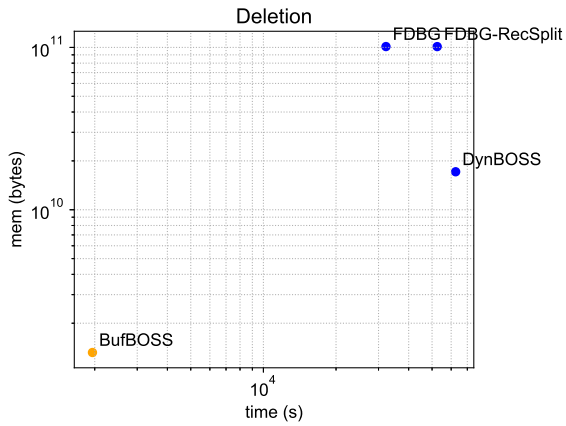


Figure 4: Deletion performance on the 28M E. coli dataset. Here we do not vary the buffer fraction parameter for BufBOSS because buffer flushes are not needed when doing only deletions.

query, which speeds up the query time per edgemer in BufBOSS, because if the queries exist in the data, we can just traverse the de Bruijn graph instead of searching all the edgemers separately.

All the queries are done on the index with the construction and addition edgemers, but without having deleted the final 25% of edgemers, because for consistency, we want all the queries to be done on the same de Bruijn graph, and Bifrost can not delete edgemers.

Again we factor out the time to load the index for all tools. Figure 5 shows the average query times on the different types of queries. Bifrost has the fastest queries on all query types, with BufBOSS coming in second. DynamicBOSS queries are significantly slower than those of BufBOSS because of the heavy price DynamicBOSS pays for using dynamic bit vectors in the implementation. We see that BufBOSS benefits significantly if the queries are given as whole reads rather than single edgemers, because if the reads are in the graph, we do not have to search every edgemer from scratch, but can traverse the graph instead. This improves the time complexity of querying all edgemers of a read of length m from $O(km \log \sigma)$ to $O(m \log \sigma)$. Querying edgemers from the set of added edgemers did not make a noticeable difference compared to querying from the initial construction set. Random edgemers are faster to query than single existing edgemers, because the search can terminate early when a prefix of the edgemer is found to not exist in the graph, yet still slower than querying whole existing reads because of the efficiency of graph traversal.

The query efficiency of Bifrost is explained partly by good memory locality. Bifrost finds the unitig containing the current edgemer, and can scan forward sequentially in memory in the unitig. On the other hand, the BOSS structure has to jump around in memory in an unpredictable way, resulting in a large number of cache misses. FDBG and FDBG-Recsplit face a similar issue because adjacent edgemers being stored by their hash values in unpredicatable memory locations. Sirén et al. (2020) recently developed a technique to improve BWT-based data structures' memory locality, but so far it has been applied only in the context of variation graphs.

7. Discussion and Future Work

We have shown that buffering updates into a BOSS data structure can provide attractive trade-offs in terms of time, memory and disk usage compared to other tools. Our approach to deletions in particular is currently clearly the most efficient method for dynamic

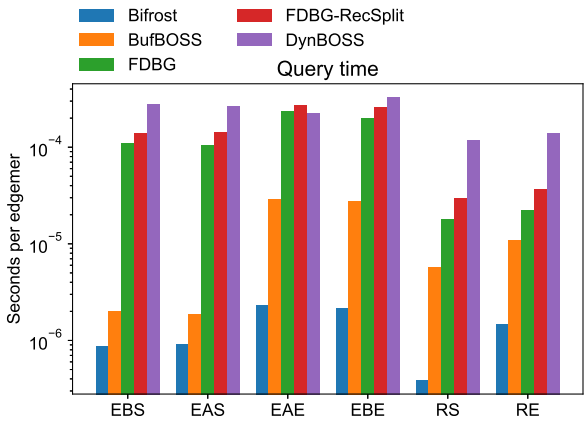


Figure 5: Edge existence queries for six different types of edgemers against the 28M E. coli index including the added sequences. The query types are the following: EBS = existing build sequence, EAS = existing added sequence, EAE = existing added edgemer, EBE = existing build edgemer, RS = random sequence, RE = random edgemer.

deletions on a de Bruijn graph. Our index construction was also the fastest, which is mainly due to the efficiency of the libraries KMC3 and stxxl used for edgemer listing and sorting. An important caveat in construction is that while BufBOSS was ran single-threaded as the other tools, the external libraries KMC3 and stxxl used for BufBOSS construction use parallelism nonetheless.

Observing the source code of the tools reveals shortcomings in the implementations of all the tools. For example, the Bifrost index on disk is just a representation of the de Bruijn graph as a collection of unitigs and edge pointers, but it does not include indexing structure, and thus Bifrost has to rebuild the index every time it is run. FDBG, FDBG-Recsplit and DynamicBOSS load all the distinct edgemers into memory at once for additions and deletions, even though this should not be needed. All the tools could in principle implement deletions efficiently by simply marking the deleted edgemers, like BufBOSS does. These kinds of shortcomings indicate that much could still be gained by more careful implementations of the methods, and that the results we obtained do not necessarily reflect the fundamental limitations of the approaches. Currently, we find Bifrost to be the most mature implementation.

The implementation of BufBOSS could also be improved. Recently, Egidi et al. (2020) consider Wheeler graph merging in addition to de Bruijn graph merging, and although, they show that the problem is computationally more challenging, they propose an algorithm specific to BOSS that has a lower peak space than Vari-Merge in theory. It is worth noting that these results are still only theoretical and they do not provide any implementation. Nonetheless, the practicality of their results warrants future investigation.

More drastic modifications possible to BufBOSS include applying the technique of Sirén et al. (2020) for improving memory locality, and maintaining a dynamic longest-common prefix (LCP) array to allow us to change the order k of the graph while navigating it, up to some maximum order set at construction time. Previously, Boucher et al. (2015) showed how the BOSS data structure can be augmented to support variable orders; Belazzougui et al. (2016) and Belazzougui et al. (2018) showed how the resulting data structure can be made bidirectional; and Díaz-Domínguez et al. (2019) showed how the LCP array can be replaced by a succinct tree shape, at the cost of the order no longer being known.

Although there are tools that recommend the best fixed order for a de Bruijn graph (Chikhi and Medvedev, 2014), variations in read coverage, particularly from single-cell sequencing, can mean that no single order is appropriate for an entire genome or set of genomes. Some tools, such as SPAdes (Bankevich et al., 2012a) and IDBA (Peng et al., 2010), use several iterations with different orders, but rebuilding massive graphs this way would be impractical and defeat the purpose of the dynamization. As far as we know, there is currently no algorithm to maintain a dynamic LCP array, it seems like a reasonable extension of BufBOSS. In contrast, it seems unlikely that Bifrost, for example, can be made to support variable orders without greatly increasing its memory usage.

8. Funding

This work was partly funded by the Academy of Finland (grant 309048) and NSERC grant RGPIN-07185-2020.

References

Alipanahi, B., Kuhnle, A., Puglisi, S., Salmela, L., Boucher, C., 2020a. Succinct Dynamic de Bruijn Graphs. Bioinformatics btaa546.

Alipanahi, B., Muggli, M., Jundi, M., Noyes, N., Boucher, C., 2020b. Metagenome SNP calling via read-colored de Bruijn graphs. Bioinformatics btaa081.

Alipanahi, B., Salmela, L., Puglisi, S.J., Muggli, M., Boucher, C., 2017. Disentangled Long-Read De Bruijn Graphs via Optical Maps, in: Prof of WABI, pp. 1:1–1:14.

Allard, M.W., Strain, E., Melka, D., Bunning, K., Musser, S.M., Brown, E.W., Timme, R., 2016. Practical value of food pathogen traceability through building a whole-genome sequencing network and database. J Clin Microbiol 54, 1975–1983.

Almodaresi, F., Pandey, P., Patro, R., 2017. Rainbowfish: A succinct colored de Bruijn graph representation, in: Proc. of WABI, pp. 251–265.

Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R., 2018. A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics 34, i169–i177.

Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., et al., 2012a. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. Journal of computational biology 19, 455–477.

Bankevich, A., et al., 2012b. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J Comput Biol 19, 455–477.

Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., Puglisi, S.J., 2016. Bidirectional variable-order de bruijn graphs, in: LATIN 2016: Theoretical Informatics, Springer. pp. 164–178.

Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., Puglisi, S.J., 2018. Bidirectional variable-order de bruijn graphs. International Journal of Foundations of Computer Science 29, 1279–1295.

Bentley, J.L., Saxe, J.B., 1980. Decomposable searching problems I: Static-to-dynamic transformation. Journal of Algorithms 1, 301–358.

Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K., 2015. Variable-order de bruijn graphs, in: 2015 data compression conference, IEEE. pp. 383–392.

Bowe, A., et al., 2012. Succinct de Bruijn graphs, in: Proc. WABI, pp. 225–235.

Cameron, D., et al., 2017. GRIDSS: sensitive and specific genomic rearrangement detection using positional de Bruijn graph assembly. Genome Res 27, 2050–2060.

Chikhi, R., Limasset, A., Medvedev, P., 2016. Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics 32, i201–i208.

Chikhi, R., Medvedev, P., 2014. Informed and automated k-mer size selection for genome assembly. Bioinformatics 30, 31–37.

Coimbra, M.E., Francisco, A.P., Russo, L.M., De Bernardo, G., Ladra, S., Navarro, G., 2020. On dynamic succinct graph representations, in: 2020 Data Compression Conference (DCC), IEEE. pp. 213–222.

Crawford, V., Kuhnle, A., Boucher, C., Chikhi, R., Gagie, T., 2018. Practical Dynamic de Bruijn Graphs. Bioinformatics , 4189 – 4195.

Danko, D., et al., 2019. Global genetic cartography of urban metagenomes and anti-microbial resistance. bioRxiv URL: `https://www.biorxiv.org/content/early/2019/10/04/724526.1`, doi:`10.1101/724526`.

Dementiev, R., Kettner, L., Sanders, P., 2008. Stxxl: standard template library for xxl data sets. Software: Practice and Experience 38, 589–637.

Díaz-Domínguez, D., Gagie, T., Navarro, G., 2019. Simulating the dna overlap graph in succinct space, in: 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Egidi, L., Louza, F., Manzini, G., 2020. Space efficient merging of de bruijn graphs and wheeler graphs. arXiv .

Esposito, E., Graf, T.M., Vigna, S., 2020a. Recsplit: Minimal perfect hashing via recursive splitting, in: 2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM. pp. 175–185.

Esposito, E., Mueller-Graf, T., Vigna, S., 2020b. RecSplit: Minimal Perfect Hashing via Recursive Splitting, in: Prof of ALENEX, pp. 175–185. doi:`10.1137/1.9781611976007.14`.

Ferragina, P., Manzini, G., 2005. Indexing compressed text. JACM 52, 552–581.

Fredman, M., Saks, M., 1989. The cell probe complexity of dynamic data structures, in: Proceedings of the twenty-first annual ACM symposium on Theory of computing, pp. 345–354.

Gagie, T., Manzini, G., Sirén, J., 2017. Wheeler graphs: A framework for bwt-based data structures. Theoretical computer science 698, 67–78.

Holley, G., 2019. Personal email communication with authors of BFT.

Holley, G., Melsted, P., 2020. Bifrost–highly parallel construction and indexing of colored and compacted de bruijn graphs. Genome Bio 21, 249.

Holley, G., Wittler, R., Stoye, J., 2015. Bloom filter trie–a data structure for pan-genome storage, in: Proc. of WABI, pp. 217–230.

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G., 2012. De novo assembly and genotyping of variants using colored de bruijn graphs. Nat Genet 44, 226–232.

Karp, R.M., Rabin, M.O., 1987. Efficient randomized pattern-matching algorithms. IBM journal of research and development 31, 249–260.

Kokot, M., Długosz, M., Deorowicz, S., 2017. Kmc 3: counting and manipulating k-mer statistics. Bioinformatics 33, 2759–2761.

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

McVean, G., et al., 2012. An integrated map of genetic variation from 1,092 human genomes. Nature 491, 56–65.

Medvedev, P., Pham, S., Chaisson, M., Tesler, G., Pevzner, P., 2011. Paired de Bruijn Graphs: A Novel Approach for Incorporating Mate Pair Information into Genome Assemblers. J Comput Biol 18, 1625–1634.

Muggli, M., Alipanahi, B., Boucher, C., 2019. Building large updatable colored de bruijn graphs via merging. Bioinformatics 35, i51–i60. doi:`https://doi.org/10.1093/bioinformatics/btz350`.

Muggli, M., Bowe, A., Noyes, N., Morley, P., Belk, K., Raymond, R., Gagie, T., Puglisi, S., Boucher, C., 2017. Succinct colored de bruijn graphs. Bioinformatics 33, 3181–3187.

Munro, I., Nekrich, Y., Vitter, J.S., 2015. Dynamic data structures for document collections and graphs, in: Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 277–289.

Noyes, N., et al., 2016. Resistome diversity in cattle and the environment decreases during beef production. eLife 5, e13195.

Overmars, M.H., van Leeuwen, J., 1981. Worst-case optimal insertion and deletion methods for decomposable searching problems. Information Processing Letters 12, 168–173.

Pandey, P., Almodaresi, F., Bender, M., Ferdman, M., Johnson, R., Patro, R., 2018. Mantis: A fast, small, and exact large-scale sequence-search index. Cell 7, 201–207.

Peng, Y., Leung, H.C., Yiu, S.M., Chin, F.Y., 2010. Idba–a practical iterative de bruijn graph de novo assembler, in: Annual international conference on research in computational molecular biology, Springer. pp. 426–440.

Peng, Y., et al., 2012. IDBA-UD: A *de novo* assembler for single-cell and metagenomic sequencing data with highly uneven depth. Bioinformatics 28.

Pevzner, P., Tang, H., Waterman, M., 2001. An Eulerian path approach to DNA fragment assembly. Proc Natl Acad Sci 98, 9748–9753.

Prezza, N., 2017. A framework of dynamic data structures for string processing, in: Proc of SEA, p. 11:1–11:15.

Ronen, R., Boucher, C., Chitsaz, H., Pevzner, P., 2012. SEQuel: improving the accuracy of genome assemblies. Bioinformatics 28, i188–i196.

Sirén, J., Garrison, E., Novak, A.M., Paten, B., Durbin, R., 2020. Haplotype-aware graph indexes. Bioinformatics 36, 400–407.

Turnbull, C., et al., 2018. The 100,000 genomes project: bringing whole genome sequencing to the nhs. Br Med J 361.

Turner, I., Garimella, K., Iqbal, Z., McVean, G., 2018. Integrating long-range connectivity information into de Bruijn graphs. Bioinformatics 34, 2556—2565.

Zerbino, D., Birney, E., 2008. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. Genome Res 18, 821–829.