On the Representation of De Bruijn Graphs

Article *in* Journal of Computational Biology · January 2014

DOI: 10.1089/cmb.2014.0160 · Source: arXiv

5 authors, including:

Antoine Limasset
Université Libre de Bruxelles
21 PUBLICATIONS **262** CITATIONS

SEE PROFILE

Shaun Dunn Jackman
BC Cancer Agency
45 PUBLICATIONS **8,058** CITATIONS

SEE PROFILE

On the representation of de Bruijn graphs

Rayan Chikhi[1], Antoine Limasset[2], Shaun Jackman[3], Jared T. Simpson[4] and Paul Medvedev[1,5,6,†]

[1]Department of Computer Science and Engineering, The Pennsylvania State University, USA
[2]ENS Cachan Brittany, Bruz, France
[3]Canada's Michael Smith Genome Sciences Centre, Canada
[4]Ontario Institute for Cancer Research, Toronto, Canada
[5]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, USA
[6]Genomics Institute of the Huck Institutes of the Life Sciences, The Pennsylvania State University, USA

Abstract. The de Bruijn graph plays an important role in bioinformatics, especially in the context of *de novo* assembly. However, the representation of the de Bruijn graph in memory is a computational bottleneck for many assemblers. Recent papers proposed a navigational data structure approach in order to improve memory usage. We prove several theoretical space lower bounds to show the limitations of these types of approaches. We further design and implement a general data structure (DBGFM) and demonstrate its use on a human whole-genome dataset, achieving space usage of 1.5 GB and a 46% improvement over previous approaches. As part of DBGFM, we develop the notion of frequency-based minimizers and show how it can be used to enumerate all maximal simple paths of the de Bruijn graph using only 43 MB of memory. Finally, we demonstrate that our approach can be integrated into an existing assembler by modifying the ABySS software to use DBGFM.

1 Introduction

De novo assembly continues to be one of the fundamental problems in bioinformatics, with new datasets coming from projects such as the Genome10K [17]. The task is to reconstruct an unknown genome sequence from a set of short sequenced fragments. Most state-of-the-art assemblers (e.g. [14,23,2,41]) start by building a de Bruijn graph (dBG) [28,18], which is a directed graph where each node is a distinct k-mer present in the input fragments, and an edge is present between two k-mers when they share an exact $(k-1)$-overlap. The de Bruijn graph is the basis of many steps in assembly, including path compression, bulge removal, graph simplification, and repeat resolution [26]. In the workflow of most assemblers, the graph must, at least initially, reside in memory; thus, for large genomes, memory is a computational bottleneck. For example, the graph of a human genome consists of nearly three billions nodes and edges and assemblers require computers with hundreds of gigabytes of memory [14,23]. Even these large resources can be insufficient for many genomes, such as the 20 Gbp white spruce. Recent assembly required a distributed-memory approach and around a hundred large-memory servers, collectively storing a 4.3 TB de Bruijn graph data structure [3].

Several articles have pursued the question of whether smaller data structures could be designed to make large genome assembly more accessible [10,40,27,9,7]. Conway and Bromage [10] gave a lower bound on the number of bits required to encode a de Bruijn graph consisting of n k-mers: $\Omega(n \lg n)$ (assuming $4^k > n$). However, two groups independently observed that assemblers use dBGs in a very narrow manner [9,7] and proposed a data structure that is able to return the set of neighbors of a given node but is not necessarily able to determine if that node is in the graph. We refer to these as *navigational data structures* (NDS). The navigational data structures proposed in [9,7] require $O(n \lg k)$ and $O(n)$[1] bits (respectively), beating the Conway-Bromage lower bound both in theory and in practice [9].

What is the potential of these types of approaches to further reduce memory usage? To answer this question, we first formalize the notion of a navigational data structure and then show that any NDS requires at least $3.24n$ bits. This result leaves a gap with the known upper bounds; however, even if a NDS could be developed to meet this bound, could we hope to do better on inputs that occur in practice? To answer this, we consider a very simple class of inputs: simple paths. We show that on these inputs (called linear dBGs), there are both navigational and general data structures that asymptotically use $2n$ bits and give matching lower bounds. While dBGs occurring in practice are not linear, they can nevertheless be often decomposed into a small collection of long simple paths (where all the internal nodes have in- and out-degree of 1). Could we then take advantage of such a decomposition to develop a data structure that can achieve close to $2n$ bits on practical inputs?

We describe and implement a data structure (DBGFM) to represent de Bruijn graphs in low memory. The first step of the construction uses existing k-mer counting software to transform, in constant memory, the input

†Corresponding author, pashadag@cse.psu.edu

[1] The paper only showed the number of bits is $O(n \lg n)$. However, the authors recently indicated in a blog post [6] how the dependence on $\lg(n)$ could be removed, though the result has not yet been published.

sequencing dataset to a list of k-mers (i.e. nodes) stored on disk [29]. The second step is a novel low memory algorithm that enumerates all the maximal simple paths without loading the whole graph in memory. We achieve this through the use of non-lexicographic minimizers, ordered based on their frequency in the data. Finally, we use the FM-index [12] to store the simple paths in memory and answer membership and neighborhood queries.

We prove that as the number of simple paths decreases, the space utilization of DBGFM approaches $2n$ bits. In practice, DBGFM uses $4.76n$ bits on a human whole-genome dataset and $3.53n$ bits on a human chr14 dataset, improving the state-of-the-art [33] by 46% and 60%, respectively. We demonstrate the efficiency of frequency-based minimizers by collapsing the dBG of the human whole-genome dataset using only 43 MB of memory. Finally, we show how DBGFM can be integrated into an existing assembler by modifying the ABySS software [38] to use DBGFM instead of a hash table.

2 Previous Work

In the last three years, several papers and assemblers have explored novel data structures designed to reduce the space usage of dBGs, and we provide a brief summary of the results here.

ABySS was one of the first genome assemblers capable of representing large dBGs [38]. It uses an open-addressing hash table that stores the k-mers of the graph in the keys. The edges can be inferred from the nodes and do not need to be stored. For every k-mer, ABySS uses $2k$ bits to store the k-mer, plus an additional 43 bits of associated data (stored in 64 bits for ease of implementation). Therefore, in total, the space usage of the dBG data structure in ABySS is $(\ell^{-1}(2k + 64))$ bits per k-mer, where ℓ is the load factor of the hash table (set to 0.8). In the following, we focus on the space needed to store just the dBG, since the type of associated data varies greatly between different assemblers.

Conway and Bromage [10] gave a $\lg \binom{4^k}{n}$ bits lower bound for representing a dBG and demonstrated a sparse bit array data structure that comes close to achieving it. They used an edge-centric definition of the dBG (where edges are all the $(k + 1)$-mers, and nodes are prefixes and suffixes of length k), but their results trivially translate to node-centric dBGs by storing k-mers instead of $(k+1)$-mers. For a dataset with $k = 27$ and $12 \cdot 10^9$ edges (i.e. $(k + 1)$-mers), their theoretical minimum space is 22 bits per edge while their implementation achieves 28.5 bits per edge.

Later work explored the trade-offs between the amount of information retained from the de Bruijn graph and the space usage of the data structure. Ye *et al.* [40] showed that a graph equivalent to the de Bruijn graph can be stored in a hash table by sub-sampling k-mers. The values of the hash table record sequences that would correspond to paths between k-mers in the de Bruijn graph. The theoretical memory usage of this approach is $\Omega(k/g)$ bits per k-mer, where g is the distance between consecutive sampled k-mers. Pell *et al.* [27] proposed a practical lossy approximation of the de Bruijn graph that stores the nodes in a Bloom filter [4]. They found that a space usage of 4 bits per k-mer provided a reasonable approximation of the de Bruijn graph for their purpose (partitioning and down-sampling DNA sequence datasets). Yet, the structure has not yet been directly applied to *de novo* assembly.

Chikhi and Rizk [9] built upon the structure of Pell *et al.* by additionally storing the set of Bloom filter false positives (false neighbors of true nodes in the graph). In this way, their structure is no longer lossy. They obtained a navigational data structure that allowed the assembler to exactly enumerate the in- and out-neighbors of any graph node in constant time. However, the structure does not support node membership queries, and also does not support storing associated data to k-mers. The theoretical space usage is $(1.44 \lg(\frac{16k}{2.08}) + 2.08)$ bits per k-mer, under certain assumptions about the false positive rate of the Bloom filter. This corresponds to 13.2 bits per k-mer for $k = 27$.

The structure has recently been improved by Salikhov *et al.* with cascading Bloom filters [33], replacing the hash table by a cascade of Bloom filters. In theory, if an infinite number of Bloom filters is used, this scheme would require 7.93 bits per k-mer independently of k. The authors show that using only 4 Bloom filters is satisfactory in practice, yet they do not provide a formula for the theoretical space usage in this case. For $k = 27$ and $2.7 \cdot 10^9$ nodes, they computed that their structure uses 8.4 bits per k-mer. Bowe *et al.* [7] used a tree variant of the Burrows-Wheeler transform [8] to support identical operations. They describe a theoretical navigational data structure for representing the dBG of a set of input sequences that uses a space $4m + M \lg(m) + o(m)$ bits, where M is the number of input strings and m the number of graph edges. Note that the space is independent of k. Another data structure based on a similar principle has been recently proposed [32].

3 Preliminaries

We assume, for the purposes of this paper, that all strings are over the alphabet $\Sigma = \{A, C, G, T\}$. A string of length k is called a k-mer and U is the universe of all k-mers, i.e. $U = \Sigma^k$. The binary relation $u \to v$ between

two strings denotes an exact suffix-prefix overlap of length $(k-1)$ between u and v. For a set of k-mers S, the *de Bruijn graph* of S is a directed graph such that the nodes are exactly the k-mers in S and the edges are given by the $\rightarrow$ relation. We define S to be a *linear* dBG if there exists a string x where all the $(k-1)$-mers of x are distinct and S is the set of k-mers present in x. Equivalently, S is a linear dBG if and only if the graph is a simple path. The de Bruijn graph of a string s is the de Bruijn graph of all the k-mers in s. We adopt the node-centric definition of the de Bruijn graph, where the edges are implicit given the vertices; therefore, we use the terms de Bruijn graph and a set of k-mers interchangeably.

For a node x in the de Bruijn graph, let $\overleftarrow{ext}(x)$ be its four potential in-neighbors (i.e. $\overleftarrow{ext}(x) = \{y \ : \ y \in \Sigma^k, y \rightarrow x\}$) and $\overrightarrow{ext}(x)$ be its four potential out-neighbors. Let $ext(x) = \overrightarrow{ext}(x) \cup \overleftarrow{ext}(x)$. For a given set of k-mers S, let $ext(S) = \{ext(x), x \in S\}$ (similarly for $\overrightarrow{ext}(S)$ and $\overleftarrow{ext}(S)$).

We will need some notation for working with index sets, which is just a set of integers that is used to select a subset of elements from another set. Define $\text{IDX}(i, j)$ as a set of all index sets that select j out of i elements. Given a set of i elements Y and $X \in \text{IDX}(i, j)$, we then write $Y[X]$ to represent the subset of j elements out of Y, as specified by X. We assume that there is a natural ordering on the elements of the set Y, e.g. if Y is a set of strings, then the ordering might be the lexicographical one.

The families of graphs we will use to construct the lower bounds of Theorems 1 and 2 have k be a polylogarithmic function of $|S|$, i.e. $k = O(\log^c |S|)$ for some c. We note that in some cases, higher lower bounds could be obtained using families of graphs with $k = \Theta(n)$; however, we feel that such values of k are unrealistic given the sequencing technologies. On one hand, the value of k is a bounded from above by the read length, which is fixed and independent of the number of k-mers. On the other hand, k must be at least $\log_4(|S|)$ in order for there to be at least $|S|$ distinct k-mers.

4 Navigational data structures

We use the term *membership data structure* to refer to a way of representing a dBG and answering k-mer membership queries. We can view this as a pair of algorithms: $(\text{CONST}, \text{MEMB})$. The CONST algorithm takes a set of k-mers S (i.e. a dBG) and outputs a bit string. We call CONST a constructor, since it constructs a representation of a dBG. The MEMB algorithm takes as input a bit string and a k-mer x and outputs true or false. Intuitively, MEMB takes a representation of a dBG created by CONST and outputs whether a given k-mer is present. Formally, we require that for all $x \in \Sigma^k$, $\text{MEMB}(\text{CONST}(S), x)$ is true if and only if $x \in S$. An example membership data structure, as used in ABySS, is one where the k-mers are put into a hash table (the CONST algorithm) and membership queries are answered by hashing the k-mer to its location in the table (the MEMB algorithm).

Recently, it was observed that most assemblers use the MEMB algorithm in a limited way [9,7]. They do not typically ask for membership of a vertex that is not in $ext(S)$, but, instead, ask for the neighborhood of nodes that it already knows are in the graph. We formalize this idea by introducing the term navigational data structure (NDS), inspired by the similar idea of performing navigational queries on trees [11]. An NDS is a pair of algorithms, CONST and NBR. As before, CONST takes a set of k-mers and outputs a bit string. NBR takes a bit string and a k-mer, and outputs a set of k-mers. The algorithms must satisfy that for every dBG S and a k-mer $x \in S$, $\text{NBR}(\text{CONST}(S), x) = ext(x) \cap S$. Note that if $x \notin S$, then the behavior of $\text{NBR}(\text{CONST}(S), x)$ is undefined. We observe that a membership data structure immediately implies a NDS because a NBR query can be reduced to eight MEMB queries.

To illustrate how such a data structure can be useful, consider a program that can enumerate nodes using external memory (e.g. a hard drive or a network connection). Using external memory to navigate the graph by testing node membership would be highly inefficient because of long random access times. However, it is acceptable to get a starting node from the device and access the other nodes using the proposed data structure.

There are several important aspects of both a navigational and membership data structures, including the space needed to represent the output of the constructor, the memory usage and running time of the constructor, and the time needed to answer either neighborhood or membership queries. For proving space lower bounds, we make no restriction on the other resources so that our bounds hold more generally. However, adding other constraints (e.g. query time of $\lg n$) may allow us to prove higher lower bounds and is an interesting area for future work.

5 Navigational data structure lower bound for de Bruijn graphs

In this section, we prove that a navigational data structure on de Bruijn graphs needs at least 3.24 bits per k-mer to represent the graph:

Theorem 1. *Consider an arbitrary NDS and let* CONST *be its constructor. For any* $0 < \epsilon < 1$*, there exists a* k *and* $x \subseteq \Sigma^k$ *such that* $|\text{CONST}(x)| \geq |x| \cdot (c - \epsilon)$*, where* $c = 8 - 3 \lg 3 \approx 3.25$.

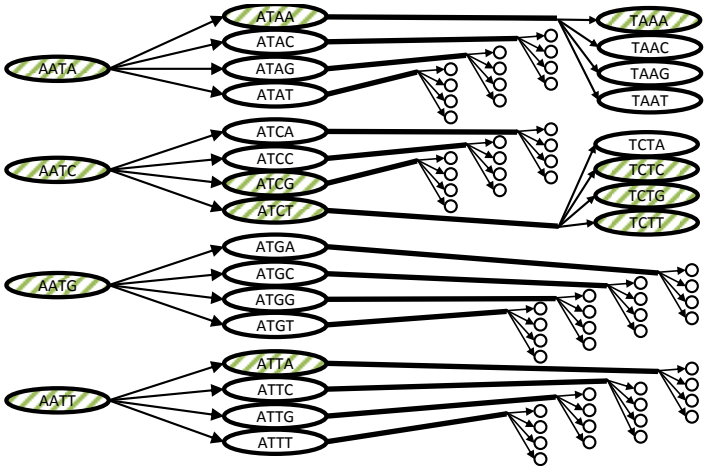

Fig. 1. Example of lower bound construction for $k = 4$. The figure shows T along with some of the node labels. The four nodes on the left form T_0, the 16 nodes in the middle are T_1, and the nodes on the right are T_2. For space purposes, some of the edges from T_1 to T_2 are grouped together. An example of a member from the family is shown with shaded vertices. Note that there are four vertices at each level, and together they form a subforest of T.

Our proof strategy is to construct a family of graphs, for which the number of navigational data structures is at least the size of the family. The full proof of the theorem is in the Appendix, however, we will describe the construction used and the overall outline here. Our first step is to construct a large dBG T and later we will choose subsets as members of our family. Fix an even $k \geq 2$, let $\ell = k/2$, and let $m = 4^{\ell-1}$. T will be defined as the union of $\ell + 1$ levels, $T = \cup_{0 \leq i \leq \ell} T_i$. For $0 \leq i \leq \ell$, we define the i^{th} level as $T_i = \{\text{“A}^{\ell-i}\text{T}\alpha\text{”} \ : \ \alpha \in \Sigma^{i+\ell-1}\}$. Observe that $T_i = \overrightarrow{ext}(T_{i-1})$, for $1 \leq i \leq \ell$. See Figure 1 for a small example.

We focus on constructing dBGs that are subsets of T because T has some desirable properties. In fact, one can show that the set of k-mers T induces a forest in the dBG of Σ^k (Lemmas 1 and 2 in the Appendix). Each member of our family will be a subforest of T that contains m vertices from every level.

Formally, suppose we are given a sequence of index sets $X = X_1, \ldots, X_\ell$ where every index set is a member of $\text{IDX}(4m, m)$. Each index set will define the subset of vertices we select from a level, and we define $L_0^X = T_0$ and $L_i^X = \overrightarrow{ext}(L_{i-1}^X)[X_i]$, for $1 \leq i \leq \ell$. Note that $L_i^X \subseteq T_i$. In this manner, the index sets define a set of k-mers $s(X) = \cup_{0 \leq i \leq \ell} L_i^X$. Finally, the family of graphs which we will use for our proof is given by:

$$S^k = \{s(X_1, \ldots, X_\ell) \ : \ \ell = k/2, m = 4^{\ell-1}, X_i \in \text{IDX}(4m, m)\}$$

To prove Theorem 1, we first show that each of the dBGs of our family have the same amount of k-mers (proof in Appendix):

Property 1. *For all $x \in S^k$, $|x| = 4^{\ell-1}(\ell + 1)$.*

Next, we show that each choice of X leads to a unique graph $s(X)$ (Lemma 3 in the Appendix) and use it to show that the numbers of graphs in our family is large, relative to the number of k-mers in each set (proof in the Appendix):

Property 2. *$|S^k| = \binom{4m}{m}^\ell \geq 2^{(c-\epsilon_0)\ell m}$, where $c = 8 - 3\lg 3$ and $\epsilon_0 = \lg(12m)/m$.*

Finally, we need to show that for any two graphs in the family, there is at least one k-mer that appears in both graphs but with different neighbors:

Property 3. *Let $x = s(X) \in S^k$ and $y = s(Y) \in S^k$ be two distinct elements in S^k. Then, there exists a k-mer $u \in x \cap y$ such that $\overrightarrow{ext}(u) \cap x \neq \overrightarrow{ext}(u) \cap y$.*

The proof of Theorem 1 now follows by using the pigeonhole principle to argue that the number of navigational data structures must be at least the size of our family, giving a lower bound on the bits per k-mer.

6 Linear de Bruijn graphs

In this section, we study data structures to represent linear de Bruijn graphs. Though a linear dBG will never occur in practice, it is an idealized scenario which lets us capture how well a data structure can do in the best case. The

bounds obtained here also serve as motivation for our approach in later sections, where we build a membership data structure whose space usage approaches our lower bound from this section the "closer" the graph is to being linear.

We can design a naive membership data structure for linear dBGs. A linear dBG with n k-mers corresponds to a string of length $n + k - 1$. The constructor builds and stores the string from the k-mers, while the membership query simply does a linear scan through the string. The space usage is $2(n + k - 1)$ bits. The query time is prohibitively slow, and we show in Section 7 how to achieve a much faster solution at the cost of a space increase.

We now prove that a NDS on linear de Bruijn graphs needs at least $2n$ bits to represent the graph, meaning one cannot do much better than the naive representation above. In general, representing all strings of length $n + k - 1$ would take $2(n + k - 1)$ bits, however, not all strings of this length correspond to linear dBGs. Fortunately, we can adapt a probabilistic result of Gagie [13] to quantify the number of strings of this length that have no duplicate k-mers (Lemma 5 in Appendix). Our strategy is similar to that of Section 5. We construct a large family of linear dBGs such that for any pair of members, there is always a k-mer that belongs to both but whose neighborhoods are different. We build the family by taking the set of all strings without duplicate $(k-1)$-mers and identifying a large subset having the same starting k-mer. We then show that by increasing the length of the strings and k, we can create a family of size arbitrarily close to 4^n (Lemma 6 in Appendix). Finally, we show that because each string in the family starts with the same k-mer, there always exists a distinguishing k-mer for any pair of strings. Using the pigeonhole principle, this implies that number of navigational data structures must be at least the number of distinct strings (proof in Appendix):

Theorem 2. *Consider an arbitrary NDS for linear de Bruijn graphs and let* CONST *be its constructor. Then, for any $0 < \epsilon < 1$, there exists (n, k) and a set of k-mers S of cardinality n, such that $|\text{CONST}(S)| \geq 2n(1 - \epsilon)$.*

Note that our naive membership data structure of $2(n+k-1)$ bits immediately implies a NDS of the same size. Similarly, Theorem 2's lower bound of $2n$ bits on a NDS immediately implies the same bound for membership data structures. In practice, k is orders of magnitude less than n, and we view these results as saying that the space usage of membership and navigational data structures on linear dBGs is essentially $2n$ and cannot be improved.

These results, together with Theorem 1, suggested that the potential of navigational data structures may be dampened when the dBG is linear-like in structure. Intuitively, the advantage of a linear dBG is that all the k-mers of a path collapse together onto one string and require storing only one nucleotide per k-mer, except for the overhead of the first k-mer. If the dBG is not linear but can still be decomposed into a few paths, then we could still take advantage of each path while paying an overhead of only a single k-mer per path. This in fact forms the basis of our algorithm in the next section.

7 Data structure for representing a de Bruijn graph in small space (DBGFM)

Recall that a simple path is a path where all the internal nodes have in- and out-degree of 1. A *maximal* simple path is one that cannot be extended in either direction. It can be shown that there exists a unique set of edge-disjoint maximal simple paths that completely covers the dBG, and each path p with $|p|$ nodes can be represented compactly as a string of length $k + |p| - 1$. We can thus represent a dBG S containing n k-mers as a set of strings corresponding to the maximal simple paths, denoted by $sp_k(S)$. Let $c_k(S) = |sp_k(S)|$ be the number of maximal simple paths, and let s to be the concatenation of all members of $sp_k(S)$ in arbitrary order, separating each element by a symbol not in Σ (e.g. \$). Using the same idea as in Section 6, we can represent a dBG using s in $2|s| = \sum_{p \in sp_k(S)} 2(|p| + k) \leq 2(n + (k + 2)c_k(S))$ bits. However, this representation requires an inefficient linear scan in order to answer a membership query.

We propose the use of the FM-index of s to speed up query time at the cost of more space. The FM-index [12] is a full-text index which is based on the Burrows-Wheeler transform [8,1] developed for text compression. It has previously been used to map reads to a reference genome [22,20,24], perform *de novo* assembly [36,37,21], and represent the dBG for the purpose of exploring genome characteristics prior to *de novo* assembly [35].

The implementation of the FM-index stores the Huffman-coded Burrows-Wheeler transform of s along with two associated arrays and some $o(1)$ space overhead. Our software, called DBGFM[2], follows the implementation of [12], and we refer the reader there for a more thorough description of how the FM-index works. Here, we will only state its most relevant properties. It allows us to count the number of occurrences of an arbitrary pattern q in s in $O(|q|)$ time. In the context of dBGs, we can test for the membership of a k-mer in S in time $O(k)$. Two sampling parameters (r_1 and r_2) trade-off the size of the associated arrays with the query time. For constructing the FM-index, there are external memory algorithms that do not use more intermediate memory than the size of the final output [12]. The space usage of DBGFM is $|s|(H_0(s) + 96r_1^{-1} + 384r_2^{-1}) + o(1) \leq n(H_0(s) + 96r_1^{-1} +$

[2] Source code available at `http://github.com/jts/dbgfm`

$384r_2^{-1})(1 + \frac{k+2}{n}c_k(S)) + o(1)$ bits. H_0 is the zeroth order entropy [1]: $H_0(s) = -\sum_{c \in \Sigma \cup \{\$\}} f_c \lg f_c$, where f_c is the frequency of character c in s. Note that for our five character alphabet H_0 is at most $\lg 5$.

As the value of $c_k(S)$ approaches one, $f_\$$ approaches 0 and hence the upper bound on H_0 approaches 2. If we further set the sampling parameters to be inversely proportional to n, the space utilization approaches at most $2n$ bits. However, this results in impractical query time and, more realistically, typical values for the sampling parameters are $r_1 = 256$ and $r_2 = 16384$, resulting in at most $2.32n$ bits as $c_k(S)$ approaches 1. For the error-free human genome with $k = 55$, there are $c_{55}(S) = 12.9 \cdot 10^6$ maximal simple paths and $n = 2.7 \cdot 10^9$ k-mers. The resulting $H_0(S)$ is at most 2.03, and the space utilization is at most 2.43 bits per k-mer.

An additional benefit of the FM-index is that it allows constant-time access to the in-neighbors of nodes — every edge is part of a simple path, so we can query the FM-index for the symbols preceding a k-mer x. Thus, DBGFM is a membership data structure but supports faster in-neighbor queries. However we note that this is not always the case when reverse complementarity is taken into account.

We wanted to demonstrate how the DBGFM data structure could be incorporated into an existing assembler. We chose ABySS, a popular *de novo* sequence assembly tool used in large-scale genomic projects [38]. In modifying ABySS to look up k-mer membership using DBGFM, we replace its hash table with a simple array. DBGFM associates each k-mer with an integer called a suffix array index (SAI), which could be used to index the simple array. However, some of the k-mers of the DBGFM string include a simple path separator symbol, \$, and, hence, not every SAI corresponds to a node in the dBG. We therefore use a rank/select data structure [15] to translate the SAIs into a contiguous enumeration of the nodes, which we then use to index our simple array. We also modified the graph traversal strategy in order to maximize the number of in-neighborhood queries, which are more efficient than out-neighborhood or membership queries.

8 Algorithm to enumerate the maximal simple paths of a de Bruijn graph in low memory (BCALM)

The DBGFM data structure of Section 7 can construct and represent a dBG in low space from the set of maximal simple paths ($sp_k(S)$). However, constructing the paths (called compaction) generally requires loading the k-mers into memory, which would require large intermediate memory usage. Because our goal is a data structure that is low-memory during both construction and the final output, we developed an algorithm for compacting de Bruijn graphs in low-memory (BCALM[3]).

Our algorithm is based on the idea of minimizers, first introduced by [30,31] and later used by [25]. The ℓ-minimizer of a string u is the smallest ℓ-mer that is a sub-string of u (we assume there is a total ordering of the strings, e.g. lexicographical). We define $\text{Lmin}(u)$ (respectively, $\text{Rmin}(u)$) to be the ℓ-minimizer of the $(k-1)$-prefix (respectively suffix) of u. We refer to these as the left and right minimizers of u, respectively. We use minimizers because of the following observation:

Observation 1. *For two strings u and v, if $u \to v$, then Rmin(u) = Lmin(v).*

We will first need some definitions. Given a set of strings S, we say that $(u, v) \in S^2$ are *compactable* in a set $V \subseteq S$ if $u \to v$ and, $\forall w \in V$, if $w \to v$ then $w = u$ and if $u \to w$ then $w = v$. The compaction operation is defined on a pair of compactable strings u, v in S. It replaces u and v by a single string $w = u \cdot v[k+1 \ldots |v|]$ where '$\cdot$' is the string concatenation operator. Two strings (u, v) are *m-compactable* in V if they are compactable in V and if $m = \text{Rmin}(u) = \text{Lmin}(v)$. The m-compaction of a set V is obtained from V by applying the compaction operation as much as possible in any order to all pairs of strings that are m-compactable in V. It is easy to show that the order in which strings are compacted does not lead to different m-compactions. Compaction is a useful notion because a simple way to obtain the simple paths is to greedily perform compaction as long as possible. In the following analysis, we identify a string u with the path $p = u[1 \ldots k] \to u[2 \ldots (k+1)] \to \ldots \to u[(|u| - k + 1) \ldots |u|]$ of all the k-mers of u in consecutive order.

We now give a high-level overview of Algorithm 1. The input is a set of k-mers S and a parameter $\ell < k$ which is the minimizer size. For each $m \in \Sigma^\ell$, we maintain a file F_m in external memory. Each file contains a set of strings, and we will later prove that at any point during the execution, each string is a sub-path of a simple path (Lemma 9 in the Appendix). Moreover, we show that at any point of execution, the multi-set of k-mers appearing in the strings and in the output does not change and is always S (Property 4 in the Appendix).

At line 5, we partition the k-mers into the files, according to their ℓ-minimizers. Next, each of the files is processed, starting from the file of the smallest minimizer in increasing order (line 6). For each file, we load the strings into memory and m-compact them (line 7), with the idea being that the size of each of the files is kept

[3] Source code available at `http://github.com/Malfoy/bcalm`

Algorithm 1 BCALM: Enumeration of all maximal simple paths in the dBG

1: **Input:** Set of k-mers S, minimizer size $\ell < k$
2: **Output:** Sequences of all simple paths in the de Bruijn graph of S
3: Perform a linear scan of S to get the frequency of all ℓ-mers (in memory)
4: Define the ordering of the minimizers, given by their frequency in S
5: Partition S into files F_m based on the minimizer m of each k-mer
6: **for** each file F_m in increasing order of m **do**
7: $C_m \leftarrow m$-compaction of F_m (performed in memory)
8: **for** each string u of C_m **do**
9: $B_{min} \leftarrow$ min(Lmin (u),Rmin (u))
10: $B_{max} \leftarrow$ max(Lmin (u),Rmin (u))
11: **if** $B_{min} \leq m$ and $B_{max} \leq m$ **then**
12: Output u
13: **else if** $B_{min} \leq m$ and $B_{max} > m$ **then**
14: Write u to $F_{B_{max}}$
15: **else if** $B_{min} > m$ and $B_{max} > m$ **then**
16: Write u to $F_{B_{min}}$
17: Delete F_m

small enough so that memory usage is low. The result of the compaction is a new set of strings, each of which is then either written to one of the files that has not been yet processed or output as a simple path.

The rule of choosing which file to write to is based on the left and right minimizers of the string. If both minimizers are no more than m, then the string is output as a simple path (line 12). Otherwise, we identify m', the smallest of the two minimizers that is bigger than m, and write the string to the file $F_{m'}$. Finally, the file F_m is discarded, and the next file is processed. The rule for placing the strings into the files ensures that as each file F_m is processed (line 6), it will contain every k-mer that has m as a minimizer (Lemma 8 in the Appendix). We can then use this to prove the correctness of the algorithm (proof in Appendix):

Theorem 3. *The output of Algorithm 1 is the set of maximal simple paths of the de Bruijn graph of S.*

There are several implementation details that make the algorithm practical. First, reverse complements are supported in the natural way by identifying each k-mer with its reverse complement and letting the minimizer be the smallest ℓ-mer in both of them. Second, we avoid creating 4^ℓ files, which may be more than the file system supports. Instead, we use virtual files and group them together into a smaller number of physical files. This allowed us to use $\ell = 10$ in our experiments. Third, when we load a file from disk (line 7) we only load the first and last k-mer of each string, since the middle part is never used by the compaction algorithm. We store the middle part in an auxiliary file and use a pointer to keep track of it within the strings in the F_m files.

Consequently, the algorithm memory usage depends on the number of strings in each file F_m, but not on the total size of those files. For a fixed input S, the number of strings in a file F_m depends on the minimizer length ℓ and the ordering of minimizers. When ℓ increases, the number of $(k-1)$-mers in S that have the same minimizer decreases. Thus, increasing ℓ yields less strings per file, which decreases the memory usage. We also realized that, when highly-repeated ℓ-mers are less likely to be chosen as minimizers, the sequences are more evenly distributed among files. We therefore perform in-memory ℓ-mer counting (line 3) to obtain a sorted frequency table of all ℓ-mers. This step requires an array of $64|\Sigma|^\ell$ bits to store the count of each ℓ-mer in 64 bits, which is negligible memory overhead for small values of ℓ (8 MB for $\ell = 10$). Each ℓ-mer is then mapped to its rank in the frequency array, to create a total ordering of minimizers (line 4). Our experiments showed a drastic improvement over lexicographic ordering (results in Section 9).

9 Results

We tested the effectiveness of our algorithms to assemble two sequencing datasets. Experiments in Tables 1, 2 and 3 were run on a single core of a desktop computer equipped with an Intel i7 3.2 GHz processor, 8 GB of memory and a 7200 RPM hard disk drive. Experiments in Tables 4 and 5 were run on a single core of a cluster node with 24 GB of memory and 2.67 GHz cores. In all experiments, at most 300 GB of temporary disk space was used. The first dataset is 36 million 155bp Illumina human chromosome 14 reads (2.9 GB compressed fastq) from the GAGE benchmark [34]. The second dataset is 1.4 billion Illumina 100bp reads (54 GB compressed fastq) from the NA18507 human genome (SRX016231). We first processed the reads with k-mer counting software, which is the first step of most assembly pipelines. We used a value of $k = 55$ as we found it gives reasonably good results

Dataset	DSK	BCALM	DBGFM
Chromosome 14	43 MB	19 MB	38 MB
	25 mins	15 mins	7 mins
Whole human genome	1.1 GB	43 MB	1.5 GB
	5 h	12 h	7 h

Table 1. Running times (wall-clock) and memory usage of DSK, BCALM and DBGFM construction on the human chromosome 14 and whole human genome datasets ($k = 55$ and $\ell = 10$ for both).

	DBGFM	Salikhov *et al.*	Conway & Bromage
chr14	38.0 MB	94.9 MB	> 875 MB
Full human dataset	1462 MB	2702 MB	> 22951 MB

Table 2. Memory usage of de Bruijn graph data structures, on the human chromosome 14 and whole human genome datasets ($k = 55$ for both). We did not run the algorithm of Conway and Bromage because our machine does not have sufficient memory for the whole genome. Instead, we report the theoretical size of their data structure, assuming that it would be constructed from the output of DSK. As described in [10], this gives a lower bound on how well their implementation could perform.

on both datasets. We used DSK [29], a software that is designed specifically for low memory usage and can also filter out low-count k-mers as they are likely due to sequencing errors (we used < 5 for chr14 and < 3 for the whole genome).

First, we ran BCALM on the of k-mers computed by DSK. The output of BCALM was then passed as input to DBGFM, which constructed the FM-index. Table 1 shows the resulting time and memory usage of DSK, BCALM, and DBGFM. For the whole genome dataset, BCALM used only 43 MB of memory to take a set of $2.5 \cdot 10^9$ 55-mers and output 40 million sequences of total length 4.6 Gbp. DBGFM represented these paths in an index of size 1.5 GB, using no more than that memory during construction. The overall construction time, including DSK, was roughly 24 hours. In comparison, a subset of this dataset was used to construct the data structure of Salikhov *et al.* in 30.7 hours [33].

We compared the space utilization of our DBGFM representation with that of other low space data structures, Salikhov *et al.* [33] and Conway and Bomage [10] (Table 2). Another promising approach is that of Bowe *et al.* [7], but they do not have an implementation available. We use 3.53 bits per k-mer (38.0 MB total) for chr14 and 4.76 bits per k-mer (1462 MB total) for the whole-genome. This is a 60% and 46% improvement over the state-of-the art, respectively.

During algorithm development, we experimented with different ways to order the minimizers and the effect on memory usage (Table 3). Initially, we used the lexicographical ordering, but experiments with the chromosome 14 dataset showed it was a poor choice, resulting in 804 MB memory usage with $\ell = 8$. The lexicographically smallest ℓ-mer is $m_0 = A^\ell$, which is overly abundant in human chromosomes for $\ell \leq 10$, resulting in a large file F_{m_0}. In a second attempt, we created a uniformly random ordering of all the ℓ-mers. While A^ℓ is no longer likely to have a small value, it is still likely that there is a highly repeated ℓ-mer that comes early in the ordering, resulting in 222 MB memory usage. Finally, we ordered the ℓ-mers according to their frequency in the dataset. This gave a memory usage of 19 MB, resulting in a 40-fold improvement over the initial lexicographical ordering. The running times of all three orderings were comparable. We also evaluated the effect that the minimizer size ℓ has on memory usage and running time (Table 4). Larger ℓ will generally lead to smaller memory usage, however we did not see much improvement past $\ell = 8$ on this dataset.

Finally, we evaluated the performance of ABySS using DBGFM compared with that of the hash table implementation (Table 5). Note, however, that only the graph traversal and marking steps were implemented in the DBGFM version, and none of the graph simplifications. The DBGFM version used 70% less memory, albeit the hash version was 33% faster, indicating the time/space trade-off inherent in the FM-index. In addition to storing

Ordering type	Lexicographical	Uniformly Random	ℓ-mer frequency
Memory usage	804 MB	222 MB	19 MB

Table 3. Memory usage of BCALM with three different minimizer orderings: lexicographical, uniformly random, and according to increasing ℓ-mer frequencies. The dataset used is the human chromosome 14 with $k = 55$ and $\ell = 8$.

Minimizer size ℓ	2	4	6	8	10
Memory usage	9879 MB	3413 MB	248 MB	19 MB	19 MB
Running time	27m19s	22m2s	20m5s	18m39s	21m4s

Table 4. Memory usage and wall-clock running time of BCALM with increasing values of minimizer sizes ℓ on the chr14 data. By grouping files into virtual files, these values of ℓ require respectively 4, 16, 64, 256 and 1024 physical files on disk. The ordering of minimizers used is the one based on ℓ-mer counts.

Data structure	Memory usage	Bytes/k-mer	dBG (B/k-mer)	Data (B/k-mer)	Overhead (B/k-mer)	Run time
sparsehash	2429 MB	29.50	16	8	5.50	14m4s
DBGFM	739 MB	8.98	0.53	8	0.44	21m1s

Table 5. Memory usage and run time (wall clock) of the ABySS hash table implementation (sparsehash) and of the DBGFM implementation, using a single thread to assemble the human chromosome 14 data set. The dBG bytes/k-mer column corresponds to the space taken by encoded k-mers for sparsehash, and the FM-index for DBGFM. The Data bytes/k-mer column corresponds to associated data. The Overhead bytes/k-mer corresponds to the hash table and heap overheads, as well as the rank/select bit array. The run time of the DBGFM row does not include the time to construct the DBGFM representation.

the graph, ABySS associates data with each k-mer: the count of each k-mer and its reverse complement (two 16 bits counters), the presence or absence of the four possible in- and out-edges (8 bits), three boolean flags indicating whether the k-mer and its reverse complement have been visited in graph traversal (2 bits), and whether they have been removed (1 bit). While in the hash implementation, the graph structure takes 54% of the memory, in the DBGFM version it only used 6% of memory. This indicates that further memory improvements can be made by optimizing the memory usage of the associated data.

10 Conclusion

This paper has focused on pushing the boundaries of memory efficiency of de Bruijn graphs. Because of the speed/memory trade-offs involved, this has come at the cost of slower data structure construction and query times. Our next focus will be on improving these runtimes through optimization and parallelization of our algorithms.

We see several benefits of low-memory de Bruijn graph data structures in genome assembly. First, there are genomes like the 20 Gbp white spruce which are an order of magnitude longer than the human which cannot be assembled by most assemblers, even on machines with a terabyte of RAM. Second, even for human sized genomes, the memory burden poses unnecessary costs to research biology labs. Finally, in assemblers such as ABySS that store the k-mers explicitly, memory constraints can prevent the use of large k values. With DBGFM, the memory usage becomes much less dependent on k, and allows the use of larger k values to improve the quality of the assembly.

Beyond genome assembly, our work is also relevant to many *de novo* sequencing applications where large de Bruijn graphs are used, e.g. assembly of transcriptomes and meta-genomes [16,5], and *de novo* genotyping [19].

Acknowledgements The authors would like to acknowledge anonymous referees from a previous submission for their helpful suggestions and for pointing us to the paper of Gagie [13].

References

1. Adjeroh, D., Bell, T.C., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer (2008)
2. Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S.K., Prjibelski, A.D., Pyshkin, A., Sirotkin, A., Vyahhi, N., Tesler, G., Alekseyev, M.A., Pevzner, P.A.: SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology 19(5), 455–477 (2012)
3. Birol, I., Raymond, A., Jackman, S.D., Pleasance, S., Coope, R., Taylor, G.A., Yuen, M.M.S., Keeling, C.I., Brand, D., Vandervalk, B.P., Kirk, H., Pandoh, P., Moore, R.A., Zhao, Y., Mungall, A.J., Jaquish, B., Yanchuk, A., Ritland, C., Boyle, B., Bousquet, J., Ritland, K., MacKay, J., Bohlmann, J., Jones, S.J.: Assembling the 20 Gb white spruce (Picea glauca) genome from whole-genome shotgun sequencing data. Bioinformatics (2013)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13(7), 422–426 (1970)

5. Boisvert, S., Raymond, F., Godzaridis, É., Laviolette, F., Corbeil, J., et al.: Ray Meta: scalable de novo metagenome assembly and profiling. Genome Biology 13(12), R122 (2012)
6. Bowe, A.: Succinct de Bruijn graphs (blog post). http://alexbowe.com/succinct-debruijn-graphs/, accessed: 2013-10-18
7. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: WABI. Lecture Notes in Computer Science, vol. 7534, pp. 225–235. Springer (2012)
8. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical report 124. Tech. rep., Palo Alto, CA: Digital Equipment Corporation (1994)
9. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In: WABI. Lecture Notes in Computer Science, vol. 7534, pp. 236–248. Springer (2012)
10. Conway, T.C., Bromage, A.J.: Succinct data structures for assembling large genomes. Bioinformatics 27(4), 479 (2011)
11. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM 57(1) (2009)
12. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. pp. 390–398. IEEE (2000)
13. Gagie, T.: Bounds from a card trick. Journal of Discrete Algorithms 10, 2–4 (2012)
14. Gnerre, S., MacCallum, I., Przybylski, D., Ribeiro, F.J., Burton, J.N., Walker, B.J., Sharpe, T., Hall, G., Shea, T.P., Sykes, S.: High-quality draft assemblies of mammalian genomes from massively parallel sequence data. Proceedings of the National Academy of Sciences 108(4), 1513 (2011)
15. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)(Greece, 2005). pp. 27–38 (2005)
16. Grabherr, M.G., Haas, B.J., Yassour, M., Levin, J.Z., Thompson, D.A., Amit, I., Adiconis, X., Fan, L., Raychowdhury, R., Zeng, Q., et al.: Full-length transcriptome assembly from RNA-Seq data without a reference genome. Nature Biotechnology 29(7), 644–652 (2011)
17. Haussler, D., O'Brien, S.J., Ryder, O.A., Barker, F.K., Clamp, M., Crawford, A.J., Hanner, R., Hanotte, O., Johnson, W.E., McGuire, J.A., et al.: Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. Journal of Heredity 100(6), 659–674 (2008)
18. Idury, R.M., Waterman, M.S.: A new algorithm for DNA sequence assembly. Journal of Computational Biology 2(2), 291–306 (1995)
19. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nature Genetics 44(2), 226–232 (2012)
20. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature Methods 9(4), 357–359 (2012)
21. Li, H.: Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics 28(14), 1838–1844 (2012)
22. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics 25(14), 1754–1760 (2009)
23. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K.: De novo assembly of human genomes with massively parallel short read sequencing. Genome Research 20(2), 265 (2010)
24. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009)
25. Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., Suri, S.: Memory efficient minimum substring partitioning. In: Proceedings of the 39th international conference on Very Large Data Bases. pp. 169–180. VLDB Endowment (2013)
26. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics 95(6), 315–327 (2010)
27. Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J.M., Brown, C.T.: Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. Proceedings of the National Academy of Sciences 109(33), 13272–13277 (2012)
28. Pevzner, P.A.: l-Tuple DNA sequencing: computer analysis. Journal of Biomolecular Structure & Dynamics 7(1), 63–73 (Aug 1989)
29. Rizk, G., Lavenier, D., Chikhi, R.: DSK: k-mer counting with very low memory usage. Bioinformatics 29(5), 652–653 (2013)
30. Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., Yorke, J.A.: Reducing storage requirements for biological sequence comparison. Bioinformatics 20(18), 3363–3369 (2004)
31. Roberts, M., Hunt, B.R., Yorke, J.A., Bolanos, R.A., Delcher, A.L.: A preprocessor for shotgun assembly of large genomes. Journal of Computational Biology 11(4), 734–752 (2004)
32. Rødland, E.A.: Compact representation of k-mer de bruijn graphs for genome read assembly. BMC Bioinformatics 14(1), 313 (2013)
33. Salikhov, K., Sacomoto, G., Kucherov, G.: Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In: Darling, A., Stoye, J. (eds.) Algorithms in Bioinformatics, Lecture Notes in Computer Science, vol. 8126, pp. 364–376. Springer Berlin Heidelberg (2013)
34. Salzberg, S.L., Phillippy, A.M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T.J., Schatz, M.C., Delcher, A.L., Roberts, M., et al.: GAGE: A critical evaluation of genome assemblies and assembly algorithms. Genome Research 22(3), 557–567 (2012)

35. Simpson, J.T.: Exploring genome characteristics and sequence quality without a reference. arXiv preprint arXiv:1307.8026 (2013)
36. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics 26(12), 367–373 (2010)
37. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Research 22(3), 549–556 (2012)
38. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J., Birol, İ.: ABySS: a parallel assembler for short read sequence data. Genome Research 19(6), 1117–1123 (2009)
39. Sondow, J., Stong, R.: Choice bounds: 11132. The American Mathematical Monthly 114(4), pp. 359–360 (2007)
40. Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., Douglas, W.Y.: Exploiting sparseness in de novo genome assembly. BMC Bioinformatics 13(Suppl 6), S1 (2012)
41. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Research 18(5), 821–829 (2008)

11 Appendix

This Appendix contains lemmas and proofs that are omitted in the main text.

11.1 Lower bound for general de Bruijn graphs

Lemma 1. *Let $y \in T$. There exists a unique $0 \leq i \leq \ell$ such that $y \in T_i$.*

Proof. Take two arbitrary levels $i_1 < i_2$ and two arbitrary vertices in those levels, $x_1 \in T_{i_1}$ and $x_2 \in T_{i_2}$. Let $z \in \{1, 2\}$. The k-mer represented by x_z is "$\mathrm{A}^{\ell - i_z + 1}\mathrm{T}\alpha_z$", where α_z is some string. At position $\ell - i_1 + 1$, x_1 has a T, while x_2 has an A. Therefore, $x_1 \neq x_2$ and the lemma follows. □

Lemma 2. *For all distinct x_1 and x_2 in T that are not in the last level (T_ℓ), $\overrightarrow{ext}(x_1) \cap \overrightarrow{ext}(x_2) = \emptyset$.*

Proof. By Lemma 1, there exist unique levels i_1 and i_2 such that $x_1 \in T_{i_1}$ and $x_2 \in T_{i_2}$. We first observe that $\overrightarrow{ext}(x_z) \in T_{i_z+1}$, for $z \in \{1, 2\}$. If it is the case that $i_1 \neq i_2$, then Lemma 1 applied to the vertices in the extensions prove the lemma. Now suppose that $i_1 = i_2$, and we write $i = i_1$. Then, for $z \in \{1, 2\}$, the k-mer represented by x_z is "$\mathrm{A}^{\ell - i}\mathrm{T}\alpha_z$", where α_z is a $(\ell + i - 1)$-mer and $\alpha_1 \neq \alpha_2$. We can then write the extensions as $\overrightarrow{ext}(x_z) = \{\text{"}\mathrm{A}^{\ell - i - 1}\mathrm{T}\alpha_z\beta\text{"} \ : \ \beta \in \{A, C, G, T\}\}$. Because $\alpha_1 \neq \alpha_2$, the sets $\overrightarrow{ext}(x_1)$ and $\overrightarrow{ext}(x_2)$ share no common elements. □

Property 1. *For all $x \in S^k$, $|x| = 4^{\ell-1}(\ell + 1)$.*

Proof. Follows directly from Lemmas 1 and 2. □

Lemma 3. *Let $X = X_1, \ldots, X_\ell$ and $Y = Y_1, \ldots, Y_\ell$ be two sequences of index sets. Then $s(X) = s(Y)$ if and only if $X = Y$.*

Proof. Since the construction is fully deterministic and depends only on the index sets, then $X = Y$ immediately implies $s(X) = s(Y)$. For the other direction, suppose that $X \neq Y$. Let $i > 0$ be the smallest index such that $X_i \neq Y_i$. Then there exists a vertex y such that $y \in L_i^X$ but $y \notin L_i^Y$. Since y is in T_i but not in L_i^Y, Lemma 1 implies that $y \notin s(Y)$. □

Lemma 4. *For all $m > 0$, $\binom{4m}{m} \geq 2^{(c-\epsilon_0)m}$, where $c = 8 - 3\lg 3$ and $\epsilon_0 = \lg(12m)/m$.*

Proof. Follows directly from an inequality of Sondow and Stong [39]: $\binom{rm}{m} > \frac{2^{cm}}{4m(r-1)}$. □

Property 2. *$|S^k| = \binom{4m}{m}^\ell \geq 2^{(c-\epsilon_0)\ell m}$, where $c = 8 - 3\lg 3$ and $\epsilon_0 = \lg(12m)/m$.*

Proof. Lemma 3 tells us that the size of S^k is the number of possible ways one could choose $X_1, \ldots, X_\ell$ during the construction of each element $s(X_1, \ldots, X_\ell)$. The choice for each X_i is independent, and there are $\binom{4m}{m}$ possibilities. Hence, there are $\binom{4m}{m}^\ell$ total choices. The inequality follows from Lemma 4. □

Property 3. *Let $x = s(X) \in S^k$ and $y = s(Y) \in S^k$ be two distinct elements in S^k. Then, there exists a k-mer $u \in x \cap y$ such that $\overrightarrow{ext}(u) \cap x \neq \overrightarrow{ext}(u) \cap y$.*

Proof. By Lemma 3, $X \neq Y$. Let i be the smallest index such that $X_i \neq Y_i$, and let v be an element in L_i^X but not in L_i^Y. By construction, there exists a vertex $u \in L_{i-1}^X$ (and hence in L_{i-1}^Y) such that $v \in \overrightarrow{ext}(u)$. Lemma 1 tells us that v is not in y and hence u satisfies the condition of the lemma. □

Theorem 1. *Consider an arbitrary NDS and let* CONST *be its constructor. For any* $0 < \epsilon < 1$*, there exists a* k *and* $x \subseteq \Sigma^k$ *such that* $|\text{CONST}(x)| \geq |x| \cdot (c - \epsilon)$*, where* $c = 8 - 3\lg 3 \approx 3.25$.

Proof. Assume for the sake of contradiction that for all x, $|\text{CONST}(x)| < |x|(c-\epsilon)$. Let k be a large enough integer such that $k > 2c\epsilon^{-1}$ and $\epsilon_0 < (\epsilon(\ell+1) - c)/\ell$ holds (with m, ℓ, ϵ_0 as defined above). The second inequality is verified for any large value of k, since $\epsilon_0 = \Theta(\ell/4^\ell)$ converges to 0 and $(\epsilon(\ell+1) - c)/\ell$ converges to ϵ. Let $n = 4^{k/2-1}(k/2+1)$. Consider the outputs of CONST on the elements of S^k. When the input is constrained to be of size n, the output must use less than $(c-\epsilon)n$ bits (by Lemma 1). Hence the range of CONST over the domain S^k has size less than $2^{(c-\epsilon)n}$. At the same time, Lemma 2 states that there are at least $2^{(c-\epsilon_o)\ell m}$ elements in S^k.

From the inequality $\epsilon_0 < (\epsilon(\ell+1)-c)/\ell$ we derive that $(c-\epsilon_0)\ell > (c-\epsilon)(\ell+1)$ and thus $2^{(c-\epsilon_o)\ell m} > 2^{(c-\epsilon)n}$. Therefore, there must exist distinct $s_1, s_2 \in S^k$ such that $\text{CONST}(s_1) = \text{CONST}(s_2)$. We can now apply Lemma 3 to obtain an element $y \in s_1 \cap s_2$ that is a valid input to $\text{CONST}(s_1)$ and to $\text{CONST}(s_2)$. Since the two functions are the same, the return value must also the same. However, we know that the out-neighborhoods of y are different in s_1 and in s_2, hence, one of the results of NBR on y must be incorrect. This contradicts the correctness of CONST. □

11.2 Lower bound for linear de Bruijn graphs

Lemma 5. *The number of DNA strings of length* m *where each* k*-mer is seen only once is at least* $4^m(1-\binom{m}{2}4^{-k})$.

Proof. This Lemma was expressed in a probabilistic setting in [13], but we provide a deterministic proof here. We define a set of strings S and show that it contains all strings with at least one repeated k-mer. Let $\overline{s}^k$ be the string obtained by repeating the pattern s as many times as needed to obtain a string of length exactly k, possibly truncating the last occurrence.

$$S = \{s \mid \exists\,(i,j), 1 \leq i < j \leq m, \exists\, t, |t| = (m-k), |s| = m$$
$$s[1\ldots j] = t[1\ldots j], s[j+k+1\ldots m] = t[j+1\ldots m-k], s[j+1\ldots j+k] = \overline{s[i\ldots j]}^k\}$$

Let s' be a string which contains at least one repeated k-mer. Without loss of generality, assume that $i < j$ are two starting positions of identical k-mers ($s'[j\ldots j+k-1] = s'[i\ldots i+k-1]$). Setting t to be the concatenation of $s'[1..j]$ and $s'[j+k+1\ldots n]$, it is clear that s' is in S. Thus S contains all strings of length n having at least one repeated k-mer. Since there are $\binom{m}{2}$ choices for (i,j) and 4^{m-k} choices for t, the cardinality of S is at most $\binom{m}{2}4^{m-k}$, which yields the result. □

Lemma 6. *Given* $0 < \epsilon < 1$*, let* $n = \lceil 3\epsilon^{-1} \rceil$ *and* $k = \lceil 1 + (2+\epsilon)\log_4(2n) \rceil$*. The number of DNA strings of length* $(n+k-1)$ *which start with the same* k*-mer, and do not contain any repeated* $(k-1)$*-mer, is strictly greater than* $4^{n(1-\epsilon)}$.

Proof. Note that $k < n$, thus $k > (1+(2+\epsilon)\log_4(n+k-1))$ and $4^{-k+1} < (n+k-1)^{(-2-\epsilon)}$. Using Lemma 5, there are at least $(4^{n+k-1}(1-\binom{n+k-1}{2}4^{-k+1})) > (4^{n+k-1}(1-\frac{1}{2(n+k-1)^\epsilon}))$ strings of length $(n+k-1)$ where each $(k-1)$-mer is unique. Thus, each string has exactly n k-mers that are all distinct. By binning these strings with respect to their first k-mer, there exists a k-mer k_0 such that there are at least $4^{n-1}(1-\frac{1}{2(n+k-1)^\epsilon})$ strings starting with k_0, which do not contain any repeated $(k-1)$-mer. The following inequalities hold: $4^{-1} > 4^{-n\epsilon/2}$ and $(1-\frac{1}{2(n+k-1)^\epsilon}) > \frac{1}{2} > 4^{-n\epsilon/2}$. Thus, $4^{n-1}(1-\frac{1}{2(n+k-1)^\epsilon}) > 4^{n(1-\epsilon)}$.

□

Lemma 7. *Two different strings of length* $(n+k-1)$ *starting with the same* k*-mer and not containing any repeated* $(k-1)$*-mer correspond to two different linear de Bruijn graphs.*

Proof. For two different strings s_1 and s_2 of length $(n+k-1)$, which start with the same k-mer and do not contain any repeated $(k-1)$-mer, observe that their sets of k-mers cannot be identical. Suppose they were, and consider the smallest integer i such that $s_1[i\ldots i+k-2] = s_2[i\ldots i+k-2]$ and $s_1[i-k+1] \neq s_2[i-k+1]$. The k-mer $s_1[i\ldots i+k-1]$ appears in s_2, at some position $j \neq i$. Then $s_2[i\ldots i+k-2]$ and $s_2[j\ldots j+k-2]$ are identical $(k-1)$-mers in s_2, which is a contradiction. Thus, s_1 and s_2 correspond to different sets of k-mers, and therefore correspond to two different linear de Bruijn graphs. □

Theorem 2. *Consider an arbitrary NDS for linear de Bruijn graphs and let* CONST *be its constructor. Then, for any $0 < \epsilon < 1$, there exists (n, k) and a set of k-mers S of cardinality n, such that $|\text{CONST}(S)| \geq 2n(1-\epsilon)$.*

Proof. Assume for the sake of contradiction that for all linear de Bruijn graphs, the output of CONST requires less than $2(1-\epsilon)$ bits per k-mer. Thus for a fixed k-mer length, the number of outputs $\text{CONST}(S)$ for sets of k-mers S of size n is no more than $2^{2n(1-\epsilon)}$. Lemma 6 provides values (k, n, k_0), for which there are more strings starting with a k-mer k_0 and containing exactly n k-mers with no duplicate $(k-1)$-mers (strictly more than $2^{2n(1-\epsilon)}$) than the number of outputs $\text{CONST}(S)$ for n k-mers.

By the pigeonhole principle, there exists a navigational data structure constructor $\text{CONST}(S)$ that takes the same values on two different strings s_1 and s_2 that start with the same k-mer k_0 and do not contain repeated $(k-1)$-mer. By Lemma 7, $\text{CONST}(S)$ takes the same values on two different sets of k-mers S_1 and S_2 of cardinality n. Let p be the length of longest prefix common to both strings. Let k_1 be the k-mer at position $(p-k+1)$ in s_1. Note that k_1 is also the k-mer that starts at position $(p-k+1)$ in s_2. By construction of s_1 and s_2, k_1 does not appear anywhere else in s_1 or s_2. Moreover, the k-mer at position $(p-k)$ in s_1 is different to the k-mer at position $(p-k)$ in s_2. In a linear de Bruijn graph corresponding to a string where no $(k-1)$-mer is repeated, each node has at most one out-neighbor. Thus, the out-neighbor of k_1 in the de Bruijn graph of S_1 is different to the out-neighbor of k_1 in the de Bruijn graph of S_2, i.e. $\text{NBR}(\text{CONST}(S_1), k_1) \neq \text{NBR}(\text{CONST}(S_2), k_1)$, which is a contradiction. □

11.3 Algorithm to enumerate the simple paths of a de Bruijn graph in low memory

Property 4. *At any point of execution after line 5, the multi set of k-mers present in the files and in the output is S.*

Proof. We prove by induction. It is trivially true after the partition step. In general, note that the compaction operation preserves the multi set of k-mers. Because the only way the strings are ever changed is through compaction, the property follows. □

Lemma 8. *For each minimizer m, for each k-mer u in S such that Lmin$(u) = m$ (resp. Rmin$(u) = m$), u is the left-most (resp. right-most) k-mer of a string in F_m at the time F_m is processed.*

Proof. We prove this by induction on m. Let m_0 be the smallest minimizer. All k-mers that have m_0 as a left or right minimizer are strings in F_{m_0}, thus the base case is true. Let m be a minimizer and u be a k-mer such that $\text{Lmin}(u) = m$ or $\text{Rmin}(u) = m$, and assume that the induction hypothesis holds for all smaller minimizers. If $\min(\text{Lmin}(u), \text{Rmin}(u)) = m$, then u is a string in F_m after execution of line 5. Else, without loss of generality, assume that $m = \text{Rmin}(u) > \text{Lmin}(u)$. Then, after line 5, u is a string in $F_{\text{Lmin}(u)}$. Let $F_{m^1}, \ldots, F_{m^t}$ be all the files, in increasing order of the minimizers, which have a simple path containing u before the maximal-length simple path containing u is outputted by the algorithm. Let m^i be the largest of these minimizers strictly smaller than m. By the induction hypothesis and Property 4, u is at the right extremity of a unique string s_u in F_{m^i}. After the m^i-compactions, since $m = \text{Rmin}(s_u) > m^i$, s_u does not go into the output. It is thus written to the next larger minimizer. Since $m = \text{Rmin}(u) \leq m^{i+1}$, then it must be that $m^{i+1} = m$, and s_u is written to F_m, which completes the induction. □

Lemma 9. *In Algorithm 1, at any point during execution, each string in F_m corresponds to a sub-path of a maximal-length simple path.*

Proof. We say that a string is *correct* if it corresponds to a sub-path of a maximal-length simple path. We prove the following invariant inductively: at the beginning of the loop at line 6, all the files F_m contain correct strings. The base case is trivially true as all files contain only k-mers in the beginning. Assume that the invariant holds before processing F_m. It suffices to show that no wrong compactions are made; i.e. if two strings from F_m are m-compactable, then they are also compactable in S. The contrapositive is proven. Assume, for the sake of obtaining a contradiction, that two strings (u, v) are not compactable in S, yet are m-compactable in F_m at the time it is processed. Without loss of generality, assume that there exists $w \in S$ such that $u \to w$ and $w \neq v$. Since $u \to v$ and $u \to w$, $m = \text{Rmin}(u) = \text{Lmin}(v) = \text{Lmin}(w)$. Hence, by Lemma 8, w is the left-most k-mer of a string in F_m at the time F_m is processed. This contradicts that (u, v) are m-compactable in F_m at the time it is processed. Thus, all compactions of strings in F_m yield correct strings, and the invariant remains true after F_m is processed. □

Theorem 3. *The output of Algorithm 1 is the set of maximal simple paths of the de Bruijn graph of S.*

Proof. By contradiction, assume that there exists a maximal-length simple path p that is not returned by Algorithm 1. Every input k-mer is returned by Algorithm 1 in some string, and by Lemma 9, every returned string corresponds to a sub-path of a maximal-length simple path. Then, without loss of generality, assume that a simple path of p is split into sub-paths $p_1, p_2, \ldots, p_i$ in the output of Algorithm 1. Let u be the last k-mer of p_1 and v be the first k-mer of p_2. Let $m = \text{Rmin}(u) = \text{Lmin}(v)$ (with Observation 1). By Lemma 8, u and v are both present in F_m when it is processed. As u and v are compactable in S (to form p), they are also compactable in F_m and thus the strings that include u and v in F_m are compacted at line 7. This indicates that u and v cannot be extremities of p_1 and p_2, which yields a contradiction. □