Variable-Order de Bruijn Graphs

Alex Bowe, Christina Boucher, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane

Abstract—The de Bruijn graph G_K of a set of strings S is a key data structure in genome assembly that represents overlaps between all the K-length substrings of S. Construction and navigation of the graph is a space and time bottleneck in practice and the main hurdle for assembling large genomes. This problem is compounded because state-of-the-art assemblers do not build the de Bruijn graph for a single order (value of K) but for multiple values of K: they build d de Bruijn graphs, each with a specific order, i.e., $G_{K_1}, G_{K_2}, \ldots, G_{K_d}$. Although, this paradigm increases the quality of the assembly produce but it greatly increases runtime, because of the need to construct d graphs instead of one. In this paper, we show how to augment a succinct de Bruijn graph representation by Bowe et al. (Proc. WABI, 2012) to support new operations that let us change order on the fly, effectively representing all de Bruijn graphs up to some maximum order K in a single data structure. Our experiments show our variable-order de Bruijn graph only modestly increases space usage, construction time, and navigation time compared to a single order graph.

1 Introduction

Accurate assembly of genomes is a fundamental problem in bioinformatics and is vital to several ambitious scientific projects, including the 10,000 vertebrate genomes (Genome 10K) [1], *Arabidopsis* variations (1001 genomes) [2], human variations (1000 genomes) [3], and the Human Microbiome [4] projects. The genome assembly process builds long contiguous DNA sequences, called *contigs*, from shorter DNA fragments, called *reads*, typically 100-150 (DNA) symbols in length.

In Eulerian sequence assembly [5], [6], a *de Bruijn graph* is constructed with a vertex v for every K-mer (substring of length K) present in an input set of reads, and an edge (v, v') for every observed $(K+1)$-mer in the reads with K-mer prefix v and K-mer suffix v'. Contigs are then extracted from this graph. Most state-of-the-art assemblers use this paradigm [7], [8], [9], [10], [11], and follow the same general outline: extract $(K+1)$-mers from the reads; construct the de Bruijn graph on the set of $(K+1)$-mers; simplify the graph; and construct the contigs (simple paths in the graph). The value of K can be, and is often required to be, specified by the user.

Determining an appropriate value of K is important and has a direct impact on assembly quality. Stated very briefly, when K is too small the resulting graph is complicated by spurious edges and nodes, and when K is too large the graph becomes too sparse and possibly disconnected.

In an attempt to circumvent the need to choose a single, ideal value of K, SPAdes [7] and IDBA [8] use a number of different K values. IDBA [8] builds a number of de Bruijn graphs for each a fixed set of K values. At a given iteration of the algorithm, the de Bruin graph for the current value of K is built from the reads and the contigs for that graph are constructed, then all the reads that align to at least one of those contigs are removed from the current set of reads. In the next iteration the graph is built by converting every edge from the previous graph to a vertex while treating contigs as edges. SPAdes [7] uses a similar approach but uses all the reads at each iteration.

- *A. Bowe is with the Department of Informatics, National Institute of Informatics, Japan. A. Bowe is a correponding author. E-mail: alex@nii.ac.jp*
- *C. Boucher is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA. C. Boucher is a correponding author. E-mail: christina.boucher@colostate.edu*
- *T. Gagie, S.J. Puglisi are with the Department of Computer Science, University of Helsinki, Finland.*
- *K. Sadakane is with the Department of Mathematical Informatics, University of Tokyo, Japan.*

1.1 Our Contribution

SPAdes [7] and IDBA [8] represent the state-of-the-art for genome assemblers, producing assemblies of greatly improved quality compared to previous approaches. However, their need to construct several de Bruijn graphs of different orders over the assembly process makes them extremely slow on large genomes.

In this paper we address this problem by describing a succinct data structure that, for a given K, efficiently represents *all* the de Bruijn graphs for $k \leq K$ and allows navigation within and between each graph. In addition also describe an alternative representation which is smaller but slower.

We have implemented the faster version of our data structure and shown that in practice it requires around 3.5 times the space of a graph for a single K, and incurs a modest slow down in construction time and on navigation operations. Compared with the conference version of this paper [12], we have implemented an external memory construction algorithm, and demonstrated the scalability of our structure on much larger data sets.

1.2 Related Work

There are several succinct data structures for the de Bruijn graph of a single order (i.e. value of K). One of the first approaches was introduced by Simpson et al. [10] as part of the development of the ABySS assembler. Their method stores the graph as a distributed hash table and thus requires 336 GB to store the graph corresponding to a set of reads from a human genome (HapMap: NA18507). In 2011, Conway and Bromage [13] reduced space requirements by using a sparse bitvector (by Okanohara and Sadakane [14]) to represent the

$(K+1)$-mers (the edges), and used rank and select operations (to be described shortly) to traverse it. As a result, their representation took 32 GB for the same data set. Minia, by Chikhi and Rizk [15], uses a Bloom filter to store edges. They traverse the graph by generating all possible outgoing edges at each node and testing their membership in the Bloom filter. Using this approach, the graph was reduced to 5.7 GB on the same dataset. Contemporaneously, Bowe, Onodera, Sadakane and Shibuya [16] developed a different succinct data structure based on the Burrows-Wheeler transform [17] that requires 2.5 GB. Their representation, which henceforth we refer to as BOSS from the authors' initials, is a starting point for our methods and we will discuss it in detail below. Very recently Chikhi et al. [18] implemented the de Bruijn graph using an FM-index and *minimizers*. Their method uses 1.5 GB on the same NA18507 data.

The data structure of Bowe et al. [16] is combined with ideas from IDBA-UD [19] in a metagenomics assembler called MEGAHIT [20]. In practice MEGAHIT requires more memory than competing methods but produces significantly better assemblies.

Lastly, Lin and Pevzner [21] recently introduced the *manifold de Bruijn graph*, which associates arbitrary substrings with nodes (the substrings are fixed during preprocessing), rather than K-mers. Lin and Pevzner's structure is mainly of theoretical interest since it has not yet been implemented.

1.3 Roadmap

Section 2 sets notation, and formally lays down the problem and auxiliary data structures we use. Section 3 gives details of the BOSS representation [16]. Sections 4 and 5 then describes our variable-order de Bruijn graph structure. In Section 6 we report on experiments comparing the practical performance of our data structure to that of a single-order de Bruijn graph. Section 7 offers directions for future work.

2 Preliminaries

2.1 De Bruijn Graphs

Given an alphabet Σ of σ symbols and a set of strings $\{S_1, S_2, \ldots, S_t\}$, $S_i \in \Sigma^+$, the *de Bruijn graph* of order K, denoted G_K^S, or just G_K, when the context is clear, is a directed, labelled graph defined as follows.

Let M_K be the set of distinct K-mers (strings of length K) that occur as substrings of some S_i. M_{K+1} is defined similarly. G_K has exactly $|M_K|$ nodes and with each node u we associate a distinct K-mer from M_K, denoted $\mathsf{label}(u)$. Edges are defined by M_{K+1}: for each string $T \in M_{K+1}$ there is a directed edge, labelled with symbol $T[K+1]$, from node u to node v, where $\mathsf{label}(u) = T[1,K]$ and $\mathsf{label}(v) = T[2,K+1]$.

2.2 Rank and Select

Two basic operations used in almost every succinct and compressed data structure are *rank* and *select*. Given a sequence (string) $S[1,n]$ over an alphabet $\Sigma = \{1,\ldots,\sigma\}$, a character $c \in \Sigma$, and integers i,j, $\mathsf{rank}_c(S,i)$ is the number of times that c appears in $S[1,i]$, and $\mathsf{select}_c(S,j)$ is the position of the j-th occurrence of c in S. For a binary string $B[1,n]$, the classic solution for rank and select [22] is built upon the input sequence, requiring $o(n)$ additional bits. Generally, rank_1 and select_1 are considered the default rank and select queries. More advanced solutions (e.g. [14]) achieve zero-order compression of B, representing it in just $nH_0(B) + o(n)$ bits of space, and supporting rank and select operations in constant time.

2.3 Wavelet Trees

To support rank and select on larger alphabet strings, the wavelet tree [23], [24] is a commonly used data structure that occupies $n \log \sigma + o(n \log \sigma)$ bits of space and supports rank and select queries in $\mathcal{O}(\log \sigma)$ time. Wavelet trees also support a variety of more complex queries on the underlying string (see, e.g. [25]), in $\mathcal{O}(\log \sigma)$ time, and we will make use of some of this functionality in Section 5.

3 BOSS Representation

Conceptually, to build the BOSS representation [16] of a Kth-order de Bruijn graph from a set of $(K+1)$-mers, we first add enough dummy $(K+1)$-mers starting with \$s so that if αa is in the set, then some $(K+1)$-mer ends with α (α a K-mer, a a symbol). We also add enough dummy $(K+1)$-mers ending with \$ that if $b\alpha$ is in the set, with α containing no \$ symbols, then some $(K+1)$-mer starts with α. We then sort the set of $(K+1)$-mers into the right-to-left lexicographic order of their first K symbols (with ties broken by the last symbol) to obtain a matrix. If the ith through jth $(K+1)$-mers start with α, then we say node $[i,j]$ in the graph has label α, with $j-i+1$ outgoing edges labelled with the last symbols of the ith through jth $(K+1)$-mers. If there are n nodes in the graph, then there are at most σn rows in the matrix, i.e., $(K+1)$-mers.

For example, if $K=3$ and the matrix is the one from Bowe et al.'s paper, shown in the left of Fig. 1, then the $n=11$ nodes are

$$[1,1], [2,2], [3,3], [4,5], [6,6], [7,7], [8,9], [10,10], [11,11], [12,12], [13,13]$$

with labels

$$\$\$\$, \mathrm{CGA}, \$\mathrm{TA}, \mathrm{GAC}, \mathrm{TAC}, \mathrm{GTC}, \mathrm{ACG}, \mathrm{TCG}, \$\$\mathrm{T}, \mathrm{ACT}, \mathrm{CGT},$$

respectively. The 3rd-order de Bruijn graph itself is shown in the right of the figure.

Bowe et al. described a number of queries on the graph, all of which can be implemented in terms of the following three with at most an $\mathcal{O}(\sigma)$-factor slowdown:

- $\mathsf{forward}(v,a)$ returns the node w reached from v by an edge labelled a, or NULL if there is no such node;
- $\mathsf{backward}(v)$ lists the nodes u with an edge from u to v;
- $\mathsf{lastchar}(v)$ returns the last character of v's label.

In our example, $\mathsf{forward}([8,9],\mathrm{A}) = [2,2]$, $\mathsf{backward}([2,2]) = [8,9],[10,10]$ and $\mathsf{lastchar}([8,9]) = \mathrm{G}$. Since $\mathsf{backward}$ always returns at least one node, we can recover any non-dummy node's entire label by K calls to $\mathsf{lastchar}$ interleaved with $K-1$ calls to $\mathsf{backward}$.

1)	\$\$\$	T
2)	CGA	C
3)	\$TA	C
4)	GAC	G
5)	GAC	T
6)	TAC	G
7)	GTC	G
8)	ACG	A
9)	ACG	T
10)	TCG	A
11)	\$\$T	A
12)	ACT	\$
13)	CGT	C

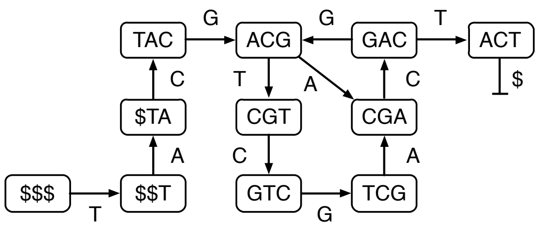

Fig. 1. The BOSS matrix (left) and de Bruijn graph (right) for the quadruples CGAC, GACG, GACT, TACG, GTCG, ACGA, ACGT, TCGA, CGTC.

4 Varying Order

If we delete the first column of the matrix in Figure 1, the result is *almost* the BOSS matrix for a 2nd-order de Bruijn graph whose nodes

$$[1,1],[2,2],[3,3],[4,6],[7,7],[8,10],[11,11],[12,12],[13,13]$$

have labels

$$\$\$,\mathrm{GA},\mathrm{TA},\mathrm{AC},\mathrm{TC},\mathrm{CG},\$\mathrm{T},\mathrm{CT},\mathrm{GT}\,,$$

respectively. Similarly, if we delete the first two columns of the original matrix, the result is almost the BOSS matrix for a 1st-order graph whose nodes

$$[1,1],[2,3],[4,7],[8,10],[11,13]$$

have labels

$$\$,\mathrm{A},\mathrm{C},\mathrm{G},\mathrm{T}\,,$$

respectively. If we delete the first three columns, the result is almost the BOSS graph for the 0th-order graph whose single node $[1,13]$ has an empty label. Notice we allow the same node to appear in different graphs, with labels of different lengths. If readers find this confusing, they can imagine that nodes are triples instead of pairs, with the additional component storing the label's length.

The truncated form of a higher order BOSS differs from the BOSS of a lower order in that some rows are repeated, which could prevent the BOSS representation from working properly. Suppose that, instead of trying to apply forward, backward and lastchar directly to nodes in the new graphs, we augment the BOSS representation of the original graph to support the following three queries:

- shorter(v,k) returns the node whose label is the last k characters of v's label;
- longer(v,k) lists nodes whose labels have length $k \leq K$ and end with v's label;
- maxlen(v,a) returns some node in the original graph whose label ends with v's label, and that has an outgoing edge labelled a, or NULL otherwise.

If we want a node in the original graph whose label ends with v's label but we do not care about its outgoing edges, then we write maxlen$(v,*)$. Notice shorter and longer are symmetric, in the sense that if v's label has length k_v and $x \in$ longer(v,k_v), then shorter$(x,k_v) = v$. In our example, shorter$([4,5],2) = [4,6]$ while longer$([4,6],3) = [4,5],[6,6]$ and maxlen$([4,6],\mathrm{G})$ could return either $[4,5]$ or $[6,6]$, while maxlen$([4,6],\mathrm{T}) = [4,5]$ and maxlen$([4,6],\mathrm{A}) = \mathrm{NULL}$.

If v is a node in the original graph — e.g., v is returned by maxlen — then we can use the BOSS implementations of forward, backward and lastchar. Otherwise, if v's label has length k_v then

$$\begin{aligned}\mathsf{forward}(v,a) &= \mathsf{shorter}(\mathsf{forward}(\mathsf{maxlen}(v,a),a),k_v)\\ \mathsf{lastchar}(v) &= \mathsf{lastchar}(\mathsf{maxlen}(v,*))\,.\end{aligned}$$

Assuming queries can be applied to lists of nodes, we can compute backward(v) as

$$\mathsf{shorter}(\mathsf{backward}(\mathsf{maxlen}(\mathsf{longer}(v,k_v+1),*)),k_v),$$

removing any duplicates.

To see why we can compute backward like this, suppose v's label is αa, so longer(v,k_v+1) returns a list of all $d \leq \sigma$ nodes whose labels have the form $b\alpha a$. Applying maxlen to this list returns a second list of d nodes, with labels $\beta_1 b_1 \alpha a,\ldots,\beta_d b_d \alpha a$ of length K. Applying backward to this second list returns yet a third list, of all the at most σd nodes whose labels have the form $c\beta_i b_i \alpha$. We need only one node returned calling backward on each node in the second list, so we can discard all but at most d nodes in the third list. Finally, applying shorter to the third list returns a fourth list, of all d nodes whose labels have the form $b_i\alpha$, each of which may be repeated at most σ times in the list.

5 Implementing shorter, longer and maxlen

The BOSS representation includes a wavelet tree over the last column W of the BOSS matrix, and a bitvector L of the same length with 1s marking where nodes' intervals end. In our example, W = TCCGTGGATAA\$C and $L = 1110111011111$.

Now we can implement maxlen$([i,j],a)$ in $\mathcal{O}(\log\sigma)$ time: we use rank and select on W to find an occurrence $W[r]$ of a in $W[i..j]$, if there is one; we then use rank and select on L to find the last bit $L[i'-1] = 1$ with $i' \leq r$ and the first bit $L[j'] = 1$ with $j' \geq r$, and return $[i',j']$. (If there is no occurrence of 1 strictly before $L[r]$, then we set $i' = 1$.) We can implement maxlen$([i,j],*)$ in $\mathcal{O}(1)$ time: instead of using rank and select on W to find r, we simply choose any r between i and j.

In our example, for $\mathsf{maxlen}([4,6],\mathrm{G})$ we first find an occurence $W[r]$ of G in $W[4..6]$, which could be either $W[4]$ or $W[6]$; if we choose $r=4$ then the last bit $L[i'-1]=1$ with $i' \leq r$ is $L[3]$ and the first bit $L[j']=1$ with $j' \geq r$ is $L[5]$, so we return $[i',j']=[4,5]$; if we choose $r=6$ then the last bit $L[i'-1]=1$ with $i' \leq r$ is $L[5]$ and the first bit $L[j']=1$ with $j' \geq r$ is $L[6]$, so we return $[i',j']=[6,6]$.

To implement shorter and longer, we store a wavelet tree over the sequence L^* in which $L^*[i]$ is the length of the longest common suffix of the label of the node in the original graph whose interval includes i, and the label of the node whose interval includes $i+1$; this takes $\mathcal{O}(\log K)$ bits per $(K+1)$-mer in the matrix. To save space, we can omit Ks in L^*, since they correspond to 0s in L and indicate that i and $i+1$ are in the interval of the same node in the original graph; the wavelet tree then takes $\mathcal{O}(\log K)$ bits per node in the original graph and $\mathcal{O}(n \log K)$ bits in total. In our example, $L^* = 0,1,0,3,2,1,0,3,2,0,1,1$ (and we can omit the 3s to save space).

For $\mathsf{shorter}([i,j],k)$, we use the wavelet tree over L^* to find the largest $i' \leq i$ and the smallest $j' \geq j$ with $L^*[i'-1], L^*[j'] < k$ and return $[i',j']$, which takes $\mathcal{O}(\log K)$ time. For $\mathsf{longer}([i,j],k)$, we use the wavelet tree to find the set $B = \{b : L^*[b] < k\,;\, i-1 \leq b \leq j\}$ — which includes $i-1$ and j — and then, for each consecutive pair (b,b') in B, we report $[b+1,b']$; this takes a total of $\mathcal{O}(|B| \log K)$ time. With these implementations, if the time bounds for $\mathsf{forward}(v,a)$, $\mathsf{backward}(v)$ and $\mathsf{lastchar}(v)$ are $\mathcal{O}(t_{\mathsf{forward}})$, $\mathcal{O}(t_{\mathsf{backward}})$ and $\mathcal{O}(t_{\mathsf{lastchar}})$ when v is a node in the original graph, respectively, then they are $\mathcal{O}(t_{\mathsf{forward}} + \log\sigma + \log K)$, $\mathcal{O}(\sigma(t_{\mathsf{backward}} + \log K))$ and $\mathcal{O}(t_{\mathsf{lastchar}} + 1)$ when v is not a node in the original graph.

In our example, for $\mathsf{shorter}([4,5],2)$ we find the largest $i' \leq 4$ and the smallest $j' \geq 5$ with $L^*[i'-1], L^*[j'] < 2$ — which are 4 and 6, respectively — and return $[4,6]$. For $\mathsf{longer}([4,6],3)$ we find the set $B = \{b : L^*[b] < 3\,;\, 3 \leq b \leq 6\} = \{3,5,6\}$ and report $[4,5]$ and $[6,6]$.

A smaller but slower approach is not to store L^* explicitly but to support access to any cell $L^*[i]$ by finding the nodes in the original graph whose intervals include i and $i+1$, then using backward and lastchar to compute their labels and find the length of their longest common suffix; this takes a total of $\mathcal{O}(K(t_{\mathsf{backward}} + t_{\mathsf{lastchar}}))$ time. To implement shorter and longer, we store a range-minimum data structure [26] over L^*, which takes $2n + o(n)$ bits and returns the position of the minimum value in a specified substring of L^* in $\mathcal{O}(1)$ time.

For $\mathsf{shorter}([i,j],k)$, we use binary search and range-minimum queries to find the largest $i' \leq i$ and the smallest $j' \geq j$ with $L^*[i'-1], L^*[j'] < k$ and return $[i',j']$, which takes $\mathcal{O}(K(t_{\mathsf{backward}} + t_{\mathsf{lastchar}}) \log(n\sigma))$ time. For $\mathsf{longer}([i,j],k)$, we recursively split $[i,j]$ into subintervals with range-minimum queries, at each step using backward and lastchar to check that the minimum value found is less than k; this takes $\mathcal{O}(K(t_{\mathsf{backward}} + t_{\mathsf{lastchar}}))$ time per node returned. With these implementations, $\mathsf{forward}(v,a)$, $\mathsf{backward}(v)$ and $\mathsf{lastchar}(v)$ take $\mathcal{O}(t_{\mathsf{forward}} + K(t_{\mathsf{backward}} + t_{\mathsf{lastchar}}) \log(n\sigma))$, $\mathcal{O}\big(\sigma K(t_{\mathsf{backward}} + t_{\mathsf{lastchar}}) \log(n\sigma) + \sigma^2 t_{\mathsf{backward}}\big)$ and $\mathcal{O}(t_{\mathsf{lastchar}} + 1)$ time, respectively, when v is not a node in the original graph.

For $\sigma = \mathcal{O}(1)$, our bounds are summarized in the following theorem.

Theorem 1. When $\sigma = \mathcal{O}(1)$, we can store a variable-order de Bruijn graph in $\mathcal{O}(n \log K)$ bits on top of the BOSS representation, where n is the number of nodes in the Kth-order de Bruijn graph, and support forward and backward in $\mathcal{O}(\log K)$ time and lastchar in $\mathcal{O}(1)$ time. We can also use $\mathcal{O}(n)$ bits on top of the BOSS representation, at the cost of using $\mathcal{O}(K \log n / \log\log n)$ time for forward and backward.

6 Experiments

We have implemented the wavelet tree based data structure on top of an efficient implementation of the BOSS single-K data structure[1]. Both structures make use of the SDSL-lite software library[2] for succinct data structures, and the the construction code makes use of the STXXL software library[3] for external memory data structures and sorting. The construction code is also concurrent in many places.The smaller but slower version was not implemented.

Our test machine was a server with a hyperthreaded quad-core 2.93 Ghz Intel Core i7-875K CPU and 16 GB RAM running Ubuntu Server 14.04. Four Samsung 850 EVO 250GB SSDs were used for temporary storage for STXXL, with a fifth identical drive used for temporary storage for SDSL-Lite and final graph output. In order to make use of STXXL's parallel disk and asynchronous I/O support[4], the SSDs were not in a RAID configuration. The input files were read from a mechanical 2TB 7200 RPM disk.

To minimize the effect of external factors on our results, each experiment was repeated three times with the minimum values reported. The swap file was disabled, forcing the operating system to keep each graph completely in memory, and there were no other users on the server.

6.1 Test Data

In order to test the scalability of our approach, we repeated the experiment on readsets of varying size. Our first data set consists of 27 million paired-end 100 character reads (strings) from *E. coli* (substr. K-12). It was obtained from the NCBI Short Read Archive (accession ERA000206, EMBL-EBI Sequence Read Archive). The total size of this data set is around 2.3 GB compressed on disk (6 GB uncompressed).

The second data set is 36 million 155 character reads from the Human chromosome 14 Illumina reads used in the GAGE benchmark[5], totalling 1.3 GB compressed on disk (6 GB uncompressed).

For our third data set we obtained 1,415 million paired-end 100 character Human genome reads (SRX01231) that

1. The implementation is released under GPLv3 license at http://github.com/cosmo-team/cosmo. As Cosmo is under continuous development, a static snapshot of the code used in this paper is available at https://github.com/cosmo-team/cosmo/tree/varord-paper.
2. https://github.com/simongog/sdsl-lite
3. https://github.com/stxxl/stxxl
4. http://stxxl.sourceforge.net/tags/master/design_algo_sorting.html
5. http://gage.cbcb.umd.edu/

TABLE 1
Input size (top), construction time, memory use, and structure size (middle), and mean time taken for each navigation operation (lower), for all data sets and both structures. For variable-order, the multipliers in parenthesis are the increase over the fixed-order results. Cells marked "N/A" for fixed-order indicate operations not possible with that structure.

Dataset	*E. coli*		Human chromosome 14		Human		Parrot	
DSK Size (GB)	1.52		6.88		26.74		70.28	
Number of K-mers	204,098,902		461,445,333		1,794,522,954		4,716,731,435	
BOSS Order	fixed	variable	fixed	variable	fixed	variable	fixed	variable
Construction (mins)	3.93	5.09 (1.30×)	14.37	18.72 (1.30×)	64.45	83.85 (1.30×)	162.58	225.73 (1.39×)
Graph Size (GB)	0.16	0.41 (2.56×)	0.40	1.38 (3.45×)	1.67	5.42 (3.25×)	4.20	13.60 (3.24×)
Peak RAM (GB)	3.16	3.16 (1.00×)	3.22	3.22 (1.00×)	7.65	9.31 (1.22×)	15.30	15.29 (1.00×)
Peak Disk (GB)	12.17	12.17 (1.00×)	56.68	56.68 (1.00×)	248.37	248.37 (1.00×)	562.28	562.28 (1.00×)
forward (μs)	6.00	17.03 (2.84×)	6.24	16.17 (2.59×)	7.07	18.31 (2.59×)	7.77	19.39 (2.50×)
backward (μs)	8.23	59.77 (7.26×)	8.47	55.63 (6.57×)	9.27	62.85 (6.78×)	10.46	63.87 (6.11×)
lastchar (μs)	0.01	0.01 (1.00×)	0.01	0.01 (1.00×)	0.01	0.01 (1.00×)	0.01	0.01 (1.00×)
maxlen (μs)	N/A	1.43	N/A	1.56	N/A	2.02	N/A	2.46
maxlen$_c$ (μs)	N/A	5.41	N/A	5.98	N/A	6.71	N/A	7.49
shorter$_1$ (μs)	N/A	14.65	N/A	17.72	N/A	19.54	N/A	19.84
shorter$_2$ (μs)	N/A	14.83	N/A	17.79	N/A	19.68	N/A	19.98
shorter$_4$ (μs)	N/A	15.11	N/A	18.02	N/A	19.90	N/A	20.20
shorter$_8$ (μs)	N/A	15.73	N/A	18.39	N/A	20.29	N/A	20.64
longer$_1$ (μs)	N/A	21.53	N/A	18.61	N/A	21.06	N/A	20.57
longer$_2$ (μs)	N/A	56.96	N/A	41.08	N/A	49.01	N/A	47.07
longer$_4$ (μs)	N/A	503.60	N/A	323.50	N/A	446.51	N/A	428.97
longer$_8$ (μs)	N/A	6441.33	N/A	5338.38	N/A	18349.80	N/A	24844.80

were generated by Illumina Genome Analyzer (GA) IIx platform. The total size of this data set is 130 GB compressed on disk (470 GB uncompressed).

Our fourth data set is 700 million paired-end 101 character reads, and 131 paired-end 75 character reads from the short insert libraries of the Parrot data (ERA201590) provided in Assemblathon 2 [27]. The total size of this data set is 64 GB compressed on disk (245 GB uncompressed).

We used DSK [28] on each data set to find the unique $(K+1)$-mers. It is usual to have DSK ignore low-frequency $(K+1)$-mers (as they may result from sequencing errors). However, removing such $(K+1)$-mers may result in the removal of some k-mers with $k \leq K$ that would otherwise have an acceptable frequency. We therefore set the frequency threshold to be as low as possible: 1 (accepting all $(K+1)$-mers) for all data sets except for the Human genome data set, which was too big for our SSDs during construction, and too big to fit into RAM afterwards. Hence, for the Human genome data set, the frequency threshold was 2.

A value of $K = 27$ was chosen for the *E. coli* data, and $K = 55$ for the Human data sets as these values produced good assemblies in previous papers (see, e.g., [18]). $K = 55$ was also chosen for the Parrot data set, as it produced a graph that almost filled the main memory. The resulting file sizes and $(K+1)$-mer totals are shown in Table 1.

6.2 Construction

In order to convert the input DSK data to the format required by BOSS (in the correct order, with dummy edges, as required by both single-K and variable-K structures), we use the following process, which has been designed with disk I/O in mind.

While reading the DSK input data, we generate and add the reverse complements for each $(K+1)$-mer, then sort them by their first K symbols (the source nodes). Concurrently, we also sort another copy of the $(K+1)$-mers and their reverse complements by their last K symbols (the target nodes). Let the resulting tables be A and B, respectively.

Next, we calculate the set differences $A - B$, comparing only the K-length prefixes to the K-length suffixes respectively. This tells us which source nodes do not appear as target nodes, which we prepend with \$ signs to create the required incoming dummy edges (K each), and then sort by the first K symbols. Concurrently, we also calculate $B - A$ to give us the nodes requiring outgoing dummy edges (to which we append \$). Let the resulting tables be I and O, respectively. At this point B can be deleted.

Finally, we perform a three-way merge (by first K symbols) of A, I, and O, outputting the rightmost column. In the case of the variable-K graph, we also calculate the L^* values while merging. Finally, we construct the necessary succinct indexes from the output.

The time bottleneck in the above process is clearly in sorting the A and I tables. $|I|$ can be as big as $K|A|$, but in practice only 1% or fewer nodes require incoming dummies. Our elements are of size $\mathcal{O}(K)$, thus, overall, construction of both data structures takes $\mathcal{O}\big(K^2|A|\log|A|\big)$ time and $\mathcal{O}\big(K^2|A|\big)$ space in theory, but in practice takes $\mathcal{O}(K|A|\log|A|)$ time and $\mathcal{O}(K|A|)$ space.

6.3 Results

For each data set, the $(K+1)$-mers from DSK (and their reverse complements) were converted into the BOSS format using the process outlined in 6.2, using the external memory vectors and multithreaded, external memory sort from STXXL. The BOSS structure and L^* wavelet tree were then built using indexes from SDSL-lite.

Construction times and structure sizes are shown in Table 1. While the variable-K BOSS structure is around 30% slower to build, and 2.6 to 3.5 times larger than the standard BOSS structure, this is clearly much faster and less space consuming than building K separate instances of the BOSS structure. The peak RAM and disk usage is the same for both structures except in the case of the Human genome data set, where the variable-K BOSS structure used 22% more RAM.

To measure navigation functions forward and backward we took the mean time over 20,000 random queries. For the variable-K graph, the k values for each node were chosen randomly between 8 and K. Results are shown in Table 1. The new structure makes the forward operation 2.5 to 3 times slower for $k < K$, though we note that for $k = K$ forward time is identical. The backward operation is much slower in the new structure, but is much less frequently used than forward in assembly algorithms (for a variation that supports fast backward calculations, see [29]). We also measured lastchar, which took only nanoseconds on both structures.

To see how fast the order can be changed, we timed shorter and longer for changes of 1, 2, 4, and 8 symbols. Our experiments show that in practice changing order by a single symbol (shorter_1 and longer_1) is a cheap operation, taking around the same time as forward. For larger changes in order, the time for shorter is stable (shorter_1, shorter_2, shorter_4, and shorter_8 all take roughly the same time), whereas longer takes significantly more time as the difference in order increases. This is because longer must compute a set of nodes, and the size of that set grows roughly exponentially with the change in order (longer takes around 10μs per node when averaged over the size of the resulting set).

As expected, maxlen is very fast (it requires a single rank and select operation over a bit vector), and only slightly affected when finding the specified outgoing edge label (which uses a rank and select over the BOSS wavelet tree instead).

7 Conclusion

We have described a method for efficiently representing multiple de Bruijn graphs of different orders in a single succinct data structure. As well as the usual graph traversal operations, the data structure supports new operations which allow the order of the de Bruijn graph to be changed on the fly. This data structure has the potential to greatly improve the memory and space usage of current state-of-the-art assemblers that build the de Bruijn graph for multiple values of K, and ultimately allow those assemblers to scale to large, eukaryote genomes. The integration of our new data structure into a real assembler is thus our most pressing avenue for future work.

Acknowledgments

We thank two anonymous reviewers for thoughtful comments that materially improved this manuscript.

References

[1] D. Haussler *et al.*, "Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species," *J. Hered.*, vol. 100, no. 6, pp. 659–674, 2009.

[2] S. Ossowski *et al.*, "Sequencing of natural strains of *Arabidopsis Thaliana* with short reads," *Genome Res.*, vol. 18, no. 12, pp. 2024–2033, 2008.

[3] The 1000 Genomes Project Consortium, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, no. 7422, pp. 56–65, 2012.

[4] P. J. Turnbaugh *et al.*, "The Human Microbiome Project: exploring the microbial part of ourselves in a changing world," *Nature*, vol. 449, no. 7164, pp. 804–810, 2007.

[5] R. Idury and M. Waterman, "A new algorithm for DNA sequence assembly," *J. Comput. Biol.*, vol. 2, no. 2, pp. 291–306, 1995.

[6] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proc. Natl. Acad. Sci.*, vol. 98, no. 17, pp. 9748–9753, 2001.

[7] A. Bankevich *et al.*, "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing," *J. Comput. Biol.*, vol. 19, no. 5, pp. 455–477, 2012.

[8] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "IDBA–a practical iterative de Bruijn graph *de novo* assembler," in *Proc. RECOMB*, 2010, pp. 426–440.

[9] R. Li *et al.*, "*De novo* assembly of human genomes with massively parallel short read sequencing," *Genome Res.*, vol. 20, no. 2, pp. 265–272, 2010.

[10] J. T. Simpson *et al.*, "ABySS: a parallel assembler for short read sequence data," *Genome Res.*, vol. 19, no. 6, pp. 1117–1123, 2009.

[11] J. Butler *et al.*, "ALLPATHS: *de novo* assembly of whole-genome shotgun microreads," *Genome Res.*, vol. 18, no. 5, pp. 810–820, 2008.

[12] C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi, and K. Sadakane, "Variable-order de Bruijn graphs," in *Proc. Data Compression Conference (DCC)*. IEEE, 2015, pp. 383–392.

[13] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, p. 479486, 2011.

[14] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. ALENEX*. SIAM, 2007, pp. 60–70.

[15] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithm. Mol. Biol.*, vol. 8, no. 22, 2012.

[16] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de Bruijn graphs," in *Proc. WABI*, 2012, pp. 225–235.

[17] M. Burrows and D. J. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.

[18] R. Chikhi, A. Limasset, S. Jackman, J. Simpson, and P. Medvedev, "On the representation of de Bruijn graphs," in *Proc. RECOMB*, 2014, pp. 35–55.

[19] Y. Peng, H. C. Leung, S. M. Yiu, and F. Y. Chin, "IDBA-UD: a *de novo* assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[20] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph," arXiv:1409.7208, Sep 2014.

[21] Y. Lin and P. Pevzner, "Manifold de Bruijn graphs," in *Proc. WABI*, 2014, pp. 296–310.

[22] I. Munro, "Tables," in *Proc. FSTTCS*, 1996, pp. 37–42.

[23] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. SODA*, 2003, pp. 841–850.

[24] G. Navarro, "Wavelet trees for all," *J. Discrete Algorithms*, vol. 25, pp. 2–20, 2014.

[25] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theor. Comp. Sci*, vol. 426, pp. 25–41, 2012.

[26] J. Fischer and V. Heun, "Space-efficient preprocessing schemes for range minimum queries on static arrays," *SIAM J. Comput.*, vol. 40, no. 2, pp. 465–492, 2011.

[27] K. R. Bradnam *et al.*, "Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, pp. 1–31, 2013. [Online]. Available: http://dx.doi.org/10.1186/2047-217X-2-10

[28] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, 2013.

[29] D. Belazzougui, T. Gagie, V. Mäkinen, M. Previtali, and S. J. Puglisi, *Bidirectional Variable-Order de Bruijn Graphs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 164–178. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49529-2_13