Succinct de Bruijn Graphs

by

Alexander BOWE

Dissertation

submitted to the Department of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
March 2020

Contents

Chapter 1

Introduction

While consumer-grade genotyping – such as that used by 23andMe – has proven a popular and inexpensive method to determine Single Nucleotide Polymorphisms (SNPs) in individuals, such methods can only detect a set of reference genes, thus limiting their ability to detect all but the simplest variations.

Whole genome sequencing (without a reference) is a powerful alternative, albeit comparatively expensive. However, the price has been steadily declining: while the Human Genome Project cost $2.7 billion to complete in 2003 [1], as of 2019 it is possible to have a genome sequenced for $299 [2], and the price continues to drop.

This decline in price is in large part owed to the advent of Next Generation Sequencing (NGS) machines. The "Sanger" sequencing method used in the Human Genome project required a high degree of human interaction, which NGS machines have subsequently automated, greatly increasing the speed and decreasing the cost. And although NGS machines produce much shorter reads (200 bases versus 800 bases in Sanger sequencing – a human genome is 3.4 billion bases), this is overcome by re-sequencing the same DNA.

The process of combining short reads into longer sequences is called assembly, and while finding the best overlap is NP-hard [3], many practical approaches have been proposed (see surveys [4, 5, 6]).

Traditionally, assembly employed an overlap graph, where each read is a node, and an edge exists if two reads have sufficient overlap [7, 8, 9]. Assembly then involves computing a Hamiltonian tour of all nodes. This was an acceptable drawback when dealing with Sanger reads, but is prohibitively expensive to deal with the abundant data that NGS machines produce.

Eulerian assembly replaces the overlap graph with a de Bruijn graph [10, 11], where every k-length substring of the reads is a node, and the directed edges are defined by the $k+1$-length substrings that contain the k-length vertices, where k is a user-selected parameter. For example,

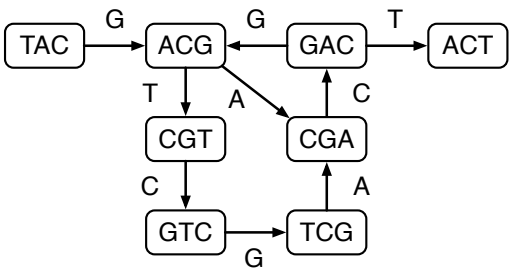


Figure 1.1: The $k = 3$ de Bruijn graph of reads 'TACGT', 'TACGA', 'ACGTC', 'GTCGA', 'CGACT', and 'CGACG'. The edges are given by the substrings of length $k + 1 = 4$ from all of the reads ('TACG', 'ACGA', 'ACGT', 'CGTC', and so on), and are represented by their right-most symbol connecting the two vertices given by their two substrings of length $k = 3$ (e.g. $TAC \xrightarrow{G} ACG$). The longest contig is found by starting at 'ACG', and following its branch labeled 'T', and all subsequent edges, until we reach another branch at vertex 'GAC' (which has two edges labeled 'G' and 'T'), giving us 'ACGTCGAC' (8 bases).

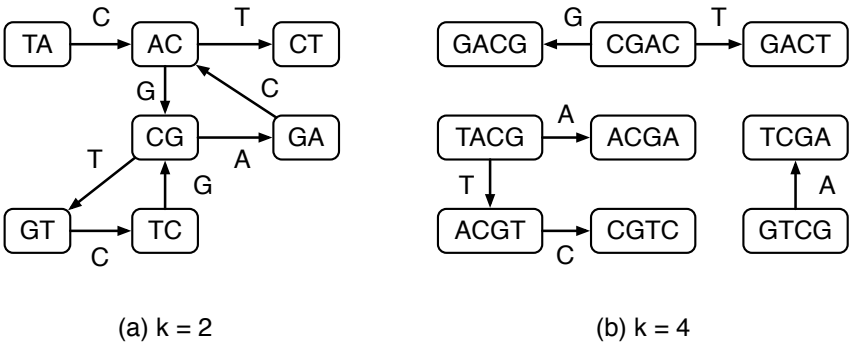


Figure 1.2: (a) The $k = 2$ de Bruijn graph of strings 'TACGT', 'TACGA', 'ACGTC', 'GTCGA', 'CGACT', and 'CGACG', and (b) the $k = 4$ de Bruijn graph of strings 'TACGT', 'TACGA', 'ACGTC', 'GTCGA', 'CGACT', and 'CGACG'. The longest contig for (a) is 'CGTCG' (5 bases), and the longest contig for (b) is 'TACGTC' (6 bases).

for $k = 3$, the sequence 'TACGT' yields the edges 'TACG' and 'ACGT', and the edge 'TACG' connects the vertices 'TAC' and 'ACG' by dropping the initial 'T' and appending a 'G'. A complete example is given in Figure 1.1.

Contigs (contiguous sequences) are then found by following the edges between two branches (see Figure 1.1). Most modern assembler programs use this paradigm [12, 13, 14, 15, 16, 17, 18, 19]. See [20] for a thorough explanation of de Bruijn graphs and their use in assembly.

While the de Bruijn graph can be constructed more efficiently than the overlap graph, it

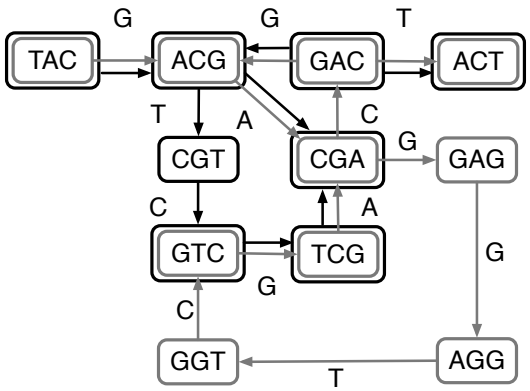


Figure 1.3: A $k = 3$ Colored de Bruijn Graph for two sets of reads. The black nodes and edges represent the reads 'TACGT', 'TACGA', 'ACGTC', 'GTCGA', 'CGACT', and 'CGACG' (the de Bruijn Graph from Figure 1.1). The gray nodes and edges represent the reads 'TACGA', 'GTCGACG', 'CGACT', 'CGAGGTC'.

remains a bottleneck in assembly, both in terms of speed and size, with a de Bruijn graph of a human genome requiring 300 GB of RAM [15]. Previous work has reduced this to 30 GB [21]. This thesis reduces this to 2 GB, bringing it in line with commodity hardware – a student or field biologist could now perform this on their laptop. Around the same time as the work done in this thesis, an alternative approach with similar performance was published [22], but the Burrows-Wheeler based approach taken in this thesis offers more flexibility and faster edge traversal.

It is common for modern assemblers to build multiple de Bruijn graphs. This is because the k parameter significantly influences the topology – if k is too large there may be too few edges, causing gaps in the graph. But if k is too small, the vertices may have too many edges, increasing ambiguity. Both of these issues lead to shorter contigs, as is demonstrated in Figure 1.2. In fact, due to non-uniform coverage of NGS data, different areas of the same graph may benefit from differing k values. To overcome this, assemblers such as Spades and IDBA [12, 13] build de Bruijn graphs for increasing values of k, and use them in tandem. This yields better quality assemblies, but is slowed down proportionally to the number of k values used. This thesis introduces the first variation of the de Bruijn graph that can be built once, yet change k values on-the-fly, at only a modest increase in size over the base succinct de Bruijn graph, taking only 3.5 times the space, and only 30% longer to construct than a graph for a *single* value of k.

Finally, in population genomics, biologists assemble multiple genomes in order to study

the variations, among, for example, 10,000 vertebrate genomes [23]. To avoid constructing multiple graphs, Iqbal et al. proposed the Colored de Bruijn Graph [24]. This graph capitalizes on the fact that DNA is rarely unique to an individual. It does this by first constructing a de Bruijn Graph of the entire populations NGS reads, and assigning each individual a unique *color*, which annotates the vertices and edges (see Figure 1.3). In this thesis, we further augment our succinct de Bruijn Graph to efficiently store these colors. When tested with four plant genomes, Iqbal's structure required 101 GB RAM, while ours only requires 4 GB of RAM. Furthermore, our structure was able to store all known E. Coli genomes in 42 GB, where Iqbal's was not able to complete, but is estimated to require 3 TB of RAM. We also demonstrate the use of our structure in creating a database of all Antimicrobial Resistance Genes, requiring 245 GB of RAM (an estimated 18 TB with Iqbal's structure), for rapidly locating resilient bacterial outbreaks in food supply chains.

1.1 Original Papers

This thesis is comprised of the following three published papers, as well as a forth paper which is included as an appendix due to its relevance to the third paper while not being core to this thesis.

Paper I: Succinct de Bruijn Graphs
Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya.
In *Algorithms in Bioinformatics. Proceedings of WABI 2012* (B. Raphael and J. Tang, editors). Lecture Notes in Computer Science, vol. 7534, pages 225–235. Springer, Berlin, Heidelberg, 2012.

We propose a new succinct de Bruijn graph representation. If the de Bruijn graph of k-mers in a DNA sequence of length N has m edges, it can be represented in $4m + o(m)$ bits. This is much smaller than existing representations. The numbers of outgoing and incoming edges of a node are computed in constant time, and the outgoing and incoming edge with given label are traversed in constant time and $\mathcal{O}(k)$ time, respectively. The data structure is constructed in $\mathcal{O}(Nk \log m / \log \log m)$ time using no additional space.

Paper II: Variable-Order de Bruijn Graphs
Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane.
In *Proceedings of the 2015 Data Compression Conference*, Snowbird, Utah, pages 383–392. IEEE, 2015.

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., $G_{K_1}, G_{K_2}, \ldots, G_{K_d}$. This paradigm increases the quality of the assembly produced, at the cost of greatly increases runtime, due to constructing 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.

Paper III: Succinct Colored de Bruijn graphs

Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher.
Bioinformatics, 33(20):3181–3187, 2017.

Iqbal et al. (Nature Genetics, 2012) introduced the *colored de Bruijn graph*, a variant of the classic de Bruijn graph, which is aimed at "detecting and genotyping simple and complex genetic variants in an individual or population". Because they are intended to be applied to massive population level data, it is essential that the graphs be represented efficiently. Unfortunately, current succinct de Bruijn graph representations are not directly applicable to the colored de Bruijn graph, which requires additional information to be succinctly encoded as well as support for non-standard traversal operations. Our data structure dramatically reduces the amount of memory required to store and use the colored de Bruijn graph, with some penalty to runtime, allowing it to be applied in much larger and more ambitious sequence projects than was previously possible.

Paper IV: Relative Select (Appendix A)

Christina Boucher, Alexander Bowe, Travis Gagie, Giovanni Manzini, and Jouni Sirén
In *String Processing and Information Retrieval. Proceedings of SPIRE 2015* (C. Iliopoulos, S. Puglisi, and E. Yilmaz, editors). Lecture Notes in Computer Science, vol. 9309, pp. 149–155. Springer, Cham, 2015.

Motivated by the problem of storing coloured de Bruijn graphs, we show how, if we can already support fast select queries on one string, then we can store a little extra information and support fairly fast select queries on a similar string.

Preface to Paper I

Conway's 2011 paper [21] received a lot of attention for being such a drastic decrease in the memory requirement of a de Bruijn graph. It was the first succinct de Bruijn graph, using one of the key building blocks of succinct data structures proposed by Okanohara and Sadakane [25] – the sparse succinct bit vector. In Conway's representation, the edges, or $(k+1)$-mers of the de Bruijn graph that were present would be represented with a 1 in the bit vector, and a 0 for every *possible* $(k+1)$-mer that was not in the graph.

While a de Bruijn graph would typically have hundreds of millions, or even multiple billions of edges in the case of humans, the number of possible edges would be on the order of 4^{k+1}. For a common k value such as 31, this would be $4^{32} \simeq 1.84 \times 10^{19}$, meaning that only one tenth of a billionth of possible edges would be present. This meant that even though the sparse bit vector was tuned to handle very sparse data, it still had to represent those 0s somehow.

After discussing this with Sadakane, he identified that the de Bruijn graph is similar to a bound-depth suffix trie, and could hence be represented with something like a Burrows–Wheeler transform, effectively removing the need for representing edges that were not in the graph, and allowing it to scale in size without a relation to k.

The following paper introduces this Burrows–Wheeler-inspired data structure in theory, with data being reported during WABI 2012, and later in the blog post in Appendix A. In the end, we reduced it to around 4GB uncompressed when using the same data and k value as Conway, and around 2GB when using a compressed Wavelet Tree to represent the edges (which was slower to traverse).

My contribution to this paper was identifying the opportunity to compress de Bruijn graphs beyond the work of Conway by avoiding storing edges that were not present in the data, implementing and testing the succinct de Bruijn graph in C++, and presenting it.

Chapter 2

Succinct de Bruijn Graphs

We propose a new succinct de Bruijn graph representation. If the de Bruijn graph of k-mers in a DNA sequence of length N has m edges, it can be represented in $4m + o(m)$ bits. This is much smaller than existing ones. The numbers of outgoing and incoming edges of a node are computed in constant time, and the outgoing and incoming edge with given label are found in constant time and $\mathcal{O}(k)$ time, respectively. The data structure is constructed in $\mathcal{O}(Nk \log m / \log \log m)$ time using no additional space.

2.1 Introduction

Within the last two decades, assembling a genome from enormous amount of reads from various DNA sequencers has been one of the most challenging and important computational problems in molecular biology. Though the problem is proved to be NP-hard [3], many algorithms have been proposed for the problem (see the surveys [4, 5, 6]). Most of these algorithms follow a so-called Overlap-Layout-Consensus strategy, where an algorithm first finds overlaps between reads, next layouts these reads, and finally finds the consensus genome. These algorithms can be categorized into two types, due to the graph used in the overlap phase.

Most old-time assembly algorithms (especially for the long Sanger reads) first construct a graph called the *overlap graph* after finding the overlapping pairs of reads, where each node represents a read and edges are constructed between nodes *iff* the corresponding two reads have an overlap of enough length [7, 8, 9]. But this strategy is difficult to apply against the huge data from more recent epoch-making next-generation sequencers (NGSs). The NGS machines can sequence vast amount of genome data. It makes it computationally very hard to compare all the pairs of reads. Moreover, most NGSs cannot read long DNA fragments (*e.g.*, at most

200bp in the case of Illumina HiSeq2000), and their read lengths are not long enough to detect overlaps with enough lengths between reads. To conquer these problems, many recent assembler algorithms utilize a graph called the *de Bruijn graph* in the overlap phase [11, 14, 15, 17, 18, 19], instead of the overlap graph.

A de Bruijn graph is a graph where each node represents a k-mer (a substring of length k) that exists in the reads, and an edge exists *iff* there is an exact overlap of length $k-1$ between the corresponding k-mers. The de Bruijn graph can be constructed more efficiently than the overlap graph in many cases, but the overlap phase is still the bottleneck of most assembly algorithms based on the de Bruijn graph. This is because storing the de Bruijn graph requires huge amount of memory. Thus we focus on reducing the memory required for the de Bruijn graph in this paper.

There have been proposed only two data structures for reducing the size of memory for the de Bruijn graph. The succinct data structure proposed by Conway and Bromage [21] is a data structure that straightforwardly represents the de Bruijn graph by a bit vector. Its representation should be smaller than a naive ordinary implementation of the de Bruijn graph, but it still requires $O(m \cdot k)$ memory, where k is the k-mer length and m is the number of edges in the de Bruijn graph, which means it would be very large when k is large. The other data structure is by Ye et al. [26], which stores only a subset of nodes of the de Bruijn graph to save memory, but it is not actually the de Bruijn graph.

In this paper, we propose a new succinct representation of a de Bruijn graph which only requires $m(2+\log\sigma)$ bit to store[1], where σ is the alphabet size (*i.e.*, $\sigma = 4$ in the case of DNA). The size of this representation is not affected by the value of k and is much smaller than either of the two previous methods. Moreover we will present the algorithm to construct the data structure on-line. Our main result is summarized as follows:

Theorem 2.1. *The k-dimensional de Bruijn graph of M string of total length N on an alphabet of size σ can be stored in $m(2+\log\sigma)+\mathcal{O}((\sigma+M)\log m)+o(m\log\sigma)$ bits where m is the number of edges in the graph. The numbers of outgoing and incoming edges of a node are computed in $\mathcal{O}(\log\sigma/\log\log m)$ time, and the outgoing and incoming edge with given label are found in $\mathcal{O}(\log\sigma/\log\log m)$ time and $\mathcal{O}(k\log^2\sigma/\log\log m)$ time, respectively. The node for a given k-mer is found in $\mathcal{O}(k\log\sigma/\log\log m)$ time. If $\sigma = \mathrm{polylog}(m)$, the time complexities become $\mathcal{O}(1)$, $\mathcal{O}(1)$, $\mathcal{O}(k\log\sigma)$, and $\mathcal{O}(k)$ time, respectively.*

Theorem 2.2. *The k-dimensional de Bruijn graph of a string of length N can be constructed in $\mathcal{O}\left(Nk\cdot\frac{\log m}{\log\log m}(1+\frac{\log\sigma}{\log\log m})\right)$ time using no additional space. This representation can be converted to the static one in $\mathcal{O}\left(\frac{m\log m}{\log\log m}(1+\frac{\log\sigma}{\log\log m})\right)$ time.*

[1]The base of logarithm is 2.

For DNA sequences ($\sigma = 4$), the succinct de Bruijn graph can be constructed in $\mathcal{O}(Nk \log m / \log\log m)$ time and its space becomes $4m + o(m)$ bits. This is much smaller than existing ones. For example, the succinct representation of Conway and Bromage [21] uses 40.8GB for storing a de Bruijn graph with m = 12,292,819,311 edges and $k = 27$ (28.5 bits per edge). On the other hand, if we use an efficient implementation of *rank*/*select* data structures [25] for our representation, the estimated size is less than 5 bits per edge. Therefore the above graph is stored in less than 8GB.

2.2 Preliminaries

2.2.1 de Bruijn graphs

In the original definition [27], the k-dimensional de Bruijn graph of σ symbols is a directed graph representing overlaps between strings of symbols defined as follows. The graph has σ^k nodes, consisting of all length-k strings of the symbols. A node is denoted by $(u_1, \ldots, u_k)$ where $u_1, \ldots, u_k$ are symbols. For any pair of nodes $u = (u_1, \ldots, u_k)$ and $v = (v_1, \ldots, v_k)$ such that $u_2 = v_1, u_3 = v_2, \ldots, u_k = v_{k-1}$, the graph has a directed edge from u to v labeled with v_k. In this paper we call it the complete k-dimensional de Bruijn graph of σ symbols.

The de Bruijn graphs considered in this paper are subgraphs of the complete de Bruijn graph. We define the k-dimensional de Bruijn graph of a string T as follows. The nodes of the graph correspond to all length-k substrings of T. If the string is of length N, the graph has at most $N - k + 1$ nodes. The edges of the graph are defined in the same way as the complete de Bruijn graph. For convenience, we add k characters \$ at the head of the string, and a \$ at the end.

We can also store a set of M strings $T_1, \ldots, T_M$ as follows. We append a terminator $\$_i$ to the tail of each string T_i, and concatenate all the strings. Then we add k characters $\$_0$ at the head. Figure 2.1 shows an example.

2.2.2 Basic succinct data structures

Let $T = T[1]T[2]\cdots T[N]$ be a string of length N on alphabet $\mathcal{A}$, that is, $T[i] \in \mathcal{A}$ for any $i = 1, \ldots, N$. Let $\sigma = |\mathcal{A}|$ denote the alphabet size. We can store T in $N\lceil \log_2 \sigma \rceil$ bits. The space does not depend on the word size of CPU. We can retrieve any character $T[i]$ in constant time using bit operations on words.

The most basic succinct data structure is the one for computing *rank*, *select*, and *access* values on strings, which are defined as follows. The value $access(T, i)$ returns $T[i]$ for $1 \le i \le N$. The value $rank_c(T, i)$ where $c \in \mathcal{A}$ and $1 \le i \le N$ is the number of c's in $T[1]\cdots T[i]$. For

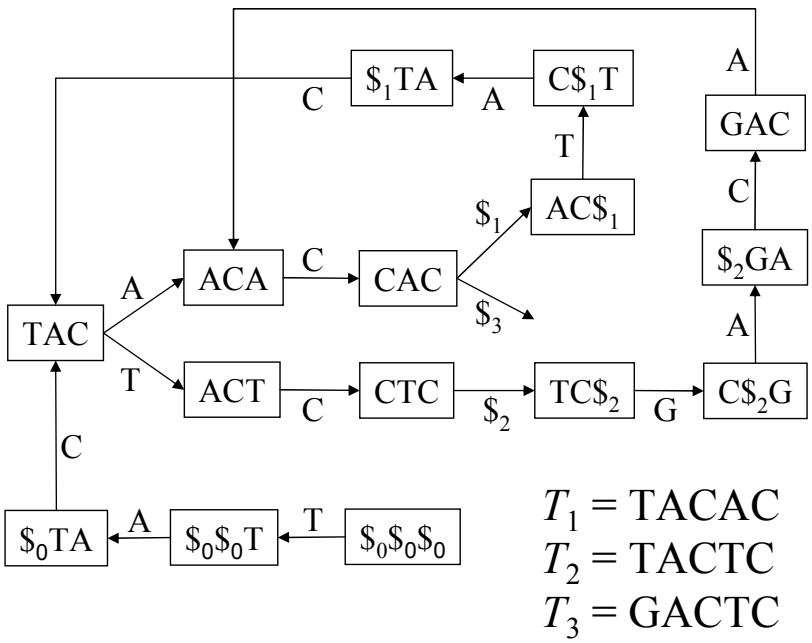


Figure 2.1: The 3-dimensional de Bruijn graph of strings 'TACAC', 'TACTC', and 'GACTC'.

any T and c we define $rank_c(T, 0) = 0$. The value $select_c(T, j)$ where $c \in \mathcal{A}$ and $1 \leq j \leq rank_c(T, N)$ is the position of j-th c in T. For any T and c we define $select_c(T, 0) = 0$ and for any $j > rank_c(T, N)$ $select_c(T, j) = N + 1$. Let $t_r(N, \sigma)$, $t_s(N, \sigma)$, and $t_a(N, \sigma)$ denote the time complexity for computing *rank*, *select*, and *access*, respectively, on a string of length N and alphabet size σ. For brevity, we assume that for any $N_1 \leq N_2$, $t_r(N_1, \sigma) \leq t_r(N_2, \sigma)$ and for any $\sigma_1 \leq \sigma_2$, $t_r(N, \sigma_1) \leq t_r(N, \sigma_2)$. Let $t_b(N, \Sigma)$ denote the maximum of $t_r(N, \sigma), t_s(N, \sigma), t_a(N, \sigma)$.

For convenience, we define $pred_c(T, i) = select_c(T, rank_c(T, i))$ which is the position of the first occurrence of c when we scan T from the position i to the head, and $succ_c(T, i) = select_c(T, rank_c(T, i - 1) + 1)$ which is the position of the first occurrence of c when we scan T from the position i to the end. If $T[i]$ is the first (last) occurrence of c, *pred* (*succ*) returns 0 ($N + 1$).

There exist many succinct data structures for *rank* and *select* on strings. Among them, we use the one by Ferragina et al. [28] for the static case (the case the string does not change). A string of T length n on an alphabet of size σ can be stored in $nH_0(T) + \mathcal{O}(\sigma \log n) + o(n \log \sigma)$ bits so that *rank*, *select* and *access* queries take $\mathcal{O}(\log \sigma / \log \log n)$ time, where $H_0(T)$ denotes the order-0 entropy of the string. Note that if the alphabet size σ is polylog(n), the queries are done in constant time. For a binary alphabet case, we can use a simpler data structure that

has the same time and space complexities [29].

For the dynamic case where the string is modified by inserting or deleting a character, we use the one by Navarro and Sadakane [30] which stores the string in $nH_0(T)+\mathcal{O}(\sigma \log n)+o(n \log \sigma)$ bits so that *rank*, *select* and *access* queries and insertion and deletion of a character take $\mathcal{O}(\frac{\log n}{\log \log n}(1+\frac{\log \sigma}{\log \log n}))$ time. For polylog-sized alphabets, the operations are done in optimal $\mathcal{O}(\log n/\log \log n)$ time. The time complexities for insert and delete are denoted by $t_u(n,\sigma)$.

2.2.3 The XBW data structure

The XBW-transform [31] is a method for compressing and indexing labeled trees. It is an extension of the Burrows-Wheeler transform [32] used for compressing and indexing strings. Given a rooted tree with n nodes where each node has a label in the set of size σ, the XBW-transform converts the tree into a representation of $2n+n \log \sigma$ bits. The size of the representation matches the information-theoretic lower bound. We can support tree navigational operations by adding small-size auxiliary indexes.

Because the XBW is for storing a tree, we cannot use it directly for storing de Bruijn graphs, which is a cyclic graph. This paper proposes a new compact representation of de Bruijn graphs of strings.

2.3 Succinct de Bruijn Graphs

Let G be a k-dimensional de Bruijn graph of a string T of length N on alphabet $\mathcal{A}$. Let n and m be the numbers of nodes and edges of G, respectively. A succinct representation of G supports the following operations:

- *outdegree*(v) returns the number of outgoing edges from node v.
- *outgoing*(v,c) returns the node w pointed to by the outgoing edge of node v with edge label c. If no such node exists, it returns -1.
- *indegree*(v) returns the number of incoming edges to node v.
- *incoming*(v,c) returns the node $w=(w_1,\ldots,w_k)$ such that there is an edge from w and v and $w_1=c$. If no such node exists, it returns -1.
- *index*(s) returns the index i of the node whose label is the string s of length k.

We define $\mathcal{A}^-$ as any set of size $|\mathcal{A}|$ such that $\mathcal{A}^- \cap \mathcal{A}=\emptyset$. Let c^- denote an element of $\mathcal{A}^-$ corresponding to an element $c \in \mathcal{A}$. We also define a function u as $u(c^-)=c$ for any $c^- \in \mathcal{A}^-$ and $u(c)=c$ for any $c \in \mathcal{A}$. We assume that the function is evaluated in constant time.

2.3.1 The succinct representation

The representation consists of the following components:

- a string $W = W[1]W[2]\cdots W[m]$ where each character is from $\mathcal{A} \cup \mathcal{A}^-$.
- a string *last* of length m on the binary alphabet $\{0, 1\}$.
- an array F of length $\sigma = |\mathcal{A}|$.

An example is shown in Figure 2.2.

The string W is defined as follows. Each character $W[i]$ represents the label of an edge of G. Each edge $u \to v$ of G is associated with the node label of u. Those edge labels are sorted in the lexicographic order of reversals of associated node labels. Ties are broken by edge labels. Let $Node[i]$ denote the node label for $W[i]$. This is not explicitly stored.

The string *last* is defined as $last[i] = 1$ if $i = n$ or $Node[i]$ is different from $Node[i+1]$, or $last[i] = 0$ otherwise. From this definition, all node labels $Node[i]$ with $last[i] = 1$ are distinct, and those indices i have one-to-one correspondence with the nodes of G. Therefore we use an index i of the strings such that $last[i] = 1$ to represent a node v. Let n denote the number of nodes.

The array F stores cumulative frequencies of the last characters of node labels. Namely, for any $c \in \mathcal{A}$, $F[c] = |\{i \mid 1 \leq i \leq m, C(i) < c\}|$ where $C(i)$ denotes the last character of $Node[i]$. Because $F[\$_i] = i$ for $i = 0, 1, \ldots, M$, we need not store them.

The array F is represented in $\mathcal{O}(\sigma \log m)$ bits. If F does not change, we can store it as it is using a simple array and $F[c]$ is computed in constant time. In a dynamic case that a new node or edge is inserted to the de Bruijn graph, we have to update F accordingly. By using a balanced binary tree, F can be maintained in $\mathcal{O}(\log \sigma)$ time.

We also use the inverse of F, that is, given i, we need to know the last character c of $Node[i]$. In a static case, this can be computed in constant time using a *rank*/*select* data structure of $\mathcal{O}(\sigma \log m) + \mathcal{O}(m \log\log m / \log m)$ bits [29]. In a dynamic case, it is done in $\mathcal{O}(\log \sigma)$ time using a balanced binary search tree. It can be improved to $\mathcal{O}(\frac{\log m}{\log\log m}(1 + \frac{\log \sigma}{\log\log m}))$ time using [30]. This data structure uses $\mathcal{O}(\sigma \log n) + \mathcal{O}(m \log\log m / \log m)$ bits. Let t_f denote the largest time complexity of those operations.

A character $W[i]$ is from either $\mathcal{A}$ or $\mathcal{A}^-$. If $W[i]$ is from $\mathcal{A}^-$, it means that there exists $j < i$ such that $W[j] = u(W[i])$ and $Node[j]$ and $Node[i]$ have the identical suffix of length $k-1$.

We can define a one-to-one mapping between indices i of *last* with $last[i] = 1$ and indices j of W with $W[j] \in \mathcal{A}$. As stated above, the indices i with $last[i] = 1$ have one-to-one correspondence with the nodes of the de Bruijn graph G. Consider indices j with $W[j] \in \mathcal{A}$. Let $Node'[j]$

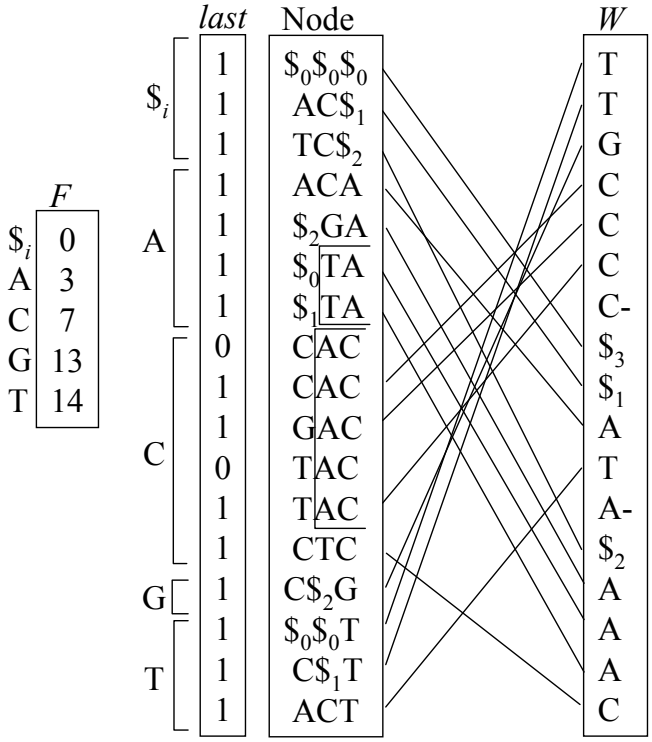


Figure 2.2: The succinct representation of the de Bruijn graph in Figure 2.1. Lines between *Node* and W show *fwd* and *bwd* functions.

denote the concatenation of the length $k-1$ suffix of $Node[j]$ and $W[j]$. For any $Node'[j]$, there exists i such that $Node[i] = Node'[j]$. Because of the definition of W, there are no indices j and j' ($j \neq j'$) such that $Node'[j] = Node'[j']$. Therefore there is a one-to-one mapping. Furthermore, the mapping is represented by *rank* and *select* queries on W. Let i, j be indices such that $last[i] = 1$ and $Node[i] = Node'[j]$. Let $c = C(i)$ be the last character of $Node[i]$ and $r = rank_1(last, i) - rank_1(last, F[c])$. Then it holds $j = select_c(W, r)$. From j, i is computed by $c = W[i]$, $r = rank_c(W, j)$ and $i = select_1(last, rank_1(last, F[c]) + r)$. We define $bwd(i) = j$ and $fwd(j) = i$. The time complexities of $bwd(i)$ and $fwd(j)$ are $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

Our data structure is similar to the XBW data structure [31] in the sense that the *last* array in ours is the same as S_{last} in the XBW. We propose a new encoding scheme for storing labels of a graph.

2.3.2 The *outdegree* and *outgoing* operations

The $outdegree(v)$ operation is easy to support. We assume that v is the index of *last* such that $last[v] = 1$ and $Node[v]$ is the label of the node. From the definition of *last*, it is obvious that $outdegree(v) = v - pred_1(last, v-1)$. The time complexity is $\mathcal{O}(t_b(m, 2))$.

The $outgoing(v, c)$ operation is done as follows. For any $1 \leq i \leq m$, we define $R(i) = [pred_1(last, i-1) + 1, succ_1(last, i)]$, which is the range of W and *last* that for all $j \in R(i)$, $Node[j]$ are identical. The labels of outgoing edges of node v are stored in $W[j]$ for $j \in R(v)$. Let j be the index such that $u(W[j]) = c$. We can find j by $pred_c(W, v)$ and $pred_{c^-}(W, v)$. Then $x = outgoing(v, c)$ can be computed by $x = fwd(j)$.

The time complexity for $outgoing(v, c)$ is $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

2.3.3 The *indegree* and *incoming* operations

Consider to compute $indegree(v)$. Let $d = C(v)$ and $x = bwd(v)$. Then it holds $d = W[x]$ and the first character of $Node[x]$ is the label of an edge pointing to v. Let $y = succ_d(W, x)$. Then all d^- between $W[x]$ and $W[y]$ correspond to parents of v. The number of such d^- is computed by *rank* on W. The time complexity is $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

To compute $incoming(v, c)$, we need to obtain the first character of $Node[i]$ such that $x \leq i < y$ and $u(W[i]) = d$. The first character of $Node[i]$ is computed by $C(b^{k-1}(i))$ where b^{k-1} stands for applying $bwd(succ_1(last, i))$ repeatedly $k-1$ times. We perform a binary search to find the index i such that $c = C(bwd^{k-1}(i))$. The time complexity is $\mathcal{O}(k(t_f + t_b(m, 2\sigma)) \log \sigma)$.

2.3.4 The *index* operation

Recall that $index(s)$ returns the index i of the node whose label is the string s of length k. Precisely, it returns i such that $last[i] = 1$ and $Node[i] = s$. The algorithm for $index(s)$ is similar to [33]. Let $i_1 < i_2 < \cdots < i_w$ be the indices such that $last[i_j] = 1$ and $Node[i_j]$ and s have the same suffix of length d ($1 \leq d \leq k$). Let i_0 be the smallest index in $R(i_1)$. Then for any i such that $i_0 \leq i \leq i_w$, $Node[i]$ and s have the same suffix of length d and for other indices this does not hold. Therefore $index(s)$ can be done by computing ranges $[i_0, i_w]$ for $d = 1, 2, \ldots, k$. Let c_d denote the d-th character of s ($1 \leq d \leq k$). For $d = 1$, the range is $[F[c_1] + 1, F[c_1 + 1]]$. Given the range $[\ell_d, r_d]$ for d, we can compute the range $[\ell_{d+1}, r_{d+1}]$ for $d + 1$ as follows. The end of the range r_{d+1} is computed by $r_{d+1} = outgoing(r_d, c_{d+1})$. The beginning of the range ℓ_d is computed by $pred_1(last, outgoing(succ_1(last, \ell_d), c_{d+1}) + 1$.

The above algorithm can be simplified. Instead of computing ranges $[i_0, i_w]$, we can use $[i_1, i_w]$. For $d = 1$, the range is $[succ_1(last, F[c_1] + 1), F[c_1 + 1]]$. Given the range $[\ell_d, r_d]$ for d,

the range for $d+1$ is obtained by $r_{d+1} = \mathit{outgoing}(r_d, c_{d+1})$ and $\ell_{d+1} = \mathit{outgoing}(\ell_d, c_{d+1})$. The time complexity is $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$.

2.3.5 Time and space complexities

We implement the above data structure for the static case using known succinct data structures. The array F is stored in $\sigma \log m$ bits. The data structure for computing $C(i)$ uses $\mathcal{O}(\sigma \log m) + \mathcal{O}(m \log\log m / \log m)$ bits. The operation time t_f is constant. The string *last* is stored in $m + o(m)$ bits so that *rank*, *select*, and *access* takes constant time [29]. The string W is stored by using [28]. Because the characters of W are from $\mathcal{A} \cup \mathcal{A}^- \cup \{\$_1, \ldots, \$_M\}$, the alphabet size is $2\sigma + M$. We can reduce the alphabet size to $2\sigma + 1$ by unifying the M terminators $\$_1, \ldots, \$_M$ into a character \$. We distinguish two terminators, but encode them using the same code.

The string W is stored in $m \log(2\sigma + M) + \mathcal{O}((\sigma + M) \log m) + o(m \log \sigma) = m + m \log \sigma + \mathcal{O}((\sigma + M) \log m) + o(m \log \sigma)$ bits, and the time complexities t_r, t_s, t_a are $\mathcal{O}(\frac{\log \sigma}{\log\log m})$. Therefore the time complexities for *outdegree*, *indegree*, *outgoing*, *incoming*, and *index* are $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{k \log^2 \sigma}{\log\log n})$, $\mathcal{O}(\frac{k \log \sigma}{\log\log n})$, respectively.

For polylog-size alphabets, *outdegree*, *indegree* and *outgoing* takes constant time, *incoming* takes $\mathcal{O}(k \log \sigma)$ time, and *index* takes $\mathcal{O}(k)$ time.

2.4 On-line construction

In this section we propose an on-line construction algorithm of the de Bruijn graph of a string. Here on-line means given the succinct de Bruijn graph G of a string $T = T[1] \cdots T[N]$, we change it to the succinct de Bruijn graph G' of the string $T' = T[1] \cdots T[N+1]$ which is made by appending a character to T.

As stated above, our succinct representation of G assumes that a character \$ is appended to the end of T. Let p be the position of \$ in W. To construct the succinct representation of G', we first change $W[p]$ from \$ to $T[N+1]$ and modify other parts if necessary, then insert \$ to another position of W. The details are as follows.

Let p be the position of \$ in W for the string $T = T[1] \cdots T[N]$. If a new character $c = T[N+1]$ is appended to the end of T, we change $W[p]$ from \$ to $T[N+1]$. We have to maintain the invariant that for all $i \in R(p)$, that is, $\mathit{Node}[i] = \mathit{Node}[p]$, $W[i]$ are distinct. Because before changing $W[p]$ they are distinct, we can check the invariant by finding the character $c = T[N+1]$ or c^- in $W[i]$ such that $i \in R(p)$. This is done by *rank* and *select* on W.

If $T[N+1]$ already exists in the range, let p' be its position. We delete $W[p]$ and $\mathit{last}[p]$ and we insert \$ in W at position $x = \mathit{fwd}(p')$. We also insert 0 in $\mathit{last}[x]$ because $\mathit{Node}[x]$ already

exists. We update $p = x$ and the array F accordingly.

If $T[N+1]$ does not exist in the range, we change $W[p] = \$$ to either $c = T[N+1]$ or c^-. To determine c or c^-, we first find the nearest occurrence of c to $W[p]$, namely, its position is $j = pred_c(W, p-1)$ if it exists ($j > 0$). We compare $Node[j]$ with $Node[p]$. If they have the same suffix of length $k-1$, we change $W[p]$ to c^-, and otherwise change $W[p]$ to c. We compare characters of $Node[j]$ and $Node[p]$ one by one using the *bwd* function. We also compare $Node[j_2]$ with $Node[p]$ where $j_2 = succ_c(W, p+1)$ if it exists ($j_2 \leq m$). If they share the length $k-1$ suffix, we change $c_2 = W[j_2]$ to c_2^-. This takes $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$ time. If the nearest c does not exist ($j = 0$), let $j = F[c]$. The position x to insert \$ is computed by $x = fwd(j)$. We insert 0 to $last[x]$ if $W[p]$ or $W[j_2]$ has a character in $\mathcal{A}^-$, or 1 otherwise. Finally we set $p = x$ and update the array F.

In total, the update operation takes $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$ time. If we use the dynamic *rank*/*select* data structure of [30] for W and *last*, $t_b = \mathcal{O}(\frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m}))$ time. We also use [30] for computing $C(i)$. Then $t_f = \mathcal{O}(\frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m}))$ and the space is $\mathcal{O}(\sigma \log n) + \mathcal{O}(m \log\log m / \log m)$ bits. Because we repeat this update operation N times for all characters of the input string, the succinct de Bruijn graph can be constructed in $\mathcal{O}\left(Nk \cdot \frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m})\right)$ time. For polylog-sized alphabets, it becomes $\mathcal{O}(Nk \cdot \frac{\log m}{\log\log m})$.

It is easy to construct the static data structure from the dynamic one. The strings *last* and W for the static one are generated by applying *access* operations to the dynamic one for $i = 1, \ldots, m$ in $\mathcal{O}(mt_b(m, 2\sigma))$ time. After constructing the static strings, the auxiliary data structures for computing *rank*/*select* are constructed in $\mathcal{O}(m)$ time.

2.5 Conclusion

We have proposed a succinct representation of de Bruijn graphs, which can be constructed with efficient time and space complexities, and in an on-line manner. Therefore they are useful for large-scale genome assembly.

The succinct de Bruijn graph can be also used for data compression. The PPM (Prediction by Partial Matching) is a text compression algorithm [34]. In the order-k PPM, a character is compressed using statistical information that it appears after a string of length k based on a given probability distribution. We can easily extend our succinct de Bruijn graph to be used for PPM compression. In addition to the array W, we use another array to store the numbers of times that each edge is traversed. Then we have enough information for compression. The succinct de Bruijn graph is used for natural language processing because it stores all n-grams in a text.

Our future work will be to improve the time complexity for the on-line construc-

tion algorithm, and to implement the proposed data structure and apply it for assembling large genomes and PPM data compression. A sample source code is available at `http://code.google.com/p/csalib/`.

Preface to Paper II

In the first paper we showed how to remove the relationship to k from the size complexity of a de Bruijn graph. However, k is still an exceedingly important factor – it determines the edges in the graph, and the edges of the graph determine the quality of the assembled output.

As there is no best k for some given data, it was common to assemble using multiple k values, and keep the best result. Soon after, iterative de Bruijn graphs were developed that would assemble with increasing values of k, using the output of the previous iteration to clean the input data at each step. This introduced a relationship to k in the time complexity of an assembly pipeline, whereby up to k graphs must be constructed, but significantly improved the assembly quality.

Since our approach used dummy edges to ensure that every base in input data would be an outgoing edge of at least one node, and all of the node strings were discarded, I wondered if we could augment our data structure to represent de Bruijn graphs of multiple k values.

The next paper introduces the first de Bruijn graph that can change k on the fly, bypassing the need to construct multiple de Bruijn graphs.

My contribution was the original concept, co-working on designing the algorithms, implementing the data structure, experimenting, writing roughly 25% of the paper, and presenting it.

Chapter 3

Variable-Order de Bruijn Graphs

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.

3.1 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) [23], *Arabidopsis* variations (1001 genomes) [35], human variations (1000 genomes) [36], and the Human Microbiome [37] 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 [10, 11], 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 [12, 13, 14, 15, 16], 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 [12] and IDBA [13] use a number of different K values. IDBA [13] 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 [12] uses a similar approach but uses all the reads at each iteration.

3.1.1 Our Contribution

SPAdes [12] and IDBA [13] 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 [38], we have implemented an external memory construction algorithm, and demonstrated the scalability of our structure on much larger data sets.

3.1.2 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. [15] 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 [21] reduced space requirements by using a sparse bitvector (by Okanohara and Sadakane [25]) 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 [22], 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 [39] developed a different succinct data structure based on the Burrows-Wheeler transform [32] 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. [40] 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. [39] is combined with ideas from IDBA-UD [41] in a metagenomics assembler called MEGAHIT [42]. In practice MEGAHIT requires more memory than competing methods but produces significantly better assemblies.

Lastly, Lin and Pevzner [43] 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.

3.1.3 Roadmap

Section 3.2 sets notation, and formally lays down the problem and auxiliary data structures we use. Section 3.3 gives details of the BOSS representation [39]. Sections 3.4 and 3.5 then describes our variable-order de Bruijn graph structure. In Section 3.6 we report on experiments comparing the practical performance of our data structure to that of a single-order de Bruijn graph. Section 5 offers directions for future work.

3.2 Preliminaries

3.2.1 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 $\mathsf{label}(v) = T[2, K+1]$.

3.2.2 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, $rank_c(S, i)$ is the number of times that c appears in $S[1, i]$, and $select_c(S, j)$ 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 [44] 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. [25]) 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.

3.2.3 Wavelet Trees

To support rank and select on larger alphabet strings, the wavelet tree [45, 46] 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. [47]), in $\mathcal{O}(\log \sigma)$ time, and we will make use of some of this functionality in Section 3.5.

3.3 BOSS representation

Conceptually, to build the BOSS representation [39] 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,

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

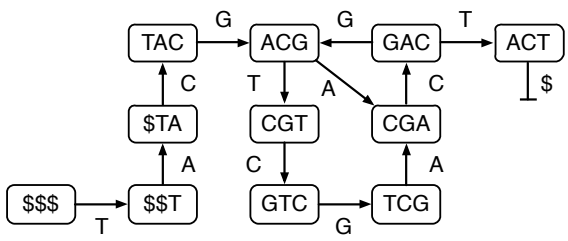


Figure 3.1: The BOSS matrix (left) and de Bruijn graph (right) for the quadruples CGAC, GACG, GACT, TACG, GTCG, ACGA, ACGT, TCGA, CGTC.

---

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. 3.1, 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 backward always returns at least one node, we can recover any non-dummy node's entire label by K calls to lastchar interleaved with $K - 1$ calls to backward.

3.4 Varying order

If we delete the first column of the matrix in Figure 3.1, 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:

- $\mathsf{shorter}(v, k)$ returns the node whose label is the last k characters of v's label;
- $\mathsf{longer}(v, k)$ lists nodes whose labels have length $k \leq K$ and end with v's label;
- $\mathsf{maxlen}(v, a)$ 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 $\mathsf{maxlen}(v, *)$. Notice shorter and longer are symmetric, in the sense that if v's label has length k_v and $x \in \mathsf{longer}(v, k_v)$, then $\mathsf{shorter}(x, k_v) = v$. In our example, $\mathsf{shorter}([4, 5], 2) = [4, 6]$ while $\mathsf{longer}([4, 6], 3) = [4, 5], [6, 6]$ and $\mathsf{maxlen}([4, 6], \mathrm{G})$ could return either $[4, 5]$ or $[6, 6]$, while $\mathsf{maxlen}([4, 6], \mathrm{T}) = [4, 5]$ and $\mathsf{maxlen}([4, 6], \mathrm{A}) = \mathrm{NULL}$.

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 $\mathsf{backward}(v)$ 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 $\mathsf{longer}(v, k_v + 1)$ 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.

3.5 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, $W = \mathrm{TCCGTGGATAA\$C}$ and $L = 1110111011111$.

Now we can implement $\mathsf{maxlen}([i,j],a)$ 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 $\mathsf{maxlen}([i,j],*)$ 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 $\mathsf{shorter}$ 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 $\mathsf{backward}$ and $\mathsf{lastchar}$ 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 $\mathsf{shorter}$ and longer, we store a range-minimum data structure [48] 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

Table 3.1: 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
$\mathsf{shorter}_1$ (μs)	N/A	14.65	N/A	17.72	N/A	19.54	N/A	19.84
$\mathsf{shorter}_2$ (μs)	N/A	14.83	N/A	17.79	N/A	19.68	N/A	19.98
$\mathsf{shorter}_4$ (μs)	N/A	15.11	N/A	18.02	N/A	19.90	N/A	20.20
$\mathsf{shorter}_8$ (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

$i' \leq i$ 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 3.1. *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.

3.6 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.

3.6.1 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 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

[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/`

Assemblathon 2[49]. The total size of this data set is 64 GB compressed on disk (245 GB uncompressed).

We used DSK [50] 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., [40]). $K = 55$ 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 3.1.

3.6.2 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.

3.6.3 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 3.6.2, 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 3.1. 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 3.1. 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 [51]). 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 ($\mathsf{shorter}_1$ 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 ($\mathsf{shorter}_1$, $\mathsf{shorter}_2$, $\mathsf{shorter}_4$, and $\mathsf{shorter}_8$ 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).

3.7 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.

Preface to Paper III

In the previous paper, we demonstrated for the first time that de Bruijn graphs using a BWT-based representation could be augmented to support additional operations and applications.

At the time, metagenomics was becoming a popular topic, with the Colored de Bruijn Graph, introduced in 2012, sitting at the center of many metagenomic tools. We wanted to see how we could extend our idea to create a succinct Colored de Bruijn Graph.

Initially, we took inspiration from Jouni Siren's 2014 paper Relative FM-Indexes, which described a way to use an FM-Index for one sequence, S_1, to provide a fully functional FM-index for another sequence, S_2, with minimal extra information. This was essentially the same as our goal for the colored de Bruijn graph, which would capitalise on the fact that most genomes in a population are very similar.

Siren's paper described how to implement a relative version of the access and rank functions, which were essential to implement a succinct de Bruijn graph, but didn't have a relative select function. We worked with Siren to describe such a function in the paper Relative Select [52], with the goal of using it to implement a succinct colored de Bruijn graph. This paper is available in Appendix A.

However, in parallel, we began experimenting with a simpler translation of the Cortex Colored de Bruijn Graph using a succinct bit matrix to store color. Cortex simply did bubble detection, which would not require the relative operations above. As a result, we only needed to represent colors using a succinct bit matrix. This simple idea yielded practical results, which are presented in the next paper.

My contribution to *Colored de Bruijn Graphs* was in identifying the opportunity, designing the data structure, assisting implementation and experimentation, and roughly 20% of the writing.

Chapter 4

Succinct Colored de Bruijn Graphs

Motivation: Iqbal et al. (Nature Genetics, 2012) introduced the *colored de Bruijn graph*, a variant of the classic de Bruijn graph, which is aimed at "detecting and genotyping simple and complex genetic variants in an individual or population". Because they are intended to be applied to massive population level data, it is essential that the graphs be represented efficiently. Unfortunately, current succinct de Bruijn graph representations are not directly applicable to the colored de Bruijn graph, which requires additional information to be succinctly encoded as well as support for non-standard traversal operations.
Results: Our data structure dramatically reduces the amount of memory required to store and use the colored de Bruijn graph, with some penalty to runtime, allowing it to be applied in much larger and more ambitious sequence projects than was previously possible.
Availability: `https://github.com/cosmo-team/cosmo/tree/VARI`

4.1 Introduction

In the 20 years since it was introduced to bioinformatics by [10], the *de Bruijn graph* has become a mainstay of modern genomics, essential to genome assembly [20, 53, 54]. The near ubiquity of de Bruijn graphs has led to a number of succinct representations, which aim to implement the graph in small space, while still supporting fast navigation operations. Formally, a de Bruijn graph constructed for a set of strings (e.g., sequence reads) has a distinct vertex v for every unique $(k-1)$-mer (substring of length $k-1$) present in the strings, and a directed edge (u, v) for every observed k-mer in the strings with $(k-1)$-mer prefix u and $(k-1)$-mer suffix v. A contig corresponds to a non-branching path through this graph. See [20] for a more thorough

explanation of de Bruijn graphs and their use in assembly.

[24] introduced the *colored de Bruijn graph*, a variant of the classical structure, which is aimed at "detecting and genotyping simple and complex genetic variants in an individual or population." The edge structure of the colored de Bruijn graph is the same as the classic structure, but now to each vertex ($(k-1)$-mer) and edge (k-mer) is associated a list of colors corresponding to the samples in which the vertex or edge label exists. More specifically, given a set of n samples, there exists a set $\mathcal{C}$ of n colors $c_1, c_2, .., c_n$ where c_i corresponds to sample i and all k-mers and $(k-1)$-mers that are contained in sample i are colored with c_i. A *bubble* in this graph corresponds to an undirected cycle, and is shown to be indicative of biological variation by [24]. Cortex, the implementation of [24], uses the colored de Bruijn graph to develop a method of assembling multiple genomes simultaneously, without losing track of the individuals from which $(k-1)$-mers (and k-mers) originated. This graph is derived from either multiple reference genomes, multiple samples, or a combination of both.

Variant information of an individual or population can be deduced from structure present in the colored de Bruijn graph and the colors of each k-mer. As implied by [24], the ultimate intended use of colored de Bruijn graphs is to apply it to massive, population-level sequence data that is now abundant due to next generation sequencing technology (NGS) and multiplexing. These technologies have enabled production of sequence data for large populations, which has led to ambitious sequencing initiatives that aim to study genetic variation for agriculturally and bio-medically important species. These initiatives include the *Genome 10K* project that aims to sequence the genomes of 10,000 vertebrate species [23], the *iK5* project [55], the 150 Tomato Genome ReSequencing project [56, 57], and the 1001 Arabidopsis project, a worldwide initiative to sequence cultivars of *Arabidopsis* [58]. Hence, the succinct colored de Bruijn graph is applicable in the context of these projects, in that it can assist in variation discovery within a species by analyzing all the data in these projects at once.

In addition to species-specific initiatives, scientific and regulatory agencies are showing increased interest in shotgun metagenomic sequences for public health purposes [59, 60], specifically monitoring for antimicrobial resistance (AMR) [61, 62]. AMR is considered one of the top public health threats, with fears that the spread of AMR will lead to increased morbitiy and mortality for many bacterial illnesses [63, 64]. AMR occurs when bacteria express genetic elements that render them impervious to antibiotic treatments. Importantly, these genetic resistance elements can be exchanged between distantly-related bacteria via multiple genetic mechanisms, which makes AMR an inherently population-level phenomenon [65]. Shotgun metagenomic sequencing allows access to the entire microbial population in a sample (the "metagenome"), which is of immense value for tracking and understanding the evolution of resistance elements within and across diverse bacteria[66]. This metagenomics approach to AMR surveillance has

been applied in both human and agricultural settings [67, 68], generating hundreds of samples with terabytes of sequence data for relatively small studies. Given the large number of samples and large size of sequence data involved in these whole-genome and metagenomic projects, it is imperative that the colored de Bruijn graph can be stored and traversed in a space- and time-efficient manner.

Our Contribution We develop an efficient data structure for storage and use of the colored de Bruijn graph. Compared to Cortex, the implementation of [24], our new data structure dramatically reduces the amount of memory required to store and use the colored de Bruijn graph, with some penalty to runtime. We demonstrate this reduction in memory through a comprehensive set of experiments across the following three datasets: (1) four plant genomes, (2) 3,765 *Escherichia coli* assemblies, and (3) 87 sequenced metagenomic samples from commercial beef production facilities. We show our method, which we refer to as Vari (Finnish for color), has better peak memory usage on all these datasets. Our plant reference genomes dataset required 101 GB of RAM for Cortex to represent while Vari required only 4 GB. And our largest two datasets contain too many k-mers and colors for Cortex's data structure to represent in the 512 GB of RAM available on our bioinformatics servers. Vari is a novel generalization of the succinct data structure for classical de Bruijn graphs due to [39], which is based on the Burrows-Wheeler transform of the sequence reads, and thus, has independent theoretical importance.

In addition to demonstrating the memory and runtime of Vari, we validate its output using the *E.coli* reference genome and a simulated variant.

Related Work As noted above, maintenance and navigation of the de Bruijn graph is a space and time bottleneck in genome assembly. Space-efficient representations of de Bruijn graphs have thus been heavily researched in recent years. One of the first approaches was introduced by [15] 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 (>38x depth paired-end reads from Illumina Genome Analyzer II, HapMap: NA18507[1]).

[21] reduced space requirements by using a sparse bitvector (by [25]) to represent the k-mers (the edges), and used rank and select operations (to be described later) to traverse it. As a result, their representation took 32 GB for the same data set. Minia, by [22], 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

[1] `https://www.ncbi.nlm.nih.gov/sra/?term=SRA010896`

reduced to 5.7 GB on the same dataset. Contemporaneously, [39] developed a different succinct data structure based on the Burrows-Wheeler transform [32] that requires 2.5 GB. The data structure of [39] is combined with ideas from IDBA-UD [41] in a metagenomics assembler called MEGAHIT [42]. In practice MEGAHIT requires more memory than competing methods but produces significantly better assemblies. [40] implemented the de Bruijn graph using an FM-index and *minimizers*. Their method uses 1.5 GB on the same NA18507 data. [69] released the Bloom Filter Trie, which is another succinct data structure for the colored de Bruiin graph; however, we were unable to compare our method against it since it only supports the building and loading of a colored de Bruijn graph and does not contain operations to support our experiments. SplitMEM [70] is a related algorithm to create a colored de Bruijn graph from a set of suffix trees representing the other genomes. Lastly, Lin et al. [71] point out the similarity between the breakpoint graph, which is traditionally viewed as a data structure to detect breakpoints between genome rearrangements, and the colored de Bruijn graph.

Roadmap In the next section, we describe our succinct colored de Bruijn graph data structure, generalizing the stucture for classic de Bruijn graphs presented by [39]. Section 4.3 then elucidates the practical performance of the new data structure, comparing it to Cortex. Section 5 offers some concluding remarks.

4.2 Methods

Our data structure for colored de Bruijn graphs is based on the succinct representation of individual de Bruijn graphs introduced by [39]—which we refer to as the BOSS representation from the authors' initials—so we start by describing that representation. We note that BOSS is itself a generalization of FM-indexes [33] obtained by extending the Burrows-Wheeler transform (BWT) from strings to the multisets of edge-labels of de Bruijn graphs. We then give a general explanation of how we add colors, and finally give details of our implementation.

4.2.1 BOSS Representation

Consider the de Bruijn graph $G = (V, E)$ for a set of k-mers, with each k-mer $a_0 \cdots a_{k-1}$ representing a directed edge from the node labelled $a_0 \cdots a_{k-2}$ to the node labelled $a_1 \cdots a_{k-1}$, with the edge itself labelled a_{k-1}. Define the nodes' co-lexicographic order to be the lexicographic order of their reversed labels. Let F be the list of G's edges sorted co-lexicographically by their ending nodes, with ties broken co-lexicographically by their starting nodes (or, equivalently, by their k-mers' first characters). Let L be the list of G's edges sorted co-lexicographically by their starting nodes, with ties broken co-lexicographically by their ending nodes (or, equivalently, by

their own labels). If two edges e and e' have the same label, then they have the same relative order in both lists; otherwise, their relative order in F is the same as their labels' lexicographic order. Defining the edge-BWT (EBWT) of G to be the sequence of edge labels sorted according to the edges' order in L, so $\mathsf{label}(L[h]) = \mathsf{EBWT}(G)[h]$ for all h, this means that if e is in position p in L, then in F it is in position

$$|\{d \,:\, d \in E,\ \mathsf{label}(d) \prec \mathsf{label}(e)\}| + \mathsf{EBWT}(G).rank_{\mathsf{label}(e)}(p) - 1\,,$$

where $\mathsf{EBWT}(G).rank_{\mathsf{label}(e)}(p)$ is the number of times $\mathsf{label}(e)$ appears in $\mathsf{EBWT}(G)[1,p]$. It follows that if we have, first, an array storing $|\{d \,:\, d \in E,\ \mathsf{label}(d) \prec c\}|$ for each character c and, second, a fast rank data structure on $\mathsf{EBWT}(G)$ then, given an edge's position in L, we can quickly compute its position in F.

Let B_F be the bitvector with a 1 marking the position in F of the last incoming edge of each node, and let B_L be the bitvector with a 1 marking the position in L of the last outgoing edge of each node. Given a character c and the co-lexicographic rank of a node v, we can use B_L to find the interval in L containing v's outgoing edges, then we can search in $\mathsf{EBWT}(G)$ to find the position of the one e labelled c. We can then find e's position in F, as described above. Finally, we can use B_F to find the co-lexicographic rank of e's ending node. With the appropriate implementations of the data structures, we can store G in $(1 + o(1))|E|(\lg \sigma + 2)$ bits, where σ is the size of the alphabet (i.e., 4 for DNA), such that when given a character c and the co-lexicographic rank of a node v, in $\mathcal{O}(\log \log \sigma)$ time we can find the node reached from v by following the directed edge labelled c, if such an edge exists.

If we know the range $L[i..j]$ of k-mers whose starting nodes end with a pattern P of length less than $(k-1)$, then we can compute the range $F[i'..j']$ of k-mers whose ending nodes end with Pc, for any character c, since

$$\begin{aligned} i' &= |\{d \,:\, d \in E,\ \mathsf{label}(d) \prec c\}| + \mathsf{EBWT}(G).rank_c(i-1) \\ j' &= |\{d \,:\, d \in E,\ \mathsf{label}(d) \prec c\}| + \mathsf{EBWT}(G).rank_c(j) - 1\,. \end{aligned}$$

It follows that, given a node v's label, we can find the interval in L containing v's outgoing edges in $\mathcal{O}(k \log \log \sigma)$ time, provided there is a directed path to v (not necessarily simple) of length at least $k-1$. In general there is no way, however, to use $\mathsf{EBWT}(G)$, B_F and B_L alone to recover the labels of nodes with no incoming edges.

To prevent information being lost and to be able to support searching for any node given its label, Bowe et al. add extra nodes and edges to the graph, such that there is a directed path of length at least $k-1$ to each original node. Each new node's label is a $(k-1)$-mer that is

prefixed by one or more copies of a special symbol \$ not in the alphabet and lexicographically strictly less than all others. Notice that, when new nodes are added, the node labelled $\$^{k-1}$ is always first in co-lexicographic order and has no incoming edges. Bowe et al. also attach an extra outgoing edge labelled \$, that leads nowhere, to each node with no original outgoing edge. The edge-BWT and bitvectors for this augmented graph are, together, the BOSS representation of G.

4.2.2 Adding Color

We cannot represent the colored de Bruijn graph for a multiset $\mathcal{G} = \{G_1, \ldots, G_t\}$ of individual de Bruijn graphs satisfactorily by simply representing each individual graph separately, for two reasons: first, the memory requirements would quickly become impractical and, second, we should be able to answer efficiently queries such as "which individual graphs contain this edge?" Therefore, we set G to be the union of the individual graphs and build the BOSS representation only for G. As long as most of the k-mers are common to most of the individual graphs, the memory needed to store G is comparable to that need to store an individual graph.

To indicate which edges of G are in which individual graphs, we build and store a two-dimensional binary array C in which $C[i, j]$ indicates whether the ith edge in G is present in the jth individual de Bruijn graph (i.e., whether that edge has the jth color). (Recall from the description above of BOSS that we consider the edges in G to be sorted lexicographically by the reversed labels of their starting nodes, with ties broken lexicographically by their own single-character labels.) If the individual graphs are sufficiently similar, then we can compress C effectively and store it in such a way that we can still access its individual bits quickly and support fast rank and select queries on the rows. (A *select* query on the ith row takes an argument r and returns the index j of the rth individual graph that contains the ith edge in G.) In the next subsection we give details of some relatively simple compression strategies that support fast access, rank and select. With these data structures, we can navigate efficiently in any of the individual graphs and switch between them. For example, we can efficiently check whether an edge has a particular color (with an access), count the number of colors it has (with a *rank* query) or list them (with repeated *select* queries). We have not yet considered more sophisticated compression schemes that could still offer fast queries while taking advantage of, e.g., correlations among the variations or grouping of the individual graphs by subpopulation.

Figure 4.1 shows an example of how we represent a colored de Bruijn graph consisting of two individual de Bruijn graphs. Suppose we are at node `ACG` in the graph, which is the co-lexicographically eighth node. Since the eighth 1 in B_L is $B_L[10]$ and it is preceded by two 0s, we see that `ACG`'s outgoing edges' labels are in $\mathsf{EBWT}[8..10]$, so they are `A`, `C` and `T`. Suppose we

Figure 4.1: **Left:** A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in red and blue. (We can consider all nodes to be present in both graphs, so they are shown in purple.) **Center:** The nodes sorted into co-lexicographic order, with each node's number of incoming edges shown on its left and the labels of its outgoing edges shown on its right. The edge labels are shown in red or blue if the edges occur only in the respective graph, or purple if they occur in both. **Right:** Our representation of the colored de Bruijn graph: the edge-BWT and bitvectors for the BOSS representation for the union of the individual graphs, and the binary array C (shown transposed) whose bits indicate which edges are present in which individual graphs.

want to follow the outgoing edge e labelled `C`. We see from $C[9, 0..1]$ (i.e., the tenth column in C^{T}) that e appears in the second individual graph but not the first one (i.e., it is blue but not red). There are four edges labelled `A` in the graph and three `C`s in $\mathsf{EBWT}(G)[0..9]$, so e is $F[6]$. (Since edges labelled `$` have only one end, they are not included in L or F.) From counting the 1s in $B_F[0..6]$, we see that e arrives at the fifth node in co-lexicographic order that has incoming edges. Since the first node, `$$$`, has no incoming edges, that means e arrives at the sixth node in co-lexicographic order, `CGC`.

4.2.3 Implementation

We now give some details of how our data structure is implemented and constructed in practice.

4.2.3.1 Data Structure

The arsenal of component tools available to succinct data structures designers has grown considerably in recent years [72], with many methods now implemented in libraries. We chose to make heavy use of the succinct data structures library (SDSL)[2] in our implementation.

$\mathsf{EBWT}(G)$, the sequence of edge labels, is encoded in a wavelet tree, which allows us to perform fast rank queries, essential to all our graph navigations. The bitvectors of the wavelet tree and the B bitvector are stored in the Raman-Raman-Rao (RRR) encoding [29]. The rows of the color matrix, C, are concatenated (i.e. C is stored in row-major order) and this single long

[2] `https://github.com/simongog/sdsl-lite`

bit string is then compressed. It is either stored with RRR encoding, or alternately Elias-Fano encoding [25, 73, 74] which supports online construction. Online construction is important for datasets where C is too large to fit in memory in uncompressed form, such as our metagenomic sample dataset. These encodings reduce the size of C considerably because we expect rows to be very sparse and both encodings exploit this sparseness.

4.2.3.2 Construction

In order to convert the input data to the format required by BOSS (that is, in correct sorted order, including dummy edges and bit vectors), we use the following process. We take care to ensure only subsets of data are needed in RAM at any one time during construction.

Our construction algorithm takes as input the set of (k-mer, color-set) pairs present in the input sets of reads, or alternately, k-mer counts for each color which we convert to the former ourselves. Here, color-set is a bit set indicating which samples the k-mer occurs in. We provide the option to use the Cortex frontend to generate the (k-mer, color-set). Unfortunately, this also limits the datasets to those that would run through Cortex. To overcome this, we provide the option to use a list of KMC2 [75] sorted k-mer counts as input. With this option, the k-mers from each k-mer count file in native KMC2 binary format are streamed through a priority queue to produce the union of all k-mer sets; initially one k-mer from each file is tagged with which file it originated from, and the (k-mer, file ID) pair is added to the queue. The priority queue ensures the lexicographically smallest k-mer instances across all files can be popped off the queue consecutively. All of the k-mer count files contributing a particular k-mer value have their corresponding color recorded as '1' bits in the bit set for that k-mer. Both the k-mer and the bit set are then appended to vectors which optionally are allocated in external memory using the STXXL[3] library. As each k-mer is popped off the queue, another k-mer is added to the queue to take the old k-mer's place (i.e. using the file identified by the popped k-mer's tag). This process continues until all files are read in their entirety. By both streaming data from the source files and streaming it to the external vectors, only a small amount of the data need exist in memory at a time; the priority queue will only contain the number of samples and only one row of the color matrix needs to exist in memory before being written out to disk.

After constructing the initial union set of k-mers and their corresponding color rows, BOSS construction mostly continues as originally described by Bowe *et al.*. The changes from the original construction algorithm are that most of the data optionally resides in external memory and the rows of the color matrix are permuted with their corresponding k-mers as they are sorted. For each of the k-mers we generate the reverse complement (giving it the same color-set

[3] `http://stxxl.sourceforge.net/`

as its twin). Then, for each k-mer (including the reverse complements), we sort the (k-mer, color-set) pairs by the first $k-1$ symbols (the source node of the edge) to give the F table (from here, the colors are moved around with rows of F, but otherwise ignored until the final stage). Independently, we sort the k-mers (without the color-sets) by the last $k-1$ symbols (the destination node of the edge) to give the L table.

With F and L tables computed, we calculate the set difference $F - L$ (comparing only the $(k-1)$-length prefixes and suffixes respectively), which tells us which nodes require incoming dummy edges. Each such node is then shifted and prepended with \$ signs to create the required incoming dummy edges ($k-1$ each). These incoming dummy edges are then sorted by the first $k-1$ symbols. Let this table of sorted dummy edges be D. Note that the set difference $L - F$ will give the nodes requiring outgoing dummy edges, but these do not require sorting, and so we can calculate it as is needed in the final stage.

Finally, we perform a three-way merge (by first $k-1$ symbols) D with F, and $L - F$ (calculated on the fly). For each resulting edge, we keep track of runs of equal $k-1$ length prefixes, and $k-2$ length suffixes of the source node, which allows us to calculate the B_F and B_L bit vectors, respectively. Next, we write the bit vectors, symbols from last column, and count of the second to last column to a packed file on disk, and the colors to a separate file. The color file is then either buffered in RAM and RRR encoded or optionally streamed from disk and then Elias-Fano encoded online (i.e. only the compressed version is ever resident). The time bottleneck in the above process is clearly in sorting the D and F tables, which are of the same size, and are made up of elements of size $O(k)$. Thus, overall, construction of the data structure takes $O(k(|F| \log |F|))$ time.

4.2.3.3 Traversal

We implemented two traversal methods based on those of Cortex with a modification in light of our intention to apply Vari to metagenomic reads looking for AMR gene presence.

The first, *bubble calling*, is a simple algorithm to detect sequence variation in genomic data. It consists of iterating over a set of k-mers in order to find places where bubbles start and terminate. When combined with the k-mer color (in a colored de Bruijn graph), this enables identification of places where genomic sequences diverge from one another. The differing region of the two sequences will form the two arms of a bubble, each colored with only one of the two sequence's colors. A bubble is identified when a vertex has two outgoing edges. Each edge is followed in turn to navigate a non-branching path until reaching a vertex with two incoming edges. If the terminating vertex is the same for both paths, we call this a bubble. Colors for the bubbles are determined by looking at the color assignment of the corresponding (k)-mers.

Our implementation in VARI closely follows the pseudocode given by [24].

CORTEX's traversal algorithms were designed for single isolates. For the beef safety experiments, which use metagenomic samples, we implemented a traversal inspired by CORTEX's *path divergence* algorithm. In the original CORTEX path divergence algorithm, bubbles are identified where a user-supplied reference sequence prescribes a walk through a (possibly tangled) sections of the graph in one arm of a bubble while the alternative arm must be branch free. This branch free requirement on the second arm could be a problem for metagenomic data. Due to the presence of tangle inducing homologous genomes and risk of inferring erroneous, chimeric sequences (which comprise reads from a mix of genomes in the sample), variant detection in metagenomic data is more complex. In the absence of a simple metagenomic-aware traversal algorithm, we implemented a variation of the path divergence algorithm which addresses a simpler problem, primarily for the purpose of measuring performance. This algorithm uses a reference guided approach and allows us to measure the memory footprint at traversal time as well as the time savings of not traversing the entire dataset. For this purpose, we focus specifically on the presence of AMR genes (our reference sequence) rather than variants of those genes; in our derived algorithm we ignore sample path segments leading away from and returning to the AMR gene path. This avoids some of the problems with tangles, incomplete coverage, or read errors. Thus as we traverse the gene path, we simply count the number of samples in each sample group that color the current edge. We note that keeping C in row major order allows us to compute this count in constant time as the difference between two *rank* queries.

4.3 Results

We evaluated VARI performance on three different datasets, described below. For this evaluation, we compare peak memory, which was measured as the maximum resident set size, and CPU core time, measured as the user+system process time as our metrics. In addition to evaluating performance, we also validated VARI by the ability to correctly call bubbles known to be present in a simulated dataset.

Our software supports a variety of options. It can consume k-mer counts from either Cortex's binary files or KMC2. For all experiments, we use the KMC2 flow because using Cortex as a front end limits designs to only those that would fit in memory with Cortex. Next, our software can compress the color matrix using either RRR or Elias-Fano encodings. The SDSL-light implementation allows the color matrix to be compressed in an on-line fashion only using the Elias-Fano encoding. This allows us to process larger designs, as the uncompressed matrix need never fit in RAM, and thus we use this option for all experiments. Finally, STXXL (which holds temporary vectors during data structure construction) allows using internal or external

Table 4.1: Characteristics of our datasets. The *E. coli* dataset represents 3,765 strains and hence only summary statistics for size and GC content are given. Accession numbers for this dataset as well as download procedure can be found in assembly_summary.txt as discussed in the main text.

Name	Accession Numbers	Aprox. Size	GC Content
Plant Species	Rice (NC_008394 to NC_008405)	430 Mbp	43.42%
	Tomato (NC_015438 to NC_015449)	950 Mbp	43.42%
	Corn (NC_024459 to NC_024468)	2.07 Gbp	35.70%
	Arabidopsis (NC_003070 to NC_003076)	135 Mbp	47.4%
E. coli strains	N/A	avg=5.1 Mbp min=2.9 Mbp max= 7.7 Mbp	50.5%
Beef safety	PRJNA292471	N/A	44.3%

memory. Again, we used the more scalable external memory option for all experiments. All experiments were performed on a machine with AMD Opteron model 6378 processors, having 512 GB of RAM and 64 cores.

4.3.1 Datasets

The three different datasets were chosen in order to test and evaluate the performance of Vari on a variety of diverse yet realistic data types that are likely to be used as input into Vari. For the first two datasets which comprise single isolates, we use preassembled genomes. Assembly serves to try correct sequencing errors which could otherwise falsely be detected as variants. To this end, Cortex includes its own optional data cleaning operations. However, by using instead the output of third party assembly software we can compare the colored de Bruijn graph performance on identical graphs. Characteristics about these datasets are provided in Table 4.1.

Our first performance dataset comprises reference genomes for four different plant species: *Oryza sativa Japonica* (rice)[4][76], *Solanum lycopersicum* (tomato)[5][56, 57], *Zea mays* (corn)[6][77], and *Arabidopsis thaliana* (Arabidopsis)[7][78]. This represents a sufficiently large dataset for comparing the performance of Vari with Cortex.

[4] `http://rice.plantbiology.msu.edu/annotation_pseudo_current.shtml`

[5] `ftp://ftp.solgenomics.net/tomato_genome/assembly/build_2.50/SL2.50ch00.fa.tar.gz`

[6] `ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/005/GCF_000005005.1_B73_RefGen_v3/GCF_000005005.1_B73_RefGen_v3_genomic.fna.gz`

[7] `ftp://ftp.ensemblgenomes.org/pub/plants/release-34/fasta/arabidopsis_thaliana/dna/`

Table 4.2: Data structure construction performance measurements. CPU time is user plus system time as reported by '/bin/time'. (Internal) memory is reported in megabytes and is the maximum resident set size. KMC2 includes both counting and sorting k-mers. VARI-dBG forms the k-mer union and builds the succinct de Bruijn graph. VARI-C compresses the color matrix.

	CORTEX		KMC2		VARI-dBG			VARI-C	
Dataset	**CPU time**	**Mem.**	**CPU time**	**Mem.**	**CPU time**	**Int. Mem.**	**Ext. Mem.**	**CPU time**	**Mem.**
Plants	2h 25m 27s	109,579	19m 50s	4,335	1h 34m 37s	5,388	156,504	3m 09s	3,528
E. coli (k=32)	N/A	N/A	3h 15m 40s	104	9h 30m 11s	126,777	319,328	53m 54s	42,043
E. coli (k=48)	N/A	N/A	4h 35m 29s	149	10h 47m 46s	128,077	427,460	1h 02m 07s	42,100
E. coli (k=64)	N/A	N/A	5h 05m 27s	189	11h 21m 08s	127,523	522,576	1h 09m 07s	42,134
Beef safety	N/A	N/A	34h 04m 46s	11,688	82h 42m 48s	109,091	4,378,840	6h 44m 12s	217,705

Our second performance dataset consists of the set of all 3,765 NCBI GenBank assemblies[8][9] having the organism_name field equal to "Escherichia coli" as of March 22, 2016. To evaluate the effects of varying k-mer size, we ran this dataset with $k = 32, 48, 64$. The union of all assemblies contains 158,501,209 k-mers for k=32, 205,938,139 k-mers for k=48, and 251,764,413 k-mers for k=64. The minimum, maximum, and average assembly lengths are 2,911,360 bp, 7,687,202 bp, and 5,156,744 bp, respectively.

Our third performance dataset consists of 87 metagenomic samples[10] taken at various timepoints during the beef production process from eight pens of cattle in two beef production facilities by [67]. Sequentially, these timepoints were feedlot arrival, feedlot exit, slaughter transport truck, slaughter holding, and slaughter trimmings and sponges. Sample reads were preprocessed using trimmomatic v0.36 by Bolger *et al.* [79]. Although further assembly or error correction would have been possible, it would reduce the biological variation which may be useful for some queries. Furthermore, building the data structure on uncorrected data better stresses our representation method. Samples were then arranged into groups based on the sample timepoints. The original study used these samples to demonstrate the advantages of shotgun metagenomic sequencing in tracking the evolution of antimicrobial resistance longitudinally within a complex environment like beef production; the results suggested that selective pressures occurred within the feedlot, but that slaughter safety measures drastically reduced both bacterial and AMR levels. In addition to the metagenomic samples, we included 4,062 AMR genes from the previously mentioned gene databases[11]. 23 genes in the databases containing IUPAC codes other than the four bases were filtered out as KMC2 and the succinct de Bruijn graph were configured with a

[8] `ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt`
[9] `https://www.ncbi.nlm.nih.gov/genome/doc/ftpfaq/`
[10] `https://www.ncbi.nlm.nih.gov/bioproject/292471`
[11] `https://meg.colostate.edu/MEGaRes/`

four symbol alphabet. Because we have the reference to guide the traversal, all AMR genes were combined into a single color. By combining AMR genes, the uncompressed color matrix that exists on disk during sorting and as intermediate file is much smaller (still occupying 1.2 TB), thus accelerating the permutation during construction and reducing the external memory and disk space requirements. The union of all samples and genes contains 40,995,794,366 32-mers and the GC content is 44.3%. While our server has enough RAM to represent a dataset with twice the memory footprint, this dataset nearly exhuasted the approximately 10 TB of disk space available when intermediate files were preserved. Thus this dataset is on the order of the upper limit for VARI in practice.

Finally, for validation purposes, we generated a dataset[12] comprising two genomes: (1) *E. coli* K-12 substraing MG 1655 reference genome, and (2) a copy of the reference genome to which we applied various simulated mutations. We simulated mutations by choosing 100 random loci and either inserting, deleting, or replacing a region of random length ranging from 200-500 bp. For each mutation locus, we record the flanking regions and the two variants (original reference and simulated) as a ground truth bubble.

4.3.2 Time and Memory Usage

To measure VARI's resource use and compare with CORTEX by [24] where possible, we constructed the colored de Bruijn graph for the plant dataset, the *E. coli* assembly dataset and the beef safety dataset. Construction time and memory is detailed in Table 4.2. We performed *bubble calling* on the first two and recorded peak memory usage and runtime. Direct resource comparison with CORTEX was only possible on the smallest dataset, as the largest two have too many k-mers and colors to fit in memory on our machine with CORTEX. Based on the data structure defined in CORTEX's source as well as the supplementry information provided by Iqbal *et al.*, it would have required more than 3 TB of RAM and more than 18 TB of RAM for its hash table entries alone, respectively.

In order to test query performance characteristics, various experiments were performed on all three performance datasets described in the previous subsection. Datasets varied in the number of k-mers in the graph from 158 million to over 40 billion, the number of colors, from 4 to 3,765, and degree of homology from disperate plants to the single *E. coli* species. This diversity shows the space savings achievable when the population is largely homologous, as is the case with the *E. coli* dataset, where the graph component is relatively small, in contrast to the plant dataset, where the graph component is relatively large. As can be seen in Table 4.3, where directly comparable, VARI used an order of magnitude less than the peak memory

[12] `https://github.com/cosmo-team/cosmo/tree/VARI/experiments/ecoli_validation`

Table 4.3: Comparison between the peak memory and time usage required to store all the k-mers and run bubble calling on the data in Cortex and Vari. The peak memory is given in megabytes (MB) or gigabytes (GB). The running time is reported in seconds (s), minutes (m), and hours (h). The succinct de Bruijn graph and compressed color matrix components of the memory footprint are listed in parenthesis as sdBG and sC, respectively.

			Cortex		Vari	
Dataset	**No. of k-mers**	**Colors**	**Memory**	**Time**	**Memory**	**Time**
Plants (k=32)	1,709,427,823	4	100.93 GB	2h 18m	3.53 GB (sdBG=0.89 GB, sC=1.95 GB)	32h 39m
E. coli (k=32)	158,501,209	3,765	N/A	N/A	42.17 GB (sdBG=0.09 GB, sC=38.35 GB)	3h 57m
E. coli (k=48)	205,938,139	3,765	N/A	N/A	42.26 GB (sdBG=0.11 GB, sC=38.42 GB)	4h 38m
E. coli (k=64)	251,764,413	3,765	N/A	N/A	42.32 GB (sdBG=0.13 GB, sC=38.45 GB)	5h 28m
Beef safety (k=32)	40,995,794,366	88	N/A	N/A	245.54 GB (sdBG=27.08 GB, sC=200.34 GB)	N/A

that Cortex required but required greater running time. This memory and time trade-off is important in larger population level data. This is highlighted by our largest two datasets which could not be run with Cortex. Hence, lowering the memory usage in exchange for higher running time deserves merit in contexts where there is data from large populations.

4.3.3 Validation on Simulated *E. coli*

We ran the implementations of bubble calling from both Vari and Cortex, using k=32 on the simulated *E. coli* dataset. Both tools reported the same set of 223 bubbles, 55 of which were in the ground truth set. This ensures our software faithfully implements the original data handling capabilities of Cortex. For biological implications of colored de Bruijn graph variant calls and in particular with parameter choices such as k see Iqbal *et al.* [24].

4.3.4 Observations on Beef Safety Dataset

While the beef safety dataset was primarily used for measuring the scalability of Vari and to determine if representing a dataset of this type and size was possible, we used Vari to additionally make observations about the presence of AMR genes in the beef production dataset. As previously described, during our path divergence derived algorithm, we compute a count of how many k-mers in each AMR gene are found across all samples within a sample group. This algorithm need only traverse the AMR genes, so despite the size of the overall dataset, it only took 20 minutes to load and access the necessary parts of the data structure. In contrast, if bubble calling were to run at the same rate for this dataset as for the *E. coli* assembly dataset, it would take 3,001 hours to complete, thus suggesting value in a targeted inquiry approach on datasets of this size.

Since longer genes have more k-mers, the counts are likely to be larger, as are those from

larger sample groups. To make these counts comparable, we normalize by both gene length and sample group size. We can then examine the number of genes having a disproportionately large ($>$ 3 std. dev. above mean) shared k-mer count for each gene and sample group combination. The number of such genes with disproportionately large normalized counts in each sample group were: feedlot arrival - 304, feedlot exit - 93, transport truck - 230, slaughter holding - 16, and slaughter trimmings and sponges - 0. This observation supports the conclusion of [67], namely, that antimicrobial interventions during slaughter were effective in reducing AMR gene presence in the trimmings and sponge samples, which represent the finished beef products just before they are shipped to retail outlets for human consumption.

4.4 Conclusion

We presented VARI, which is an implementation of a succinct colored de Bruijn graph that significantly reduces the amount of memory required to store and use the colored de Bruijn graph. In addition to the memory savings, we validated our approach using *E coli.* Moreover, we introduced the use of colored de Bruijn graph for accurately identifying the presence of AMR genes within metagenomic samples, which is an important advance as public health officials increasingly move towards a metagenomic sequence-based approach for surveillance and identification of resistant bacteria [61, 62, 64]. Possible nontrivial extensions to our work include (1) (1) using multi-threading to speed up the bubble calling, (2) compressing the color array C more effectively by taking advantage of correlations among the variations, and (3) applying more sophisticated approaches to metagenomic data.

Chapter 5

Conclusion

In this thesis, we proposed the first Burrows–Wheeler based representation of de Bruijn graphs, a linchpin in bioinformatic pipelines. This reduced the memory requirements by an order of magnitude, enabling basic DNA assembly to be performed on a laptop.

While there are competing representations with similar memory benefits, the ordered nature of the Burrows–Wheeler based representation enabled augmentation using additional data structures to support more powerful operations.

In the case of the variable order de Bruijn graph, by including the longest common suffix array, stored using a wavelet tree, we can quickly change the value of k on the fly. This is the first data structure to support this, and can be used to improve genome assembly quality, while taking only 3.5 times the space and 30% longer to construct than a single de Bruijn graph, and avoiding building multiple de Bruijn graphs like previous approaches would take.

In the case of the Colored de Bruijn graph, we utilised a compressed bit matrix to store color information in the same order as the edges. This allowed us to implement the algorithms described in [24] in almost two orders of magnitude less space, ultimately enabling this analysis for datasets that were previously unsupported. Finally, we demonstrated its practical use by applying it to locating bacterial outbreaks in food supply chains.

After publishing these papers, some open questions remained:

1. The Variable-Order de Bruijn Graph described how to change the value of k on the fly. This could in theory support any procedure utilising multiple k values during traversal, but the exact methodology was left unanswered. Using the Variable-Order de Bruijn graph, how should order change operations be applied to improve the quality of the assembly? How much could the quality be improved?

2. A method using Range Minimum Query data structures to support varying k was also

suggested, which would take a performance hit, yet reduce the size. However, experimental results were only given for the implementation using a Wavelet Tree. It would be interesting to see how they compare in practice.

3. The Colored de Bruijn Graph becomes more powerful when built with data from larger populations. This is good news, as sequence data is being produced at an increasing rate. However, as the graph is a static data structure, it must be rebuilt using the original k-mers, and new k-mers, whenever new sequence data is added. There is ongoing research into merging Burrows–Wheeler transforms – could this be utilised to allow updating a Colored de Bruijn Graph?

4. Our color matrix implementation was rudimentary, using only a general purpose succinct bit vector data structure. It should be possible to improve the compression by using an approach specifically suited to the task.

Since the papers in this thesis were published, there has been a number of papers published which further explored the power of the Burrows–Wheeler based representation, including addressing the above questions.

Belazzougui et al. were able to improve the complexity of variable-order backward traversal, which was the slowest of our proposed operations [80]. This would allow it to be practical to store only one of the reverse complements of the DNA, in order to reduce the memory it takes, while taking a smaller speed penalty.

Díaz-Domínguez et al. describe a variant of the variable-order graph, called the hidden-order de Bruijn graph, replacing the LCP array with a succinct cartesian tree, which reduced the memory requirements, and allowed them to develop a scheme to extract remarkably long contigs, which are more informative than what would be available in a simple fixed-k de Bruijn graph [81]. Later, Díaz-Domínguez et al. further augmented this to simulate a complete DNA string graph (briefly described in the introduction of this thesis), unlocking substantially longer contigs, still [82].

Muggli et al. presented a merging algorithm, enabling two Colored de Bruijn graphs to be combined into one – we could now re-use existing graphs while gathering new sequence data [83]. This was shortly after improved on by Egidi et al., who reduced the working space by half [84]. There has also been research into improving the compression scheme for the color matrix [85, 86].

There have also been a number of papers which augment the succinct de Bruijn graph with other additional information, such as long range distance information [87] and optical maps [88], providing more positional information to better disambiguate contigs.

Other research has included making the de Bruijn graph fully dynamic [89, 90], and applying colored de Bruijn graphs to real-time search of increasingly large bacterial and viral genomic databases [91]. A review is also available in [92].

By introducing a simple, customizable, succinct representation of the all-important de Bruijn graph, we are now able to better analyze increasingly large datasets. This has the potential to make research much easier, which will deepen our understanding of DNA, allowing us to develop novel treatments for illnesses.

Acknowledgements

First and foremost, I could not have completed my PhD without the sponsorship of Soukendai and NICTA. I'd also like to thank the following individuals:

- **Justin Zobel** – You helped me secure funding to start my PhD, and introduced me to the de Bruijn graph and DNA assembly. Without your help, I may not have started a PhD. You also taught many of the professors I had in University, so I owe a lot to you transitively speaking, too.
- **Kathryn Holt** – Thank you for spending countless hours teaching me about biology in your office. Coming from a school that taught creationism, this was mind expanding, and incredibly influential on the way I would think in general.
- **Tom Conway** – You kicked off the succinct de Bruijn graph subfield, and took the time to help me understand your paper over several lunches.
- **Simon Puglisi** – Thank you for introducing me to succinct data structures. I think I've always done my best work whenever I'm with you, whether that is writing a paper, or rolling a cigarette and having a glass of wine in Helsinki.
- **Kunihiko Sadakane** – Thank you for helping a young student pursue his dream of doing a PhD in succinct data structures in Japan, and being patient when I spent too much time having fun in Tokyo. You have encyclopedic knowledge, and I was incredibly lucky to be your student.
- **Uno Takeaki** – You were always able to calm my anxieties of being a 30-something PhD student, and encouraged me to enjoy writing my dissertation. Thank you for your guidance, patience, and understanding.
- **Martin Muggli** – Thank you for being somebody I could ask questions that I really should already know the answer to, and for carrying the codebase on. Thanks as well for the real-talk. I learned a lot about myself from you.

- **Travis Gagie, Jouni Siren, and Christina Boucher** – co-authors, incredibly smart and creative, and a pleasure to work with.
- **James Harland** – For introducing me to research, encouraging exploration, and for being a fantastic mentor.
- **Rayan Chikhi, Dinghua Li, Sean Jackman, Jared Simpson, Roberto Grossi, Rajeev Raman, and Srinivasa Rao Satti** – For being available to answer my questions about a field that was new to me, and for making me feel like a welcome member of our research community.
- **Nick Greenfield and Nava Whiteford** – For hiring me as a succinct de Bruijn graph consultant!
- **James Hayton and Javier Grande** – This dissertation has been a monkey on my back for years, but you cut through my anxieties and helped me get it done.

Of my family and friends, these people helped me immeasurably:

- **John Bowe** – For finding out exactly how cheap Tokyo is not. Thanks for sponsoring me.
- **Christine Mulvey** – Thank you for helping me learn to program as a kid – I remember you debugging my basic programs that I typed in from library books. And for doing my assignments in school – why didn't you do my dissertation though!?
- **James and Nikolas Bowe** – For the brotherly encouragement and competition. I can finally say I'm better than you both.
- **Christian Sea Jones** – Thanks for looking after me in Tokyo, as well as laughing every time I said "I'm trying to finish my dissertation". Do or do not, there is no try.
- **Ellen Chou** – For being the first person I met in Tokyo, showing me the cool bars and how to do purikura, and giving me an amazing birthday present a few days after I moved there.

It was incredibly nostalgic to sit here and write this list and reflect on how grateful I am to have met each of you.

Appendix A

Relative Select

Motivated by the problem of storing coloured de Bruijn graphs, we show how, if we can already support fast select queries on one string, then we can store a little extra information and support fairly fast select queries on a similar string.

A.1 Introduction

Many compressed data structures for strings rely on three fundamental queries: access, rank and select. The query $S.access(i)$ on a string S returns its ith character; the query $S.rank_a(i)$ returns the number of occurrences of character a in the prefix of S of length i; and the query $S.select_a(j)$ returns the position of the jth leftmost occurrence of a in S. Suppose we have a data structure supporting these queries on a string S_1 and we want another data structure supporting them on a similar string S_2. It is not difficult to store S_2 in small space and support access to it via access to S_1. For example, we can find a longest common subsequence of S_1 and S_2, store two bitvectors with 1s marking their characters not in that subsequence, and store the characters marked in S_2. The total number of 1s in the two bitvectors is at most twice the standard edit distance d between S_1 and S_2 (i.e., the number of single-character insertions, deletions and substitutions needed to change one into the other) so we can store them in $\mathcal{O}(d)$ space and support rank and select on them using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time using a sparse-bitvector implementation [93]. To access $S_2[i]$, we check whether it appears in the common subsequence: if so, we use rank and select queries on the bitvectors to find the corresponding character in S_1, which we access; if not, we find $S_2[i]$'s rank among characters marked in S_2 and look it up.

Last year, when describing their relative FM-index data structure, Belazzougui et al. [94] showed how to store $\mathcal{O}(d)$ extra words and support any rank query on S_2 using

$\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a rank query on S_1. In this paper we show how to store $\mathcal{O}(d)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1. We call this *relative select* and we expect it to be useful when storing compressed data structures for navigating in coloured de Bruijn graphs [24].

Belazzougui el al. were interested in saving space when storing FM-indexes [33] for many genomes from the same species. An FM-index for a genome is essentially just a data structure supporting access and rank on the Burrows-Wheeler Transform [32] (BWT) of that genome. The BWT sorts the characters of a string into the lexicographic order of the suffixes that immediately follow them. The edit distance between two genomes from the same species tends to be small relative to their lengths and in practice the edit distance between their BWTs also tends to be small. Therefore, if we store the FM-index for one genome normally, we can use Belazzougui et al.'s result to save space when storing FM-indexes for other genomes from the same species (at the cost of higher query times).

It is possible to support nearly all the functionality of an FM-index without using select queries on the underlying BWT, so Belazzougui et al. did not consider relative select. When the FM-index is used in a compressed suffix tree, however, select queries are needed for computing suffix links and for certain other operations. Our interest in relative select comes from Bowe et al.'s [39] (see also [38]) compressed representation of de Bruijn graphs — which is based on something like an FM-index and uses select queries to find nodes' predecessors, and which we call the BOSS representation for the authors' initials — and the possibility of extending it to coloured de Bruijn graphs. Our plan for future work is to view a coloured de Bruijn graph as a union of normal de Bruijn graphs, and relatively compress the BOSS representations of those graphs. Due to space constraints, we provide a brief summary of the BOSS representation and coloured de Bruijn graphs as an appendix. In Section A.2 we describe how we implement relative select, and in Section A.3 we show experimentally that our implementation is practical. For simplicity and because we are interested mainly in working with DNA, we assume throughout that the size of the alphabet is constant, and we work in the word-RAM model with $\Omega(\log(|S_1| + |S_2|))$-bit words.

A.2 Design

Although our implementation of relative select is made up of steps that are individually very simple, the overall effect might be confusing. To mitigate this, we break our presentation into pieces: first, we consider the case when S_2 is a subsequence of S_1; then, we consider the case when S_2 is a supersequence of S_1; and finally, we combine our solutions for these special cases to obtain a general solution. We close this section with a small example.

Lemma A.1. *Given a select data structure for a string S_1, and a subsequence S_2 of S_1, we can store $\mathcal{O}(|S_1| - |S_2|)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1.*

Proof. We store a bitvector $B[1..|S_1|]$ with 1s marking the characters of S_1 that do not appear in S_2. For each distinct character x, we store a bitvector $B_x[1..\mathsf{occ}(x, S_1)]$, where $\mathsf{occ}(x, S_1)$ is the number of occurrences of x in S_1, with 1s marking the occurrences of x in S_1 that do not appear in S_2. We use the same sparse-bitvector implementation as in Section A.1, so this takes a total of $\mathcal{O}(|S_1| - |S_2|)$ extra words and lets us compute

$$S_2.\mathit{select}_x(i) = B.\mathit{rank}_0(S_1.\mathit{select}_x(B_x.\mathit{select}_0(i)))$$

using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1. To see why this equality holds, consider that $B_x.\mathit{select}_0(i)$ returns the rank in S_1 of the ith x that appears in S_2; $S_1.\mathit{select}_x(B_x.\mathit{select}_0(i))$ returns the position of that x in S_1; and $B.\mathit{rank}_0(S_1.\mathit{select}_x(B_x.\mathit{select}_0(i)))$ returns the position of that x in S_2. □

Lemma A.2. *Given a select data structure for a string S_1, and a supersequence S_2 of S_1, we can store $\mathcal{O}(|S_2| - |S_1|)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on S_1.*

Proof. We store a bitvector $B[1..|S_2|]$ with 1s marking the characters of S_2 that do not appear in S_1, and a select data structure for the subsequence D of S_2 consisting of those marked characters. For each distinct character x, we store a bitvector $B_x[1..\mathsf{occ}(x, S_2)]$ with 1s marking the occurrences of x in S_2 that do not appear in S_1. We use a sparse-bitvector implementation again, so this takes a total of $\mathcal{O}(|S_2| - |S_1|)$ extra words and lets us compute

$$S_2.\mathit{select}_x(i) = \begin{cases} B.\mathit{select}_0(S_1.\mathit{select}_x(B_x.\mathit{rank}_0(i))) & \text{if } B_x[i] = 0, \\ B.\mathit{select}_1(D.\mathit{select}_x(B_x.\mathit{rank}_1(i))) & \text{if } B_x[i] = 1. \end{cases}$$

using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on S_1. To see why this equality holds, suppose the ith x in S_2 also appears in S_1, so $B_x[i] = 0$. Consider that $B_x.\mathit{rank}_0(i)$ returns the rank of that x in S_1; $S_1.\mathit{select}_x(B_x.\mathit{rank}_0(i))$ returns the position of that x in S_1; and $B.\mathit{select}_0(S_1.\mathit{select}_x(B_x.\mathit{rank}_0(i)))$ returns the position of that x in S_2. Now suppose the ith x in S_2 does not appear in S_1, so $B_x[i] = 1$. Consider that $B_x.\mathit{rank}_1(i)$ returns the rank of that x in D; $D.\mathit{select}_x(B_x.\mathit{rank}_1(i))$ returns the position of that x in D; and $B.\mathit{select}_1(D.\mathit{select}_x(B_x.\mathit{rank}_1(i)))$ returns the position of that x in S_2. □

Theorem A.1. *Given a select data structure for a string S_1, and another string S_2, we can store $\mathcal{O}(d)$ extra words, where d is the edit distance between S_1 and S_2, and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1.*

Proof. Consider a sequence of d single-character insertions, deletions and substitutions that turns S_1 into S_2. Let C be the common subsequence of S_1 and S_2 consisting of characters left unchanged by these d edits (or a longer common subsequence if we can find one). By Lemma A.1, we can store $\mathcal{O}(d)$ extra words and support any select query on C using $\mathcal{O}(\log\log|S_1|)$ time on top of a select query on S_1. By Lemma A.2, we can then store $\mathcal{O}(d)$ extra words and support any select query on S_2 using $\mathcal{O}(\log\log|S_2|)$ time on top of a select query on C. Therefore, we can store $\mathcal{O}(d)$ extra words on top of the select data structure for S_1 and support any select query on S_2 using $\mathcal{O}(\log\log(|S_1| + |S_2|))$ time on top of a select query on S_1. □

For example, consider the strings $S_1 = \mathsf{TCTGCGTAAAAGGTGC}$ and $S_2 = \mathsf{TGCTCGTAAAACGCG}$ (the BWTs of $\mathsf{GCACTTAGAGGTCAGT}$ and $\mathsf{GCACTAGACGTCAGT}$, respectively, from the running example in Belazzougui et al.'s paper). Their edit distance is 5 and their longest common subsequence is $C = \mathsf{TCTCGTAAAAGG}$. If we already have a select data structure for S_1 and we want one for S_2, we first add support for relative select on C by the bitvectors $B, B_{\mathsf{A}}, \ldots, B_{\mathsf{T}}$, shown below; then we add support for relative select on S_2 by storing bitvectors $B', B'_{\mathsf{A}}, \ldots, B'_{\mathsf{T}}$, also shown below, and a select data structure for $D = \mathsf{GCC}$. We note that if we have a relative FM-index for S_2 with respect to S_1, then it already includes B, B' and D.

$$
\begin{array}{lcl@{\qquad}lcl}
B[1..16] & = & 0001000000010101 & B'[1..15] & = & 010000000001010 \\
B_{\mathsf{A}}[1..4] & = & 0000 & B'_{\mathsf{A}}[1..4] & = & 0000 \\
B_{\mathsf{C}}[1..3] & = & 001 & B'_{\mathsf{C}}[1..4] & = & 0011 \\
B_{\mathsf{G}}[1..5] & = & 10100 & B'_{\mathsf{G}}[1..4] & = & 1000 \\
B_{\mathsf{T}}[1..4] & = & 0001 & B'_{\mathsf{T}}[1..3] & = & 000
\end{array}
$$

To compute $S_2.\mathit{select}_{\mathsf{C}}(4)$, for instance, we check $B'_{\mathsf{C}}[4]$ and see it is 1, meaning the fourth C in S_2 does not appear in C. Since $B'_{\mathsf{C}}.\mathit{rank}_1(4) = 2$, it is the second C in D. Since $D.\mathit{select}_{\mathsf{C}}(2) = 3$, it is the third character in D. Finally, since $B'_1.\mathit{select}_1(3) = 14$, it is the 14th character in S_2, meaning $S_2.\mathit{select}_{\mathsf{C}}(4) = 14$.

To compute $S_2.\mathit{select}_{\mathsf{G}}(3)$, we check $B'_{\mathsf{G}}[3]$ and see it is 0, meaning the third G in S_2 also appears in C. Since $B'_{\mathsf{G}}.\mathit{rank}_0(3) = 2$, it is the second G in C. Since

$$C.\mathit{select}_{\mathsf{G}}(2) = B.\mathit{rank}_0(S_1.\mathit{select}_{\mathsf{G}}(B_{\mathsf{G}}.\mathit{select}_0(2))) = 11\,,$$

it is the 11th character in C. Finally, since $B'_1.\mathit{select}_0(11) = 13$, it is the 13th character in S_2, meaning $S_2.\mathit{select}_{\mathsf{G}}(3) = 13$.

Table A.1: Average query times for 100 million random LF and Ψ queries on NA12878 stored relative to the human reference genome, with and without chromosome Y.

	FM-index			**Relative FM-index**			**+ Relative Select**	
ChrY	**space**	LF	Ψ	**space**	LF	Ψ	**total space**	Ψ
yes	1090 MB	0.55 µs	1.22 µs	218 MB	3.95 µs	48.0 µs	382 MB	6.11 µs
no	1090 MB	0.55 µs	1.11 µs	181 MB	3.84 µs	44.8 µs	331 MB	6.12 µs

A.3 Experiments

We augmented our implementation of the Relative FM-index with the new select structure.[1] The implementation is written in C++ and based on the Succinct Data Structures Library 2.0 [93]. We used g++ version 4.8.1 to compile the code, and ran the experiments on a system with two 16-core AMD Opteron 6378 processors and Linux kernel 2.6.32. We used a single core for the query tests.

As our reference sequence, we used the 1000 Genomes Project's version of the GRCh37 human reference genome, both with (3.096 Gbp) and without (3.036 Gbp) chromosome Y. For a target sequence, we chose the maternal haplotypes of the 1000 Genomes Project's individual NA12878 (3.036 Gbp) [95]. We built a plain FM-index for the reference sequences and the target sequence, as well as relative FM-indexes for the target sequence relative to both references and with and without structures for relative select; the lengths of the common subsequences used were 2.992 Gbp and 2.991 Gbp, respectively. In all cases, we used plain bitvectors in the wavelet trees and entropy-compressed bitvectors [29] for marking the common subsequences.

To test the performance of relative select, we ran 100 million random $\Psi(i) = \mathsf{BWT}.select_c(i - \mathsf{C}[c])$ queries on the BWT of the target sequence, using a plain FM-index and Relative FM-indexes with and without relative select. (Character c is the ith character in the BWT in sorted order, while $\mathsf{C}[c]$ is the number of occurrences of characters smaller than c in the BWT.) The implementation of Ψ in the Relative FM-index without relative select was based on binary searching with rank queries. As a comparison, we also ran $\mathsf{LF}(i) = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{BWT}.rank_{\mathsf{BWT}[i]}(i)$ queries. Table A.1 shows the results: the relative FM-indexes without relative select are each about a fifth the size of the normal FM-indexes but rank queries are about seven times slower and select queries are about forty times slower; the relative FM-indexes with relative select are about a third the size of the normal FM-indexes but select queries are only about five times slower (rank queries are unaffected).

[1] `https://github.com/jltsiren/relative-fm`

A.4 Appendix: de Bruijn Graphs

In biology, the (edge-centric) *kth-order de Bruijn graph* for a set of strings (e.g., DNA reads) is the graph whose nodes are those strings' k-mers (substrings of length k), with a directed edge (u, v) from u to v if at least one of the strings contains a substring of length $k + 1$ with u as a prefix and v as a suffix. We label (u, v) with the last character of v. Almost all state-of-the-art DNA assemblers build contigs via Eulerian assembly [10, 11] on de Bruijn graphs, making their space- and time-efficient representation an important problem in bioinformatics.

Bowe et al. add certain dummy nodes and edges, sort the edges into the right-to-left lexicographic order of the nodes they leave, and take the last column of the matrix whose rows are the edges in sorted order (or, equivalently, take the last character in each edge). The result is like a BWT in which edges correspond to characters and nodes correspond to the substrings containing all their out-edges' characters. For example, for the string TACGTCGACGACT and $k = 3$, Bowe et al. derive the edge-BWT TCCGTGGATAA$C. (This example is from [38].) With some auxiliary data structures, we can use rank and select queries on this edge-BWT to navigate forward and backward in the graph.

For the two strings TACGTCGACGACT and TACGACGCGACT and $k = 3$, the de Bruijn graph is 2 nodes larger than the graphs for strings separately. If we store whether each edge occurs in the first string, the second string, or both, then the result is a *coloured de Bruijn graph*. Coloured de Bruijn graphs were introduced by Iqbal et al. [24] for detecting variations between individuals' genomes, and are now also used in other areas of genomics. We can view the coloured de Bruijn graph as the union of each graph consisting of edges of the same colour. In a future paper we will show how to combine the BOSS representations of the individual de Bruijn graphs to obtain a representation of the coloured de Bruijn graph, and also how to relatively compress the auxiliary data structures for the BOSS representations of the individual graphs.

We can use Belazzougui et al.'s result to relatively compress the edge-BWTs of the individual graphs while still supporting rank over them. For example, the edge-BWTs for TACGTCGACGACT and TACGACGCGACT with $k = 3$ are TCCGTGGATAA$C and TCCGTGGACAA$, respectively. They are so close — edit distance 2 — because most of the strings' 4-tuples are common to both and, thus, most of their de Bruijn graphs' edges are common to both. We note that, for reasonable values of k, most of the $(k+1)$-mers in genomes from the same species should also be common to most of the genomes. In this paper we showed how to support relative select on similar strings, which we will eventually need to navigate backward across edges in our representation of coloured de Bruijn graphs.

Appendix B

Succinct de Bruijn Graphs Blog Post

This post will give a brief explanation of a Succinct implementation for storing *de Bruijn graphs*,[1] which is recent (and continuing) work I have been doing with Sadakane.

Using our new structure, we have squeezed a graph for a human genome (which took around 300 GB of memory if using previous representations) down into 2.5 GB. In addition, the construction method allows much of the work to be done on disk. Your computer might not have 300 GB of RAM, but you might have 2.5 GB of RAM and a hard disk.

I have given a talk about this a few times, so I've been itching to write it up as a blog post (if only to shamelessly plug my blog at conferences). However, since it is a lowly blog post, I won't give attention to the gory details, nor provide any experimental results. But this post is merely to *communicate the approach*. Feel free to check out our conference paper [39], and stay tuned for the journal paper.

In this blog post, I will first give an introduction to de Bruijn graphs (Section B.1) and how they are used in DNA assembly (Section B.2). Then I will briefly explain some previous implementations (Section B.3) before reaching the main topic of this post: our new succinct representation (Section B.4). The explanation first explains some preliminaries, such as how we were inspired by the Burrows Wheeler Transform (Section B.4.1), and what rank and select are (Section B.4.3), which are required to understand the construction method (Section B.4.2), and traversal interface (Section B.4.4) respectively.

For those following along at home, I have implemented a demo version in Python.[2] It doesn't use efficient implementations of rank and select, nor provide any compression – it is merely meant to demonstrate the key ideas with (hopefully) readable high level code. An optimised version will be made available at some point.

[1] `http://en.wikipedia.org/wiki/De_Bruijn_graph`
[2] `https://github.com/alexbowe/debby/blob/0.1.1/debby.py`

Okay, here we go...

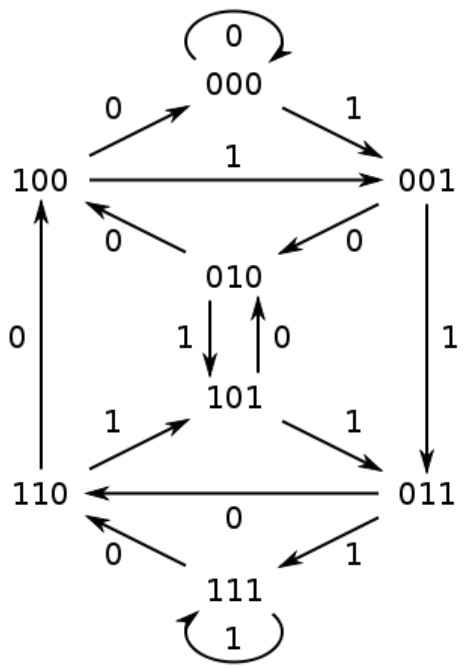
B.1 De Bruijn Graphs

De Bruijn graphs are a beautifully simple, yet useful combinatoric object which I challenge you not to lose sleep over.

Since their discovery around 1946, they have been used in a variety of applications, such as encryption, psychology, chess and even card tricks. Quite recently they have become a popular data structure for DNA assembly of short read data.

They are defined as a directed graph, where each node u represents a fixed-length string (say, of length k, and an edge exists from u to v iff they overlap by $k-1$ symbols. That is, $u[2..k] = v[1..k-1]$.

Let's make this concrete with a diagram:



Doesn't this just *feel good* to look at? Now tilt your head and look at it this way: it is essentially a *Finite State Machine*[3] with additional bounded memory of the last k visited states, if you added a few more states to allow for incomplete input at the start. For example, if X represents blank input, from a starting state XXX we might have the transition chain: XXX $\rightarrow$ XX1 $\rightarrow$ X10 $\rightarrow$ 101 (which is already a node in the graph). Put this way, it is easy to

[3] `https://en.wikipedia.org/wiki/Finite-state_machine`

see that the k previous edges define the current node. This perspective will make things easier to understand later, I promise.

Still with me? Good. Then let's also consider that if one node is defined by a k-length string, then a *pair of nodes* (i.e. an edge) can be identified by a $k+1$ length string, since they overlap and differ by 1 symbol. This will also be important later.

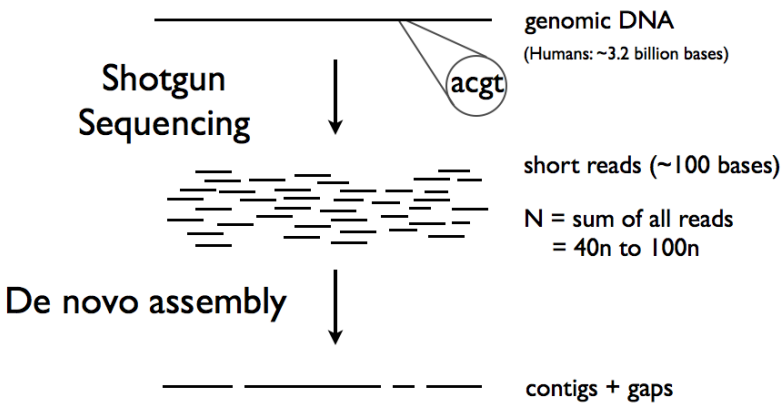
And of course, this can be extended to larger alphabet sizes than binary (say, 4...).

B.2 DNA Assembly

First suggested in 2001 by Pevzner et al. [11], we can use de Bruijn graphs to represent a network of overlapping *short read data*.

The long and short of it (heh heh) is that a DNA molecule is currently too difficult to sequence (that is, read it into a computer) in its entirety. Special methods must be used so we can sequence parts of the molecule, and hand off the putting-back-together process (assembly) to an algorithm.

One current popular sequencing method is *shotgun sequencing*, which clones the genome a bunch of times, then randomly breaks each clone into short segments. If we can sequence the short segments, then the fact that we randomly cut up the clones should lead us to have overlapping reads. Of course it is a bit more complicated than this in reality, but this is the essence of it.

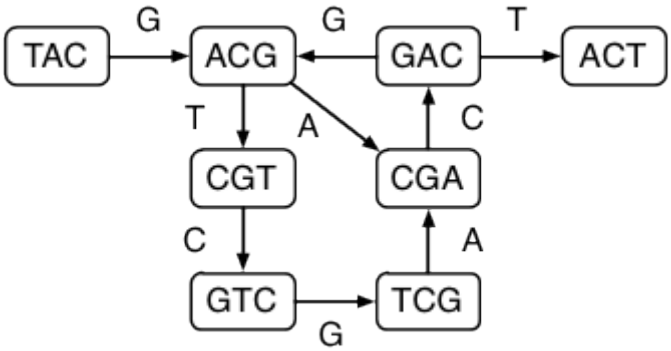


We then move a sliding window over each read, outputing overlapping k-mers (the k-length strings), which we use to create our de Bruijn graph.

About now mathematicians among you might raise your hand to tell me "that may not technically yield a de Bruijn graph". That's correct – in Bioinformatics the term "de Bruijn graph" is overloaded to mean a subgraph. Even though genomes are long strings, most genomes

won't have *every single k-mer* present, and there is usually repeated regions. This means our data will be sparse.

Consider the following contrived example. Take the sequence TACGACGTCGACT. If we set $k = 3$, our k-mers will be TAC, ACG, CGA, and so on. We would end up with this de Bruijn graph:



After we construct the graph from the reads, assembly becomes finding the "best" contiguous regions. The jury is still out on what the best method is (or what "best" even means); the point of this post isn't about assembly, but our implementation of the data structure and how it provides all required navigation options to implement any traversal method. I recommend reading this primer from Nature[4] if you want to get deeper into this.

Even though this is only a de Bruijn *subgraph*, these things still grow pretty big. It is worthwhile considering how to handle this scalability issue, if only to reduce hardware requirements of sequencing (thus proliferating personal genomics), and potentially improve traversal speed (due to better memory locality). Increased efficiency might also enable richer multiple-genomic analysis.

B.3 Previous Representations

One of the first approaches to this was to scale "horizontally". Simpson et al. [15] introduced ABySS in 2009. The graph for reads from a human genome (HapMap: NA18507),[5] which used a distributed hash table, reached 336 GB.

In 2011, Conway and Bromage [21] instead approached this problem from a "vertical" scaling perspective (that is, scaling to make better use of a single system's resources), by using a sparse

[4] `http://www.cs.ucdavis.edu/~gusfield/cs225w12/deBruijn.pdf`
[5] `ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA010/SRA010896/SRX016231/`

bitvector (by Okanohara and Sadakane [25]) to represent the $(k+1)$-mers (the edges), and used *rank and select* (to be described shortly in Section B.4.3) to traverse it. As a result, their representation took 32 GB for the same data set.

Minia, by Cikhi and Rizk (2012) [22], proposed yet another approach by using a *bloom filter*[6] (with additional structure to avoid false positive edges that would affect the assembly). They traverse by generating possible edges and testing for it in the bloom filter. Using this approach, the graph was reduced to 5.7 GB.

B.4 Our Succinct Representation

As stated, we were able to represent the same graph in 2.5 GB (after some further compression techniques, which I will save for a future post).

The key insight is that the edges define overlapping node labels. This is similar to that of Conway and Bromage, although they have some redundancy, since some *nodes* are represented more times than necessary.

We further exploit the *mutual information*[7] of edges by taking inspiration from the Burrows Wheeler Transform [32].

B.4.1 Inspiration from the Burrows Wheeler Transform

The Burrows Wheeler Transform [32] is a reversible string permutation that can be searched directly and has the admirable quality of having long strings of repeated characters (great for compression). The easiest way to calculate the BWT of a string is to sort each symbol by their prefixes in colex order (that is, alphabetic order of the reverse of the string, not reverse alphabetic!) More information can be found on Wikipedia[8] and this Dr. Dobbs[9] article.

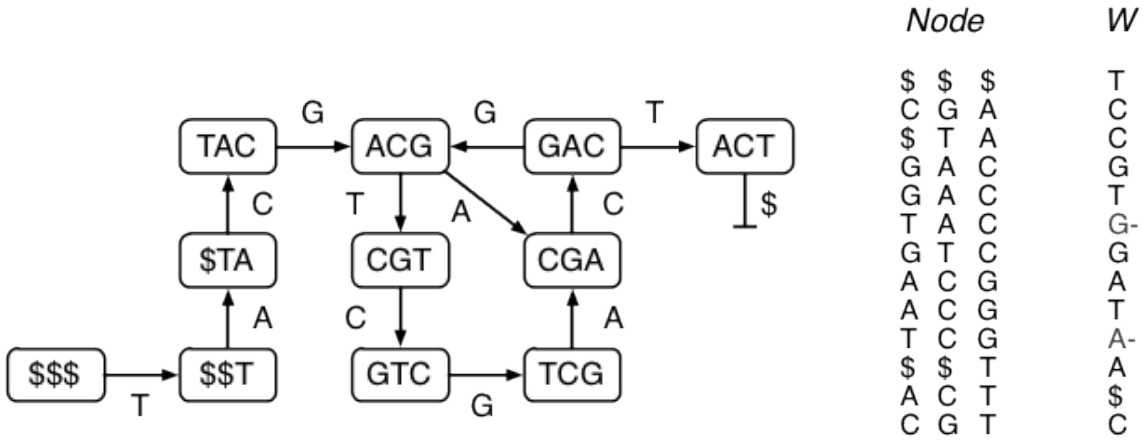
The XBW is a generalisation of the BWT that applies to rooted, labeled trees [31]. The idea is that instead of taking all suffixes, we sort all paths from the root to each node, and support tree navigation (since it isn't a linearly shaped string) with auxiliary bit vectors indicating which edges are leaves, and which are the last edges (of their siblings) of internal nodes.

I won't go into detail, but in the next section you should be able to see glimpses of these two ideas.

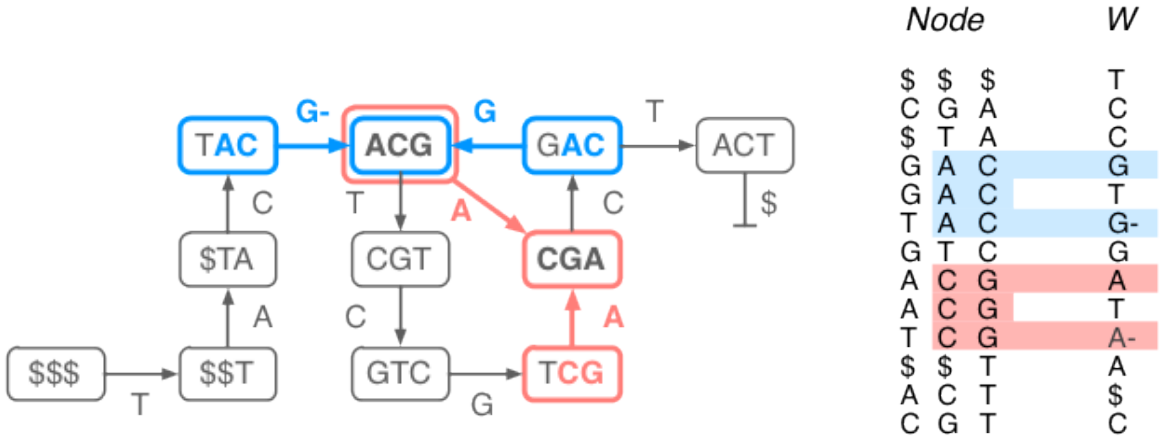
[6] `http://en.wikipedia.org/wiki/Bloom_filter`
[7] `https://en.wikipedia.org/wiki/Mutual_information`
[8] `http://en.wikipedia.org/wiki/Burrows-Wheeler_transform`
[9] `http://marknelson.us/1996/09/01/bwt/`

B.4.2 Construction

The simplest construction method[10] is to take every $< \text{node}, \text{edge} >$ pair and sort them based on the reverse of the node label (colex order), removing duplicates[11]. We also padding to ensure every node has an incoming and an outgoing edge. This maintains the fact that a node is defined by its previous k edges. An example can be seen below:



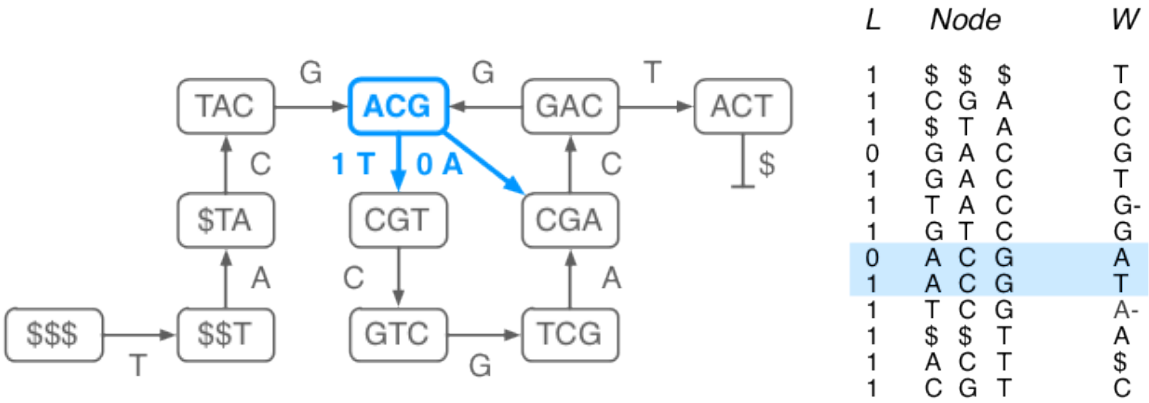
You may have spotted that we have flagged some edges with a minus symbol. This is to disambiguate identically labelled incoming edges – edges that exit separate nodes, but have the same symbol, and thus enter the same node. In the example below, the nodes ACG and CGA both have two incoming edges.



[10]The paper also describes an online construction method (where we can update by appending), and an iterative method that builds the k-dimensional de Bruijn graph from an existing (k–1)-dimensional de Bruijn graph.

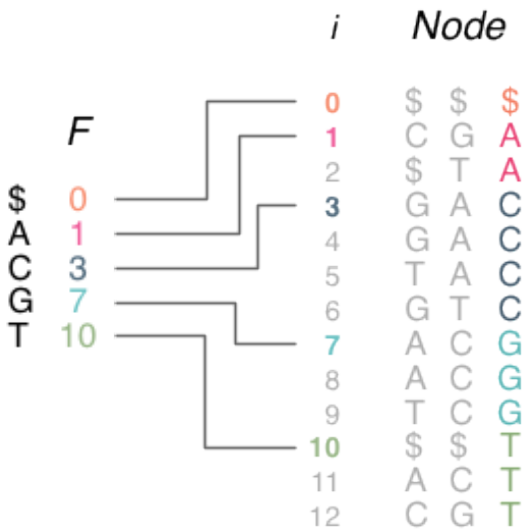
[11]We currently use an external merge sort, but intend to optimise as this is where Minia beats us time-wise.

Notice that each outgoing edge is stored contiguously? We include a bit vector to represent whether an edge is the *last* edge exiting a node. This means that each node will have a sequence of zero-or-more 0-bits, followed by a single 1-bit.



Since a 1 in the L vector identifies a unique node, we can use this vector (and *select*, explained shortly in Section B.4.3) to index nodes, whereas standard array indexing points to edges.

Finally, instead of storing the node labels we just need to store the final column of the node labels. Since the node labels are sorted, it is equivalent to store an array of first positions:



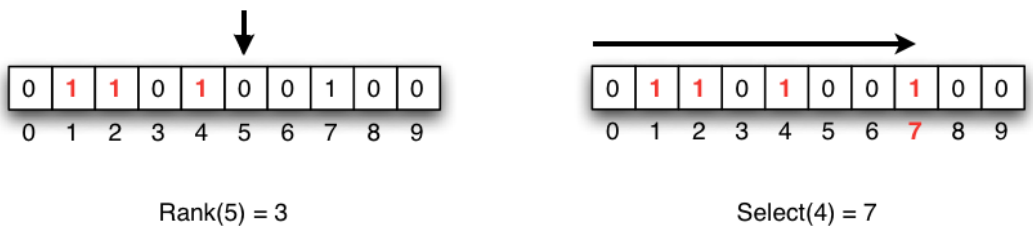
In total we have a bitvector L, an array of flagged edge labels W, and a position array F of size σ (the alphabet size). Respectively, these take m bits, $m \log 2\sigma = 3m$ bits (for DNA), and $\sigma \log m = o(m)$ bits[12], given m edges – a bit over 4 bits per edge. Using appropriate structures

[12]This is "little o" notation, which may be unfamiliar to some people. Intuitively it means "grows much slower

(not detailed) we can compress this further, to around 3 bits per edge.

B.4.3 Rank and Select

Rank and select are the bread and butter of succinct data structures, because so many operations can be implemented using them alone. $rank_c(i)$ returns the number of occurences of symbol c on the closed range $[0, i]$, whereas $select_c(i)$ returns the position of the ith occurence of symbol c.



They are *kind of* like inverse functions, although *rank*() is not *injective*[13], so cannot have a true inverse. For this reason, if you want to find the *position* of the left-nearest c, you would have to use *select*() in orchestra with *rank*() (a common pattern).

Speaking of patterns of use, it may also help to keep this in mind: rank is for counting, and select is for searching, or can be thought of as an indirect addressing technique (such as addressing a node using the L array). Two rank queries can count over a range, whereas two select queries can find a range. A rank and a select query can find a range where either a start point or end point are fixed.

Rank, select and standard array access can all be done in $\mathcal{O}(1)$ time when $\sigma = polylog(N)$,[14] if represent a bitvector using the structure described by Raman, Raman and Rao in 2007 [29] (which I explained in an earlier blog post[15]), and for larger alphabets use the index described by Ferragina, Manzini, Makinen, and Navarro in 2006 [28]. In our implementation, we use modified versions to get it down to 3 bits per edge.

than", and is stricter than big O. A formal definition can be found on Wikipedia, `https://en.wikipedia.org/wiki/Big_O_notation#Little-o_notation`

[13] `http://en.wikipedia.org/wiki/Injective_function`

[14] `http://stackoverflow.com/questions/1801135/what-is-the-meaning-of-o-polylogn-in-particular-how-is-polylogn-defined`

[15] `https://alexbowe.com/rrr`

B.4.4 Interface Overview

While it might not be obvious, these three arrays provides support for a full suite of navigation operations. An overview is given in these tables, which link to the implementation details that follow.

First, to navigate each of the edges, we define two internal functions (using rank and select calls, see Section B.4.3):

Operation	**Description**	**Complexity**
forward(i)	Return index of the *last edge* of the *node pointed to* by edge i	$\mathcal{O}(1)$
backward(i)	Return index of the *first edge* that *points to the node* that the edge at i *exits*.	$\mathcal{O}(1)$

Using these two functions, we can implement the less confusing public interface below, which operate on node indexes:

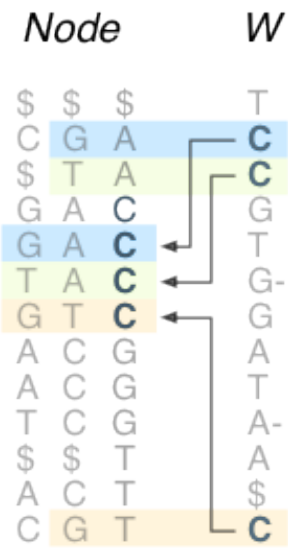
Operation	**Description**	**Complexity**
outdegree(v)	Return number of outgoing edges from node v	$\mathcal{O}(1)$
outgoing(v, c)	From node v, follow the edge labeled by symbol c.	$\mathcal{O}(1)$
label(v)	Return (string) label of node v.	$\mathcal{O}(k)$
indegree(v)	Return number of incoming edges to node v.	$\mathcal{O}(1)$
incoming(v, c)	Return predecessor node starting with symbol c, that has an edge to node v.	$\mathcal{O}(k \log \sigma)$

The details of the above functions are given in the following sections.

B.4.5 Forward

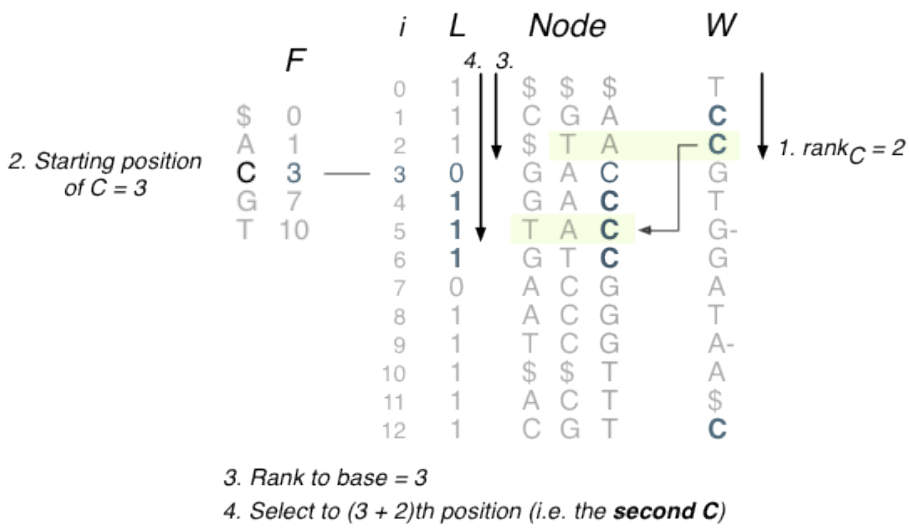
In order to support the public interface, we create for ourselves a simpler way to work with edges: the complementing forward and backward functions.

Recall that all node labels are defined by predecessor edges, then we have represented each edge in two different places: the F array (which is equivalent to the last column of the "*Node*" array), and the edge array W. It follows that, since it is sorted, the node labels maintain the same *relative order* as the edge labels. This can be seen in the following figure:



Note that the number of Cs in the last column of *Node* is different from the number of Cs in W, because the first C in W points to two edges. For this reason, we ignore the first edge from node GAC (although it doesn't affect the relative order). In fact, we ignore any edge that doesnt have $L[i] == 1$.

Then, following an edge is simply finding the corresponding relatively positioned node! All it takes is some creative counting, using rank and select (Section B.4.3), as pictured below:



First we access $W[i]$ to find the edge label, then calculate $rank_C$ up to row to gives us the relative ordering of our *edges* with this label. Let's call this relative index r. In our example

we are following the 2nd C-labeled edge.

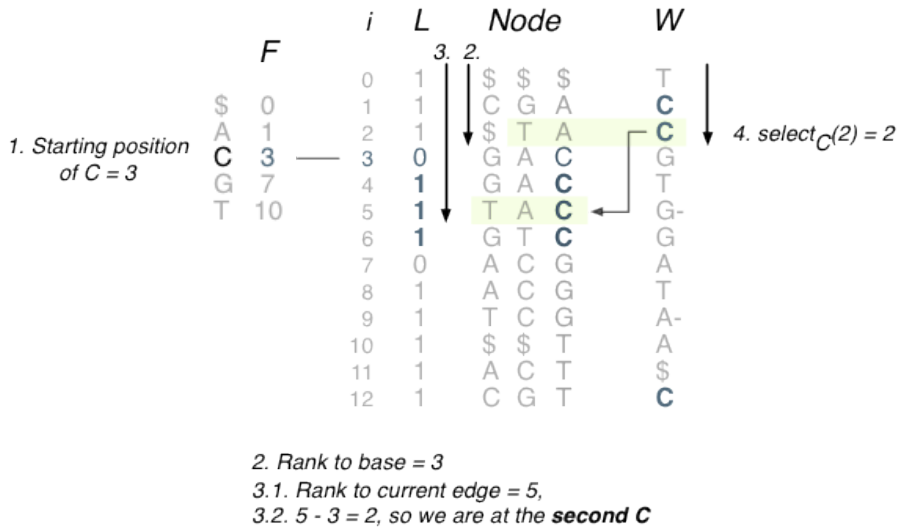
To find the 2nd occurence of C in F, first we need to know where the first occurence is. We can use F to find that. Then we can select to the 2nd one, using the last array. Because the last array is binary only, this requires us to count how many 1s there are before the run of Cs (using rank), then adding 2 to land us at the 2nd C.

The W access, rank over W, rank and select over L, accessing F, and the addition are all done in $\mathcal{O}(1)$ time, so forward also takes $\mathcal{O}(1)$ time.

B.4.6 Backward

Backward is very similar to forward, but calculated in a different order: we find the relative index of the node label first, and use that to find the corresponding edge (which may not be the only edge to point to this node, but we define it to point to the first one, that is one that isn't flagged with a minus).

We can find our relative index of the node label by issuing two rank queries instead, and using select on W to find the first incoming edge.

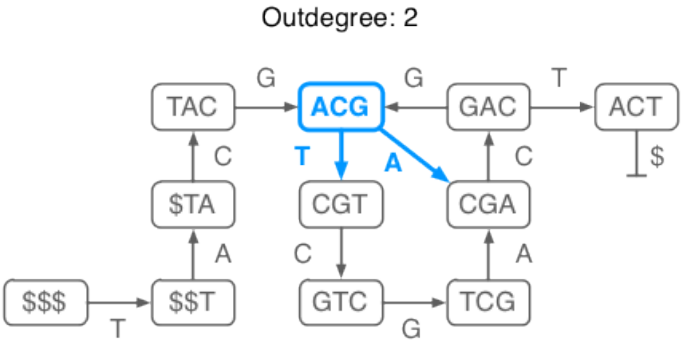


For similar reasons to Forward this is $\mathcal{O}(1)$.

B.4.7 Outdegree

This is an easy one. This function accepts a **node** (not edge!) index v, and returns the number of outgoing edges from that node. Why did I say this was easy? Well, remember that all our

outgoing edges from the same node are contiguous in our data structure. See the diagrams below, paying attention to the node ACG.



v	i	L	Node	W
		select(6) - select(5)		
0	0	**1**	$ $ $	T
1	1	**1**	C G A	C
2	2	**1**	$ T A	C
3	3	0	G A C	G
	4	**1**	G A C	T
4	5	**1**	T A C	G-
5	**6**	**1**	G T C	G
6	7	0	A C G	A
	8	**1**	A C G	T
7	9	1	T C G	A-
8	10	1	$ $ T	A
9	11	1	A C T	$
10	12	1	C G T	C

By definition (since nodes are unique and sorted), this will always be the case. We also defined our so-called "last" vector L to have a 0 for every edge from a given node, *except the last* edge. All we need to do is count how many 0s there are, then add 1. Put another way, we need to measure the distance between 1s (since the previous 1 will belong to the previous node). Since we know the node index, we can use select for that!

In the above example, we are querying the outdegree of node 6 (the 7th node due to zero-basing). First we select to find the position of the 7th 1, which gives us the last edge of that node. Then we simply subtract the position of the previous node (node 5, the 6th node): $select(7) - select(6) = 8 - 6 = 2$. Boom.

Select queries can be answered in $\mathcal{O}(1)$ time, so outdegree is also $\mathcal{O}(1)$.

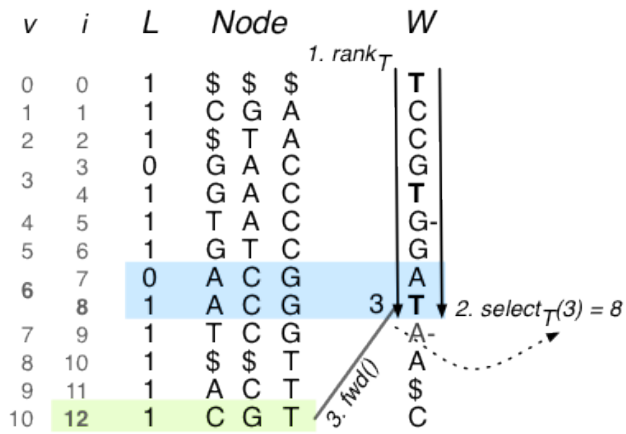
B.4.8 Outgoing

$Outgoing(v, c)$ returns the target node after traversing edge c from node v, which might not exist. The hard part is finding the correct edge index to follow; after that we can conveniently use the $forward()$ function we defined earlier.

To ease the explanation, consider a simple bit vector 00110100. To count how many 1s there are up to and including the 7th position, we would use rank. The answer is 3, but at this stage we still don't know the position of the 3rd 1 (we can't see the bitvector). In general, it may or may not be in the 7th position. We could scan backwards, or we could just use $select(3)$ to find the position (since this returns the first position i that has $rank(i) = 3$).

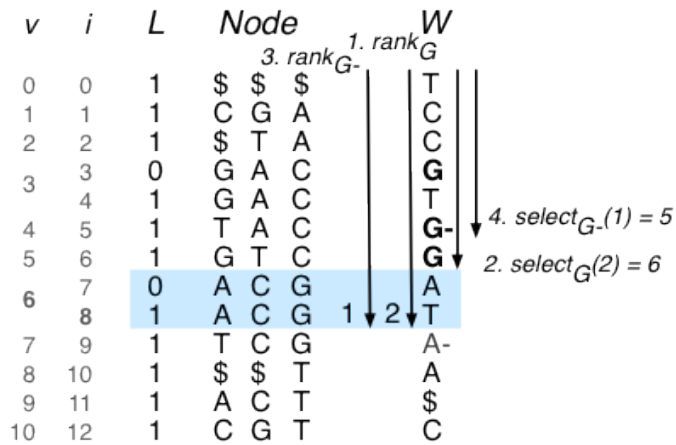
So essentially we can count how many of those edges there are before the node we are interested in, then use select to find the position. If the position is inside this nodes range (of

contiguous edges), then we follow it.



We complicated things a bit by separating our edges into flagged and non-flagged, so we may have to issue two of these queries (for the minus flags). The flagging is useful later in Indegree (Section B.4.10).

In the next example, our nonflagged edge doesn't fall in our nodes range, so we make a second query. It is possible that this one will return a positive result, but in the example it doesn't. By that stage though, we can respond with by returning -1 to signal that the edge doesn't exist.



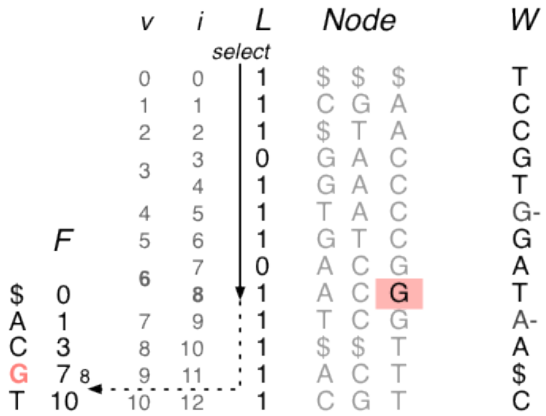
forward() is defined to move us to the last edge of the resulting node, so the value in last will be 1. Hence, we can use rank to convert our edge index into a node index before returning it.

This is a constant number of calls to $\mathcal{O}(1)$ functions, so outgoing is also $\mathcal{O}(1)$.

B.4.9 Label

At some point (e.g. during traversal) we are probably going to want to print the node labels out. Let's work out how to do that.

Remember, we aren't storing the node labels explicitly. The F array will come in handy: We can use the position of our node (found using select) as a reverse lookup into F. This can be done in constant time with a sparse bit vector, or in logarithmic time using binary search, or we can use a linear scan if our alphabet is small. In any case, lets assume it is $\mathcal{O}(1)$ time, although a linear scan might be faster in practice (fewer function calls may yield a lower constant coefficient).



In the above example, we select to node 6 (the 7th 1 in last), which gives us the last edge index (any edge will do, but the last one is the easiest to find). This happens to be row 8, so from F we know that all the last symbols between on the open range $[7, 10)$ are G.

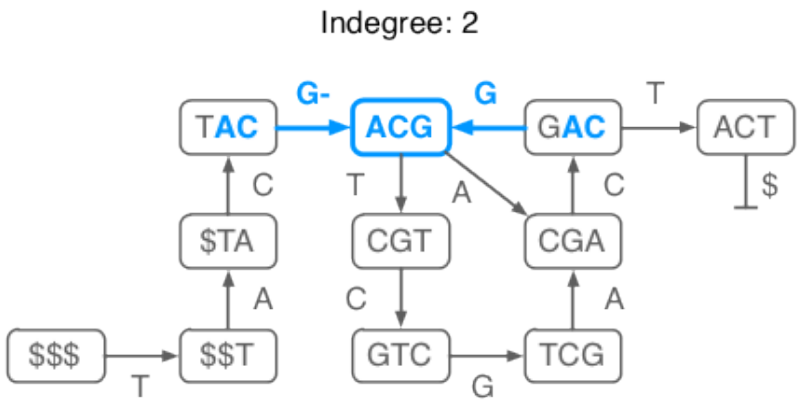
Then we just use $bwd()$ on the current edge to find an edge that pointed to this node, then rinse and repeat k times.

v	i	L	Node	W
0	0	1	$ $ $	T
1	1	1	C G A	C
2	2	1	$ T A	C
3	3	0	G A C	G
	4	1	G A C	T
4	5	1	T A C	G-
5	6	1	G T C	G
6	7	0	A C G	A
	8	1	A C G	T
7	9	1	T C G	A-
8	10	1	$ $ T	A
9	11	1	A C T	$
10	12	1	C G T	C

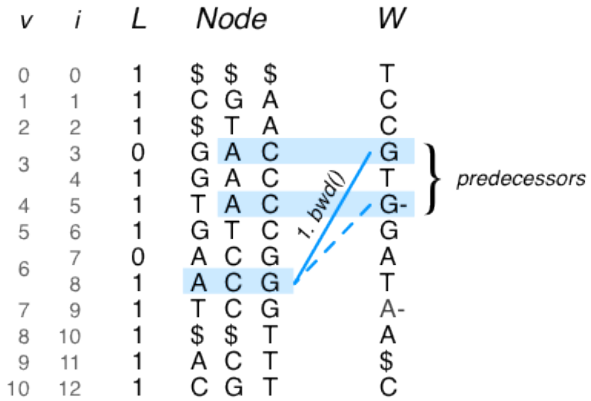
2. bwd()

B.4.10 Indegree

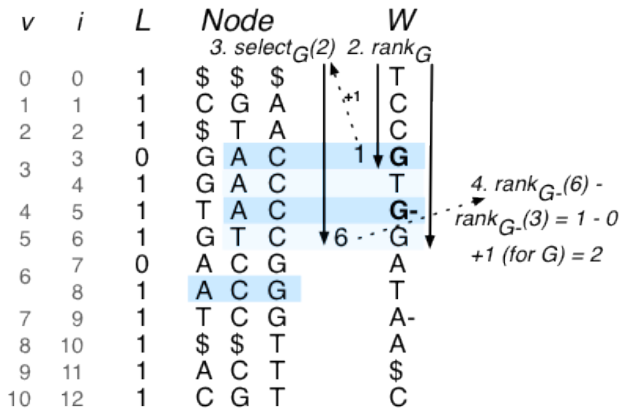
In a similar manner to *outdegree* (Section B.4.7), all we need to do is count the edges that point to the current node label. Take for example the graph:



We can easily find the first incoming edge by using *backward*(). To count the remaining edges our minus flags come in handy; In the W array, the next G (non-flagged) belongs to a different node, because we defined W to have minus flags if the source node has the same $k-1$ suffix (or, if same-labeled edges also share the same target node).



From here it is simple enough to do a linear scan (or even use select) until the next non-flagged edge; the maximum distance it could be is σ^2. For larger alphabets, a more efficient method is to use rank instead:



First we find the position of the next non-flagged G, which gives us the end point of our range (we already have the start point from the first rank). Then we use rank to calculate how many G – there are to the end position, and subtract the initial rank value from this, giving us how many G – occur within the range.

This is once again a constant number of $\mathcal{O}(1)$ function calls, which means *indegree*() is also $\mathcal{O}(1)$.

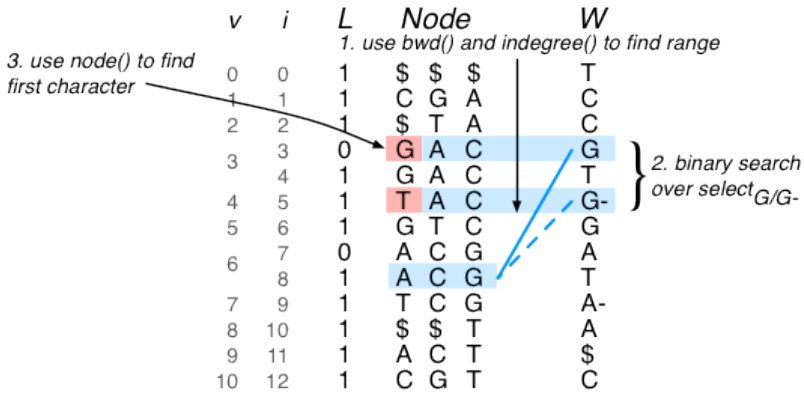
B.4.11 Incoming

Incoming, which returns the predecessor node that begins with the provided symbol, is probably the most difficult operation to implement. However, it does use approaches similar to the previous functions.

Consider this: from *indegree*(), we already know how to count each of the predecessor nodes. We can access these nodes if we instead use select to iterate over the predecessor nodes, rather than using rank to simply count. Then, to disambiguate them by first character, we can use *label*().

A linear scan over the predecessors in this fashion would work, but for large alphabets we can use binary search (with a select call before each array access) to support this in $\mathcal{O}(k \log \sigma)$ time; $\log \sigma$ for the binary search, where each access is a $\mathcal{O}(1)$ select, followed by $\mathcal{O}(k)$ to compute the label.

This is demonstrated in the example below:



B.5 Conclusion

In conclusion, by using memory more efficiently, hopefully the cost of genome seqeuencing can be reduced, both proliferating the technology, but also giving way to more advanced population analysis. To that end, we have described a novel approach to representing de Bruijn graphs efficiently, while supporting a full suite of navigation operations quickly. Much of the (BWT-inspired) construction can be done efficiently on disk, but we intend to improve this soon to compete with Minia.

The total space is a theoretical $m(2 + \log \sigma + o(1))$ bits in general, or $4m + o(m)$ bits for DNA, given m edges. Using specially modified indexes we can lower this to around 3 bits per edge.

I apologize that an efficient implementation isn't available, nor have I provided experimental results. But if you found this post interesting you can get your hands dirty with the Python implementation[16] I have provided. The results and efficient implementation are on their way.

[16] `https://github.com/alexbowe/debby/blob/0.1.1/debby.py`

Bibliography

[1] Francis S. Collins, Michael Morgan, and Aristides Patrinos. The human genome project: Lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[2] For Rare Disease Month February 2019, Dante Labs launches special genetic testing offering for rare disease patients. `https://www.prnewswire.com/news-releases/for-rare-disease-month-february-2019-dante-labs-launches-special-genetic-testing-offering-for-rare-disease-patients-300788748.html`, 2019.

[3] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Comutational Biology*, 2:275–290, 1995.

[4] M. Kasahara and S. Morishita. *Large-Scale Genome Sequence Processing*. Imperial College Press, 2006.

[5] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95:315–327, 2010.

[6] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, 2009.

[7] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. Arachne: a whole-genome shotgun assembler. *Genome Research*, 12:177–189, 2002.

[8] X. Huang and S. P. Yang. Generating a genome assembly with pcap. *Current Protocols in Bioinformatics*, Unit 11.3, 2005.

[9] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. J. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M.

Rubin, M. D. Adams, and J. C. Venter. A whole-genome assembly of drosophila. *Science*, 287:2196–2204, 2000.

[10] R.M. Idury and M.S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2:291–306, 1995.

[11] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci.*, 98(17):9748–9753, 2001.

[12] A. Bankevich et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, 19(5):455–477, 2012.

[13] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA–a practical iterative de Bruijn graph *de novo* assembler. In *Proc. RECOMB*, pages 426–440, 2010.

[14] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272, 2010.

[15] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[16] J. Butler et al. ALLPATHS: *de novo* assembly of whole-genome shotgun microreads. *Genome Res.*, 18(5):810–820, 2008.

[17] M. Sahli and T. Shibuya. Arapan-s: a fast and highly accurate whole-genome assembly software for viruses and small genomes. *BMC Research Notes*, in press.

[18] I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea, L. Williams, S. Young, C. Nusbaum, and D. B. Jaffe. Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology*, 10(R103), 2009.

[19] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18:821–829, 2008.

[20] P. E. C. Compeau, P. A. Pevzner, and G. Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[21] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[22] R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(22), 2012.

[23] D. Haussler et al. Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. *Journal of Heredity*, 100(6):659–674, 2009.

[24] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226–232, 2012.

[25] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.

[26] C. Ye, Z. S. Ma, C. H. Cannon, M. Pop, and D. W. Yu. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(Suppl 6:S1), 2012.

[27] N. G. De Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

[28] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms*, 3(2):No. 20, 2006.

[29] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), 2007.

[30] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees, 2010. Submitted for journal publication. Available at http://arxiv.org/abs/0905.0768. A preliminary version appeared in Proc. ACM-SIAM SODA, pp. 134–149, 2010.

[31] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4:1–4:33, 2009.

[32] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Systems Research Center, 1994.

[33] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[34] J. G. Cleary and I. H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. on Commun.*, COM-32(4):396–402, 1984.

[35] S. Ossowski et al. Sequencing of natural strains of *Arabidopsis Thaliana* with short reads. *Genome Res.*, 18(12):2024–2033, 2008.

[36] The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[37] P. J. Turnbaugh et al. The Human Microbiome Project: exploring the microbial part of ourselves in a changing world. *Nature*, 449(7164):804–810, 2007.

[38] Christina Boucher, Alexander Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proc. Data Compression Conference (DCC)*, pages 383–392. IEEE, 2015.

[39] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Proc. WABI*, pages 225–235, 2012.

[40] R. Chikhi, A. Limasset, S. Jackman, J.T. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. In *Proc. RECOMB*, pages 35–55, 2014.

[41] 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*, 28(11):1420–1428, 2012.

[42] 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. *Bioinformatics*, 31(10):1674–1676, 2015.

[43] Y. Lin and P.A. Pevzner. Manifold de Bruijn graphs. In *Proc. WABI*, 2014.

[44] I. Munro. Tables. In *Proc. FSTTCS*, pages 37–42, 1996.

[45] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

[46] Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[47] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci*, 426:25–41, 2012.

[48] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[49] Keith R. Bradnam et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):1–31, 2013.

[50] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 2013.

[51] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. *Bidirectional Variable-Order de Bruijn Graphs*, pages 164–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[52] Christina Boucher, Alexander Bowe, Travis Gagie, Giovanni Manzini, and Jouni Sirén. Relative select. In Costas Iliopoulos, Simon Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval*, pages 149–155, Cham, 2015. Springer International Publishing.

[53] R. Ronen, C. Boucher, H. Chitsaz, and P. Pevzner. SEQuel: Improving the accuracy of genome assemblies. *Bioinformatics (special issue of ISMB 2012)*, 28(12):i188–i196, 2012.

[54] M.D. Muggli, S.J. Puglisi, R. Ronen, and C. Boucher. Misassembly detection using paired-end sequence reads and optical mapping data. *Bioinformatics (special issue of ISMB 2015)*, 31(12):i80–i88, 2015.

[55] K.J. G.E. Robinson et al. Creating a buzz about insect genomes. *Science*, 331(6023):1386, 2011.

[56] M. Causse et al. Whole genome resequencing in tomato reveals variation associated with introgression and breeding events. *BMC Genomics*, 14:791, 2013.

[57] M. Kobayashi et al. Genome-wide analysis of intraspecific DNA polymorphism in "micro-tom", a model cultivar of tomato (solanum lycopersicum). *Plant Cell Physiology*, 55(2):445–454, 2014.

[58] D. Weigel and R. Mott. The 1001 genomes project for Arabidopsis thaliana. *Genome Biology*, 10(5):107, 2009.

[59] EMBL-EBI Metagenomics. Local surveillance of infectious diseases and antimicrobial resistance from sewage, 2016. Project (ERP015410), 25 October 2016, date last accessed.

[60] Ruth R Miller, Vincent Montoya, Jennifer L Gardy, David M Patrick, and Patrick Tang. Metagenomics for pathogen detection in public health. *Genome Medicine*, 5(9):1, 2013.

[61] F. Baquero et al. Metagenomic epidemiology: a public health need for the control of antimicrobial resistance. *Clinical Microbiology and Infection*, 18(4):67–73, 2012.

[62] Jesse A. Port, Alison C. Cullen, James C. Wallace, Marissa N. Smith, and Elaine M. Faustman. Metagenomic frameworks for monitoring antibiotic resistance in aquatic environments. *Environmental. Health Perspectives*, 122(3), 2014.

[63] The White House. National action plan for combating antibiotic-resistant bacteria. *Washington, DC*, 2015.

[64] Food and Agricultural Organization of the United Nations. The FAO action plan on antimicrobial resistance 2016-2020, 2016. Rome, Italy. 2016. (25 October 2016, date last accessed).

[65] Fernando Baquero, Ana-Sofia P Tedim, and Teresa M Coque. Antibiotic resistance shaping multi-level population biology of bacteria. *Frontiers in Microbiology*, 4:15, 2013.

[66] R Craig MacLean, Alex R Hall, Gabriel G Perron, and Angus Buckling. The population genetics of antibiotic resistance: integrating molecular mechanisms and treatment contexts. *Nature Reviews Genetics*, 11(6):405–414, 2010.

[67] Noelle R Noyes, Xiang Yang, Lyndsey M Linke, Roberta J Magnuson, Adam Dettenwanger, Shaun Cook, Ifigenia Geornaras, Dale E Woerner, Sheryl P Gow, Tim A McAllister, et al. Resistome diversity in cattle and the environment decreases during beef production. *eLife*, 5:e13195, 2016.

[68] Paula King, Long K Pham, Shannon Waltz, Dan Sphar, Robert T Yamamoto, Douglas Conrad, Randy Taplitz, Francesca Torriani, and R Allyn Forsyth. Longitudinal metagenomic analysis of hospital air identifies clinically relevant microbes. *PLoS ONE*, 11(8):e0160124, 2016.

[69] Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie–a data structure for pan-genome storage. *Algorithms in Bioinformatics*, pages 217–230, 2015.

[70] Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: A graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

[71] Y. Lin, S. Nurk, and P. Pevzner. What is the difference between the breakpoint graph and the de bruijn graph? *BMC Genomics*, 15(Suppl 6):S6, 2014.

[72] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016.

[73] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[74] Robert Mario Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[75] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[76] T. Tanaka et al. The rice annotation project database (RAP-DB): 2008 update. *Nucleic Acids Research*, 36:D1028–33, 2008.

[77] P. Schnable et al. The b73 maize genome: Complexity, diversity, and dynamics. *Science*, 326:1112–1115, 2009.

[78] D. Swarbreck et al. The Arabidopsis information resource (TAIR): gene structure and function annotation. *Nucleic Acids Research.*, 36:D1009–14, 2008.

[79] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics*, 30(15):2114–2120, 2014.

[80] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de bruijn graphs. *International Journal of Foundations of Computer Science*, 29(08):1279–1295, 2018.

[81] Diego Díaz-Domínguez, Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Gonzalo Navarro, and Simon J Puglisi. Assembling omnitigs using hidden-order de Bruijn graphs. *arXiv preprint arXiv:1805.05228*, 2018.

[82] Diego Díaz-Domínguez, Travis Ggie, and Gonzalo Navarro. Simulating the dna string graph in succinct space. *arXiv preprint arXiv:1901.10453*, 2019.

[83] Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de Bruijn graphs via merging. *bioRxiv*, 2019. `https://doi.org/10.1101/229641`.

[84] Lavinia Egidi, Felipe A Louza, and Giovanni Manzini. Space-efficient merging of succinct de Bruijn graphs. *arXiv preprint arXiv:1902.02889*, 2019.

[85] Bahar Alipanahi, Alan Kuhnle, and Christina Boucher. Recoloring the colored de Bruijn graph. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and Information Retrieval*, pages 1–11, Cham, 2018. Springer International Publishing.

[86] Harun Mustafa, Ingo Schilken, Mikhail Karasikov, Carsten Eickhoff, Gunnar Rätsch, and André Kahles. Dynamic compression schemes for graph coloring. *Bioinformatics*, 35(3):407–414, 2018.

[87] Isaac Turner, Kiran V Garimella, Zamin Iqbal, and Gil McVean. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics*, 34(15):2556–2565, 2018.

[88] Kingshuk Mukherjee, Bahar Alipanahi, Tamer Kahveci, Leena Salmela, and Christina Boucher. Aligning optical maps to de Bruijn graphs. *Bioinformatics*, 2018. `https://doi.org/10.1093/bioinformatics/btz069`.

[89] Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de Bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.

[90] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, and Marco Previtali. Fully dynamic de bruijn graphs. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval*, pages 145–152, Cham, 2016. Springer International Publishing.

[91] Phelim Bradley, Henk C Den Bakker, Eduardo P. C. Rocha, Gil McVean, and Zamin Iqbal. Real-time search of all bacterial and viral genomic data. *bioRxiv*, 2017. `https://doi.org/10.1101/234955`.

[92] Guillaume Marçais, Brad Solomon, Rob Patro, and Carl Kingsford. Sketching and sublinear data structures in genomics. *Annual Review of Biomedical Data Science*, 2(1), 2019.

[93] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

[94] D. Belazzougui, T. Gagie, S. Gog, G. Manzini, and J. Sirén. Relative FM-indexes. In *Proc. of SPIRE*, pages 52–64, 2014.

[95] J. Rozowsky et al. AlleleSeq: analysis of allele-specific expression and binding in a network framework. *Molecular Systems Biology*, 7:522, 2011.