Simulating the DNA String Graph in Succinct Space

Diego Díaz-Domínguez\textsuperscript{1,2}, Travis Gagie\textsuperscript{1,4}, and Gonzalo Navarro\textsuperscript{1,2}

\textsuperscript{1} CeBiB — Center for Biotechnology and Bioengineering, Chile
\textsuperscript{2} Department of Computer Science, University of Chile, Chile. 
\textquoteleft\{diediaz,gnavarro\}'@dcc.uchile.cl
\textsuperscript{3} School of Computer Science and Telecommunications, Diego Portales
University, Chile. travis.gagie@gmail.com

\begin{abstract}
Converting a set of sequencing reads into a lossless compact data structure that encodes all the relevant biological information is a major challenge. The classical approaches are to build the string graph or the de Bruijn graph. Each has advantages over the other depending on the application. Still, the ideal setting would be to have an index of the reads that is easy to build and can be adapted to any type of biological analysis. In this paper we propose a new data structure we call rBOSS, which gets close to that ideal. Our rBOSS is a de Bruijn graph in practice, but it simulates any length up to $k$ and can compute overlaps of size at least $m$ between the labels of the nodes, with $k$ and $m$ being parameters. If we choose the parameter $k$ equal to the size of the reads, then we can simulate a complete string graph. As most BWT-based structures, rBOSS is unidirectional, but it exploits the property of the DNA reverse complements to simulate bi-directionality with some time-space trade-offs. We implemented a genome assembler on top of rBOSS to demonstrate its usefulness. Our experimental results show that, using $k=100$, rBOSS can assemble 185 MB of reads in less than 15 minutes and using 110 MB in total. It produces contigs of mean sizes over 10,000, which is twice the size obtained by using a pure de Bruijn graph of fixed length $k$.
\end{abstract}

1998 ACM Subject Classification J.3 Life and Medical Sciences, E.1 Data Structures, G.2.2 Graph Theory

Keywords and phrases String graph, de Bruijn graph, DNA sequencing, Succinct ordinal trees

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Obtaining and extracting the relevant information from a collection of DNA reads, for assembly and other analysis purposes, usually requires a lot of time and space. The techniques for compressed indexing developed in recent years (see [24] for a full review) have significantly contributed to reduce the computational costs. There is still no technique, however, that can preprocess the reads and represent all the relevant information in succinct space so that it can be used effectively.

The classical plain, and lossless, data structure to analyze reads is the \textit{string graph}. In this model, each node represents a particular read, and two nodes $v$ and $v'$ are connected by an edge with weight $o \geq m$ if $o$ is the maximum length of a suffix of $v$ that matches a prefix

* Partially supported by by Basal Funds FB0001, Conicyt, Chile; by a Conicyt PhD Scholarship; and by Fondecyt Grant 1-171058, Chile.
of $v'$, where $m$ is a parameter to filter out spurious overlaps. Computing the string graph from a set of reads is not difficult: it can be built from the suffix tree of the set or even from its $BWT$ [30]. Representing the graph, however, is expensive: a quadratic number of edges may have to be stored. A popular technique to reduce the space is to delete transitive edges, but this approach limits the usefulness of the structure. Historically, string graphs have been used in the context of genome assembly [10, 22, 33], but as sequencing datasets have grown over the years, they have been discarded in favor of other lossy, but more succinct, representations.

The de Bruijn (dBG) graph emerged as an alternative to string graphs. This data structure encodes the relationship between all the substrings of length $k$ in the set. A dBG is easy to construct and it can be represented succinctly [5]. Besides, it encodes the context of the substrings of length $k - 1$ in its topology. Thus, for instance, if a substring of length $k - 1$ appears in several contexts of the set, then its dBG node will have several edges. As the string graph, the first application of the dBG was the assembly of genomes [2, 19, 28, 32], but through the years, its use has been extended to other kinds of analysis [6, 14].

The disadvantage of dBGs, however, is that they are lossy, because the information we can retrieve from them is limited by $k$. A way to overcome some of the restrictions imposed by $k$ is to add variable-order functionality to the dBG, that is, to encode several dBGs with different values for $k$ at the same time. The contexts of the graph can then be shortened or lengthened on demand, depending on the need to have more or less edges from a node. Some succinct dBG representations supporting variable-order functionality up to some maximum order $k$ have been proposed [1], but they increase the space requirements by a $\log k$ factor. Besides, even using variable-order functionality, the data structure remains lossy, because the order of the graph can be lengthened only up to $k$. By choosing $k$ equal to the size of the reads, the variable-order dBG becomes lossless, and equivalent to the string graph, but the $\log k$ factor becomes significant if the size of the reads is considerable.

Almost every analysis over DNA sequencing data can be reduced to looking for suffix-prefix overlaps between the reads. Because the dBG is a lightweight representation of the overlaps, even if lossless, it has adapted well to many bioinformatic applications. Still, searching for biological signals in a dBG requires the detection of complex graph substructures such as bubbles, super bubbles, tips, and so on, and those can be expensive to find. The string graph, on the other hand, is a much simpler model. It can be adapted to different applications other than assembly in the same way as the dBG, but it has the advantage of being lossless and not requiring too much preprocessing after its construction. The problem, as stated before, is the space to encode the edges of the graph. A better approach would be to have a compact data structure that can quickly compute the overlaps (i.e., the edges) on the fly instead of storing them explicitly. Such a solution would require a moderate preprocessing of the read set, would retain all the information, and would use a reasonable amount of space.

Our contribution. In this paper we address the problem of succinctly representing and analyzing a collection $\mathcal{R}$ of sequencing reads. To this end, we define a new compact data structure we call $rBOSS$. It is an intermediate structure between a dBG and a string graph, because it can compute the context of the sequences in the same way a dBG does, but it can also compute on the fly the overlaps between different substrings of $\mathcal{R}$. The $rBOSS$ index is based on $BOSS$ [5], a BWT-based representation of dBGs, which is augmented with a tree we call the overlap tree. This tree increases the size of the data structure by $4n + o(n)$ bits, where $n$ is the number of nodes in the dBG encoded by $BOSS$. By choosing $k$ equal to the length of the reads, we can simulate in compressed space a string graph whose edges have a
weight \( o \geq m \), where \( m \) is given as a parameter. The simulation of the string graph builds on the basic primitives \texttt{nextcontained} and \texttt{buildL}. Our overlap tree allows us reducing their time complexity from \( \mathcal{O}(k^2) \) to \( \mathcal{O}(1) \) and \( \mathcal{O}(k) \), respectively.

In addition to \( r\text{BOSS} \), we also formalize the idea of weighting the string graph edges according to transitive connections, and explain how this new weighting scheme can be used to solve biological problems other than assembly. Finally, we demonstrate the usefulness of \( r\text{BOSS} \) by implement a genome assembler on top of it. Our experimental results show that, by using \( k = 100 \), \( r\text{BOSS} \) can assemble 185 MB of reads in less than 15 minutes and using 110 MB in total. It produces contigs of mean sizes over 10,000, which is twice the size obtained by using a pure dBG of fixed length \( k \).

## 2 Preliminaries

**DNA strings.** A DNA sequence \( R \) is a string over the alphabet \( \Sigma = \{a, c, g, t\} \) (which we map to \([1..\sigma]\) ), where every symbol represents a particular nucleotide in a DNA molecule. The DNA \textit{complement} is a permutation \( \pi[1..\sigma] \) that reorders the symbols in \( \Sigma \) exchanging \texttt{a} with \texttt{t} and \texttt{c} with \texttt{g}. The reverse complement of \( R \), denoted \( R^{rc} \), is a string transformation that reverses \( R \) and then replaces every symbol \( R[i] \) by its complement \( \pi(R[i]) \).

**De Bruijn graphs.** A de Bruijn Graph (dBG) \([9]\) of order \( k \) of a set of strings \( \mathcal{R} = \{R_1, R_2, \ldots, R_\nu\} \), \( \text{DBG}_{R,k} \), is a labeled directed graph \( G = (V, E) \) where every node \( v \in V \) is labeled by a distinct substring of \( \mathcal{R} \) of length \( k - 1 \), and every edge \((v, u) \in E \) represents the substring \( S[1..k] \) of \( \mathcal{R} \) such that \( v \) is labeled by the prefix \( S[1..k-1] \), \( u \) is labeled by the suffix \( S[2..k] \), and the label of \((v, u) \) is the symbol \( S[k] \). We identify a node with its label.

A \textit{variable-order dBG (vo-dBG)} \([4]\), \( \text{voDBG}_{R,k} \), is formed by the union of all the graphs \( \text{DBG}_{R,k'} \), with \( 1 \leq k' \leq k \). Each \( \text{DBG}_{R,k} \) represents a \textit{context} of \( \text{voDBG}_{R,k} \). In addition to the (directed) edges of each \( \text{DBG}_{R,k} \), two nodes \( v \in \text{DBG}_{R,k} \) and \( v' \in \text{DBG}_{R,k'} \), with \( k' > k \), are connected by an undirected edge \((v, v')\) if \( v' \) is a suffix of \( v \). Following the edge \((v, v')\) or \((v', v)\) is called a \textit{change of order}. We then identify node order with length.

**BOSS representation for de Bruijn graphs.** BOSS \([5]\) is a succinct data structure, similar to the \textit{FM-index} \([11]\), for encoding dBGs. In BOSS, the nodes are represented as rows in a matrix of \( k - 1 \) columns, and are sorted in reverse lexicographical (i.e., reading the labels right to left). All the edge (one-symbol) labels of the graph are stored in a unique sequence \( E \) sorted by the BOSS order of the source nodes, so the symbols of the outgoing edges of each node fall in a contiguous range. A bitmap \( B \) of size \( e = |E| \) marks the last outgoing symbol in \( E \) of every dBG node. Finally, an array \( C[1..\sigma] \) stores in \( C[i] \) the number of node labels that end with a symbol lexicographically smaller than \( i \).

Prefixes in \( \mathcal{R} \) of size \( d < k \) are artificially represented in BOSS as \( k \) strings padded at the left with \( k - d \) symbols \$, where \$ is the smallest alphabet symbol. The symbol \$ is also referred to as the \textit{dummy} symbol. Equivalently, suffixes of size \( d < k \) are represented as \( k \) strings padded at the right with \( k - d \) dummy symbols. For this work, however, suffixes of size \( d < k \) are not necessary. Strings formed by all \$s are also called dummy.

The complete index is thus composed of the vectors \( E \), \( C \), and \( B \). It can be stored in \( c(\mathcal{H}_0(E) + \mathcal{H}_0(B))(1 + o(1)) + \mathcal{O}(\sigma \log n) \) bits, where \( \mathcal{H}_0 \) is the zero-order empirical entropy \([23\text{ Sec 2.3}]\). This space is reached with a Huffman-shaped Wavelet Tree \([21]\) for \( E \), a compressed bitmap \([29]\) for \( B \) (as it is usually very dense), and a plain array for \( C \).
Simulating the DNA string graph in succinct space

BOSS supports several navigational queries, most of them within $O(\log \sigma)$ time or less. The most relevant ones for us are:

- $\text{outdegree}(v)$: number of outgoing edges of $v$.
- $\text{forward}(v, a)$: node reached by following an edge from $v$ labeled with symbol $a$.
- $\text{indegree}(v)$: number of incoming edges of $v$.
- $\text{backward}(v)$: list of the nodes with an outgoing edge to $v$.

Boucher et al. [4] noticed that by considering just the last $k^i - 1$ columns in the BOSS matrix, with $k^i \leq k$, the resulting nodes are the same as those in the dBG of order $k^i$. To allow changing the order of the dBG in BOSS, they augmented the data structure with the longest common suffix (LCS) array. The LCS array stores, for every node of order $k$, the size of the longest suffix shared with its predecessor node in the BOSS matrix. They called this new index the variable-order BOSS (VO-BOSS), which supports the following additional operations:

- $\text{shorter}(v, k')$: range of the nodes suffixed by the last $k'$ characters of $v$.
- $\text{longer}(v, k')$: list of the nodes of length $k'$ that end with $v$.
- $\text{maxlen}(v, a)$: a node in the index suffixed by $v$, and that has an outgoing edge labeled $a$.

By using a Wavelet Tree [13], the LCS can be stored in $n \log (k - 2) + o(n \log k)$ bits, and the functions $\text{shorter}$ and $\text{longer}$ can be answered in $O(\log k)$ and $O(|U| \log k)$ time, respectively. The function $\text{maxlen}$ is implemented using the arrays $B$ and $E$ from BOSS, and hence, it is answered in $O(\log \sigma)$ time.

Succinct representation of ordinal trees. An ordinal tree $T$ with $n$ nodes can be stored succinctly as a sequence of balanced parentheses (BP) encoded as a bit vector $B[1..2n]$. Every node $v$ in $T$ is represented by a pair of parentheses ( . . ) that contain the encoding of the subtree rooted at $v$. Every node of $T$ can be identified by the position in $B$ of its open parenthesis. Many navigational operations over $T$ can be simulated over $B$ in constant time, by using a structure that requires $o(n)$ bits on top of $B$ [20].

3 rBOSS

Basic definitions. Let $\mathcal{R} = \{R_1, \ldots, R_r\}$ be a collection of $r$ reads (strings) of length $z$ and let $\mathcal{R}^{rc}$ be a collection, also of $r$ strings, with every $R_i \in \mathcal{R}^{rc}$ being the reverse complement of one $R_j \in \mathcal{R}$. Adiitonally, we define the set $\mathcal{R}^* = \mathcal{R} \cup \mathcal{R}^{rc}$. Let us denote $G = \text{DBG}_{\mathcal{R}^*,k}$ and $G' = \text{voDBG}_{\mathcal{R}^*,k}$. A traversal $P$ over $G$, or $G'$, is a sequence $(v_0, \epsilon_0, v_1, \ldots, v_{t-1}, \epsilon_t, v_t)$ where $v_0, v_1, \ldots, v_{t-1}, v_t$ are nodes and $\epsilon_1, \ldots, \epsilon_t$ are edges, $\epsilon_i$ connecting $v_{i-1}$ with $v_i$. $P$ will be a path if all the nodes are different, except possibly the first and the last. In such case, $P$ is said to be a cycle. $P$ is unary if the nodes $(v_0, v_{t-1})$ have outdegree 1 and the nodes $(v_1, v_t)$ have indegree 1. $P$ will be a right traversal over $G$ or $G'$ if all its edges $\epsilon_i$ are directed from $v_{i-1}$ to $v_i$, and a left traversal if all its edges $\epsilon_i$ are directed from $v_i$ to $v_{i-1}$. The string formed by the concatenation of the edge symbols of $P$ is referred to as its label. $P$ will be safe if it is a path or a cycle and its label appears in $\mathcal{R}^*$ as a substring of some read or if it can be generated by overlapping two or more $R_i, R_j \in \mathcal{R}^*$ in tandem and then taking the string that results from the union of those reads. The overlaps between the reads have to be of minimum size $m$.

Let $BOSS(G)$ and $VO-BOSS(G')$ be the BOSS and VO-BOSS indexes, respectively, for the graphs $G$ and $G'$. In both cases, the matrix with the $(k-1)$-length node labels is referred
to as $M_{R^*, k}$, or just $M$ when the context is clear. In $VO$-$BOSS(G')$, the range of rows in $M$ suffixed by $v$ is denoted $\overrightarrow{v}$.

Rows of $M$ representing substrings of size $k - 1$ in $R^*$ are called solid nodes and rows representing artificial $(k - 1)$-length strings padded with dummy symbols from the left, and that represent prefixes in $R^*$, are called linker nodes. For a linker node $v$, the function $llabel(v)$ returns the non-$\$ suffix of $v$. A solid node that appears as a suffix in $R$ is called an s-node and a solid node that appears as a prefix in $R^*$ is called a p-node. A linker node $v$ is said to be contained within another node $v'$ (solid or linker) if $llabel(v)$ is a suffix of $v'$.

An overlap of size $o$ between two solid nodes $v$ and $v'$, denoted $v \oplus^o v'$, occurs when the $o$-length suffix of $v$ is equal to the $o$-length prefix of $v'$. Relative to $v$, $v \oplus^o v'$ is a forward overlap and $v' \oplus^o v$ is a backward overlap. An overlap $v \oplus^o v'$ is valid if (i) $m \leq o < k - 2$, with $m$ being some parameter, and $v'$ is a p-node, or (ii) $o = k - 2$ and $v'$ is a solid node of any kind. Notice that case (ii) is equivalent to the definition of two dBG nodes connected by an edge. The overlap $v \oplus^o v'$ is considered transitive if there is another solid node $v''$ with valid overlaps $v \oplus^o v''$ and $v' \oplus^o v''$, with $o' > o''$. If there is only one $v''$, then $v \oplus^o v'$ is transitive and unique. If such $v''$ does not exist, then $v \oplus^o v'$ is irreducible. The string formed by the union of the solid nodes $v$ and $v'$ is denoted as $\overrightarrow{v \oplus^o v'}$.

### Link between variable-order and overlaps

Overlaps between reads in $R^*$ can be computed using $VO$-$BOSS(G')$ as follows: extend a unary path using solid nodes as much as possible, and if a solid node $v$ without outgoing edges is reached, then decrease its order with shorter to retrieve the vo-dBG nodes that represent both a suffix of $v$ and a prefix of some read in $R$. From these nodes, retrieve the overlapping solid nodes of $v$ by using forward and continue the graph right traversal from one of them.

In $VO$-$BOSS$, however, shorter does not ensure that the label of the output node appears as a prefix in $R^*$. The next lemma precisely the condition that must hold to ensure this.

**Lemma 1.** In $VO$-$BOSS$, applying the operation shorter to a node $v$ of order $k' \leq k$ will return a node $u$ of order $k'' < k'$ that encodes a forward overlap for $v$ only if $\overrightarrow{u}[1]$ is a linker node contained by $v$.

**Proof.** If all the left contexts in $\overrightarrow{u}$ are non-dummy strings, then $u$ does not appear as a prefix in $R^*$, and hence, following none of its edges will lead to a valid overlap of $v$. On the other hand, if a suffix $u$ of $v$ appears as a prefix in $R^*$, then there is a node $v'$ in $G$ at order $k$ whose label is formed by the concatenation of a dummy string and $u$, and that by definition is a linker node contained by $v$. Since elements in $\overrightarrow{u}$ are sorted by the left contexts, $v'$ is placed in $\overrightarrow{u}[1]$, because the dummy string is always the lexicographically smallest. ▶

Lemma 1 allows us to find overlapping nodes that are not directly linked via edges in the dBG, by looking in smaller dBG orders. We formalize this operation as follows, where we look for the longest valid suffix, that is, the one with maximum lexicographic index.

- **nextcontained**$(v)$: returns the greatest linker node $v'$, in lexicographical order, whose $llabel$ represents both a suffix of $v$ and a prefix of some other node in $G'$.

**Theorem 2.** There is an algorithm that solves nextcontained in $O(k^2 \log \sigma)$ time.

**Proof.** Incrementally decrease the order of $v$ by one until reaching a node $u$ with $\overrightarrow{u} \supseteq \overrightarrow{v}$, and that satisfies Lemma 1. If such $u$ exists, return it. If the order of $v$ decreases below $m$ before finding $u$, then $v$ does not contain any linker node $v'$ with $llabel(v') \geq m$. In such case, a dummy vo-dBG node is returned. The function reduces the order up to $k - 2$ times.
Simulating the DNA string graph in succinct space

In each iteration, the operations shorter and llabel (to check Lemma 1) are used, which take \( O(\log k) \) and \( O(k \log \sigma) \) time, respectively. Thus, the total time is \( O(k^2 \log \sigma) \). ▶

Notice, however, that a vo-dBG node in \( G' \) might have more than one contained linker node, and those linkers whose llabel is of length \( \geq m \) represent edges in the overlap graph. A useful operation is to build a set \( \text{nextcontained} \) whose leftmost child edge is labeled by a $.

\( \text{nextcontained} \) are recursively connected to the parent of \( t \) if it is compact. First, all the internal nodes of \( T \) with string depth below \( m \) are discarded, and the subtrees left are connected to the root of \( T' \). Second, every internal node \( t' \in T \) whose leftmost-child edge is not labeled by a dummy string is also discarded, and its children are recursively connected to the parent of \( t' \). Note that all the leaves of \( T \) are in \( T' \).

Function \( \text{buildL} \) applies \( \text{nextcontained} \) iteratively until reaching a node \( u \) that contains a linker node whose llabel has length below \( m \). The rationale is that if \( v \) contains \( v' \), and in turn \( v' \) contains \( v'' \), then \( v \) also contains \( v'' \). Algorithm 1 shows the pseudo code.

Note that, if we chose \( k = z + 1 \) to build VO-BOSS(\( G' \)), then we are simulating the full overlap graph in compressed space. The edges are not stored explicitly, but computed on the fly by first obtaining \( L = \text{buildL}(v, m) \), and then following the dBG outgoing edges of every \( l \in L \). Still, the complexities of the involved operations \( \text{nextcontained} \) makes VO-BOSS slow for exhaustive traversals, which is our main interest. We design a faster scheme in which follows; Figure 1 exemplifies the various concepts.

A compact data structure to compute overlaps. The function \( \text{buildL} \) can be regarded as a bottom-up traversal of the trie \( T \) induced by the \((k - 1)\)-length labels of \( M \) read in reverse. Every trie node \( t \) corresponds to a vo-dBG node \( v \) whose order is the string depth of \( t \). The traversal starts in the trie leaf \( t \) corresponding to the vo-dBG node \( v \) given to \( \text{buildL} \), and continues upward until finding the last ancestor \( t' \) of \( t \) with string depth \( \geq m \). The movement from \( t \) to \( t' \) can be regarded as a sequence of applications of \( \text{nextcontained} \).

In each such application, we move from a node \( t \) to its nearest ancestor \( t' \) that is maximal (i.e., has more than one child) and whose leftmost child edge is labeled by a $.

Since non-maximal nodes in \( T \) are not relevant for building \( L \), the function \( \text{nextcontained} \) can be reimplemented using the topology of the compact trie \( T \) (i.e., collapsing unary paths) represented with BP (see Section 2) instead of using shorter. In this way, we can get rid of the LCS structure of VO-BOSS.

The resulting rBOSS index can be built in linear time, as detailed in Appendix D.

Replacing the LCS with the topology of \( T \) in BP poses two problems, though. First, it is not possible to define a minimum dBG order \( m \) from which overlaps are not allowed, and second, Lemma 1 cannot be checked. Both problems arise because, unlike the LCS, the BP data structure does not encode the string depths of the tree nodes (and thus, the represented node lengths). Still, the topology of \( T \) can be reduced to precisely the nodes of interest for \( \text{nextcontained} \), and thus avoid any check.

▶ Theorem 3. There is a structure using \( 4 + o(1) \) bits per dBG node that implements the function \( \text{nextcontained} \) in \( O(1) \) time and the function \( \text{buildL} \) in time \( O(k) \).

Proof. The structure is the BP encoding of a tree \( T' \) that is obtained by removing some nodes from the compact trie \( T \) (\( T \) has one leaf per dBG node, and less than \( 2 \) nodes per dBG node because it is compact. First, all the internal nodes of \( T \) with string depth below \( m \) are discarded, and the subtrees left are connected to the root of \( T' \). Second, every internal node \( t' \in T \) whose leftmost-child edge is not labeled by a dummy string is also discarded, and its children are recursively connected to the parent of \( t' \). Note that all the leaves of \( T \) are in \( T' \).
Therefore, $T'$ has precisely the nodes of interest for operation \texttt{nextcontained}(v). We simply find the $i$th left-to-right leaf $t$ in $T'$, where $i$ is the row of $M$ corresponding to $v$ (note that rows of $M$ and leaves of $T$ and $T'$ are in the same order). Then, we move to the parent $t'$ of $t$ and return its leftmost child. An exception occurs if the leftmost child of $t'$ is precisely $t$, which means that $v$ is a linker node and thus its next contained node is the leftmost child of the parent of $t'$. Finally, we return the rank of the desired leftmost leaf.

Algorithm 2 shows the pseudocode. All its operations are implemented in constant time in BP, and thus \texttt{nextcontained}(v) is implemented in $O(1)$ time. The function \texttt{buildL} stays the same, and its cost is dominated by the (at most) $k-2$ calls to \texttt{nextcontained}. \hfill ▶

Once the set $L$ with the contained nodes of $v$ is built, we can compute the valid forward overlaps of $v$ by following the edges of every $l \in L$ until finding a p-node. We then define:

- $\texttt{foverlaps}(v)$: the set of p-nodes whose prefixes overlap a suffix of $v$ of length $\geq m$.

Computing the forward overlaps of $v$ by following the edges of every $l \in L$ can be exponential. We devise a more efficient approach that uses $T'$ and the reverse complements of the node labels. We need to define first the idea of bi-directionality in rBOSS.

\textbf{Simulating bi-directionality.} When building rBOSS on $\mathcal{R}^*$, the reverse complement $R_i^c$ of every read $R_i \in \mathcal{R}$ is also included, because there are several combinations in which two reads, $R_i$ and $R_j$, can have a valid suffix-prefix overlap: $(R_i, R_j^c)$, $(R_i, R_j^c)$, $(R_i^c, R_j^c)$, or $(R_i^c, R_j^c)$, and all must be encoded in $T'$. An interesting consequence of including the reverse complements is that the topology of the dBG becomes symmetric.

\textbf{Lemma 4.} The incoming symbols of a node $v$ are the DNA complements of the outgoing symbols of the node $v^c$ that represents the reverse complement of $v$. Further, the outgoing nodes of $v^c$ are the same as the DNA complements of the incoming nodes of $v$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.pdf}
\caption{A segment of rBOSS for the set $\mathcal{R}=$\{\texttt{atttggagta,gtattggaa,agtttggaa,caaatcttca}\}, with parameters $k=11$ and $m=2$. The computation of the forward overlaps for $R_1 = \mathcal{R}[1]$ are also shown. A) range of $M$ that includes the solid node $v$ that represents sequence $R_1$ (sequence in bold in row 8) and its contained nodes (grayed rows). B) subtree of $T'$ that maps the range of $M$ in A). The black leaf maps $v$ in $T'$ and the gray leaves map the linker nodes contained by $v$. The array of parentheses below represents the same subtree in $BP$. The rank of every leaf (\texttt{leafrank}) is shown above its relative position in the BP array (\texttt{leafselect}). Each dashed arrow represents a call of the function \texttt{nextcontained} and all the arrows together stand for the function \texttt{buildL}. C) Forward overlaps of $v$. Gray sequences in italic are those reads where the reverse complement matches with $v$.}
\end{figure}
Simulating the DNA string graph in succinct space

Proof. Consider the \((k - 1)\)-length substring \(bx_c\) of \(R\), and a symbol \(a\) that appears at the left of some occurrences of \(bx_c\). For building \(rBOSS\), both substrings \(abXc\) and its reverse complement \((abXc)^{rc} = c^\epsilon X^{rc}b^\epsilon\) are considered. As a result, the dBG node \(v\) labeled \(bx_c\) will have and incoming symbol \(a\), and the dBG node \(v^{rc}\) labeled \((bx_c)^{rc} = c^\epsilon X^{rc}b^\epsilon\) will have an outgoing symbol \(a^\epsilon\). Thus, the label of node \(\text{forward}(v^{rc}, a^\epsilon)\) will be \(X^{rc}b^\epsilon\), which is the reverse complement of string \(abX\), the label of node \(\text{backward}(v,a)\).

An important observation about Lemma 4 is that if \(v\) has order \(k' < k\), then the subtree of \(T'\) rooted at the node that maps \(v^{rc}\) encodes the topology of the forward overlaps of \(v\). We can use this idea to quickly compute the reverse complements of the dBG nodes via their reverse complements. We have to define first how to map \(v\) to \(v^{rc}\) and vice versa in \(rBOSS\).

Theorem 5. Computing the position in \(M\) of \(v^{rc}\) takes \(O(k \log \sigma)\) time. By augmenting \(rBOSS\) with \(s \log s\) extra bits, \(s\) being the number of solid nodes, the time decreases to \(O(1)\).

Proof. First, extract the label \(\text{lab}\) of \(v\), then compute its reverse complement \(\text{lab}^{rc}\), and finally, perform \(\text{backwardsearch}(\text{lab}^{rc})\). The label of \(v\) is extracted in time \(O(k \log \sigma)\) with the FM-index, and computing its reverse complement takes \(O(k)\) time. The function \(\text{backwardsearch}\), also defined on the FM-index, returns the range of \((k - 1)\)-length strings in \(M\) suffixed by \(\text{lab}^{rc}\), and it also takes \(O(k \log \sigma)\) time. Therefore, computing the position of \(v^{rc}\) in \(M\) takes \(O(k \log \sigma)\) time. Alternatively, we can store an explicit permutation on the \(s\) solid nodes, so that using \(s \log s\) bits we find the position of \(v^{rc}\) in \(M\) in constant time.

Theorem 6. The function \(\text{foverlaps}\) can be computed in \(O(k \log \sigma)\) time.

Proof. First, create \(L_v = \text{buildL}(v)\), and then obtain the reverse complement of the linker node \(l = \text{L}_v[\text{L}_v]\), that is, the one representing the smallest suffix of \(v\). Second, compute \(l^{rc}\) and search for the range \([i,j]\) in \(M\) of the \((k - 1)\)-length strings suffixed by \(l^{rc}\). From the edge symbols in \([i,j]\) follow the dBG path \(p_{v^{rc}}\) that spells the label of \(v^{rc}\). Finally, every time a solid node \(v'\) is reached during the traversal of \(p_{v^{rc}}\), report its reverse complement as a forward overlap for \(v\). Computing \(L\) takes \(O(k)\) time. Both searching for \([i, i]\) and traversing \(p_{v^{rc}}\) take \(O(k \log \sigma)\) time. All the shifts between reverse complements take \(O(1)\) time if we use permutations.

Note that backward overlaps can be obtained by computing \(\text{foverlaps}\) for the reverse complement of \(v\). The complexity of \(\text{foverlaps}\) is the same obtained by Simpson and Durbin [31]. Our advantage, however, is that we have access to the topology of the overlaps via \(T'\), and this can be used to retrieve extra information from the data that irreducible string graphs or dBGs do not have. We formalize this idea as weighted irreducible overlaps.

Weighting irreducible overlaps. Given an irreducible overlap \(v \ominus^o v\) between solid nodes \(v\) and \(v'\), we can use the number of unique transitive overlaps between them as a measure of confidence, \(\text{weight}(v \ominus^o v') < o - m\), for \(\text{label}(v \ominus^o v')\). In Figure A.2 we show different examples in which \(\text{weight}(v \ominus^o v')\) can be helpful to detect patterns in the data. The function \(\text{foverlaps}(v)\) in our scheme can be modified to return the list of irreducible overlaps for \(v\), with their weights included. The idea is as follows: once the range \([i,j]\) is obtained, we form an array \(Y\) with the dBG nodes in \([i,j]\) that are not contained by any other node within the same range. The set \(Y\) will represent the possible irreducible overlaps of \(v\). \(Y\) is built in one scan over \([i,j]\) by checking which \(T'\) leaves are not the leftmost children of their parent. The weight of every \(y \in Y\) is computed as its depth minus the depth of its closest ancestor in \(T'\) with more than two children. We do the subtraction because only unique transitive
connections count as weights. Every \( y \in Y \) and its weighting nodes form a subrange \( q_y \) in \([i,j]\). We perform a right traversal starting from the outgoing edges of \( q_y \) to retrieve \( p_{v^c} \) as before. In the process, however, one or more elements of \( Y \) can be discarded or their weights decreased if they do have a branch spelling the reverse complement of some \( l \in L_v \). Figure A.3 exemplifies the process.

## 4 Experiments

We implemented \( rBOSS \) as a C++ library, using the SDSL library \([12]\) as a base. In Section 2 we stated that vector \( E \) can be represented using a Huffman-shaped Wavelet Tree, but our implementation uses run-length encoding \([21]\) to exploit repetitions in the reads. We also include an extra bitmap \( S[1..n] \) that marks the position of every solid node in \( M \), which speeds up iterating over the solid nodes. Our complete code is available at https://bitbucket.org/DiegoDiazDominguez/eboss-dt/src/master/. The compilation flags we used were `-msse4.2 -O3 -funroll-loops -fomit-frame-pointer -ffast-math`.

We used wgsim \([20]\) to simulate a sequencing dataset (in FASTQ format) from the E.coli genome with 15x coverage. A total of 549,845 reads were generated, each 150 bases long, yielding a dataset of 185 MB. The input parameters for building \( rBOSS \) are \( k \) and \( m \). We used a minimum value of 50 for \( k \), and increased it up to 110 in intervals of 5. For every \( k \), we used 6 values of \( m \), from 15 to 40, also in intervals of 5. This makes up 72 indexes.

The sizes of the resulting indexes are shown in Figure 2, which grow fairly linearly with \( k \), at 0.29 \(+0.036k \) bits per input symbol (i.e., 50–100 MB for our dataset), and do not depend much on \( m \). Figure A.4 shows elapsed times and memory peaks during construction. These are also linear in \( k \); for example, with \( m = 20 \) (the most demanding value) the \( rBOSS \) index for our dataset is built in 4–6 minutes with a memory peak around 2.5 GB.

The space breakdown of our index is given in Figure A.5 and further statistics in Table A1. The most expensive data structure in terms of space (50%–65%) is the BP representation of \( T' \). The sequence \( E \) uses 20%–35%, and the rest are the bitmaps \( B \) and \( S \).

For every index, we took 1000 solid nodes at random and computed the mean elapsed time for functions `nextcontained`, `buildL`, `foverlaps` and `reversecomplement`. Table 1 shows the results. The function `nextcontained` is the fastest, and its elapsed time is similar across different values of \( m \) and \( k \), around 1.5 \( \mu \)sec. The function `buildL` becomes slower as we increment \( k \), but faster as we increment \( m \). This is expected because the larger \( k \), the longer the traversal through \( T' \), but if \( m \) grows the traversal shortens as well. In all cases, `buildL` takes under 10 \( \mu \)sec. The cost of function `foverlaps` grows linearly with \( k \) and with \( m \) reaching 1–2 milliseconds. This is much slower than previous operations, and grows with \( m \) because the time to find the reverse complement of the shortest linker node, which is done with a backward search, dominates. Finally, the time of `reversecomplement` is also a few milliseconds, growing steadily with \( k \) regardless of \( m \).

We implemented a genome assembler on top of \( rBOSS \) to test the usefulness of the data structure (algorithm described in Appendix E). We used the same E. coli dataset as before, with a minimum value for \( k \) of 60, increasing it up to 100 in intervals of 5 for building the indexes. For each \( k \), we selected 5 values for \( m \), from 30 to 50, also in intervals of 5. The results are shown in Figure 3; time and space are again linear in \( k \). Using \( m = 30 \), our assembler generates contigs in 7–14 minutes and has a memory peak of 70–105 MB, just 18–21 MB on top of the index itself.

Figure 3.C compares the quality of the assembly using variable \( k \) and \( rBOSS \), with \( m = 30 \), versus the corresponding assembly generated with a fixed dBG that uses the same \( k \). The
Simulating the DNA string graph in succinct space

Figure 2 Sizes of the rBOSS indexes generated. Shapes are the different values for m.

Table 1 Mean elapsed time, in µseconds, for the functions proposed in this article.

| k   | m   | next-contained | buildl | fovertaps | reverse-complement |
|-----|-----|----------------|--------|-----------|--------------------|
| 50  | 20  | 1.49           | 5.09   | 920.63    | 1226.53            |
| 50  | 30  | 1.53           | 4.22   | 1035.14   | 1209.02            |
| 50  | 40  | 2.00           | 3.38   | 1083.23   | 1226.56            |
| 70  | 20  | 1.55           | 5.66   | 1126.44   | 1620.22            |
| 70  | 30  | 1.57           | 5.82   | 1206.69   | 1620.78            |
| 70  | 40  | 1.54           | 5.26   | 1285.46   | 1621.98            |
| 90  | 20  | 1.73           | 8.11   | 1329.24   | 2013.00            |
| 90  | 30  | 1.58           | 7.35   | 1408.68   | 2012.36            |
| 90  | 40  | 1.54           | 6.67   | 1494.98   | 2016.41            |
| 110 | 20  | 1.67           | 9.25   | 1530.05   | 2411.10            |
| 110 | 30  | 1.77           | 8.64   | 1616.05   | 2410.03            |
| 110 | 40  | 1.64           | 8.10   | 1701.20   | 2414.11            |

Figure 3 Results of genome assembly experiments. A) The elapsed time for the assembly. B) The memory peak achieved during the assembly. C) Comparison of the mean contig size in rBOSS (gray line) versus the mean unitig size of a fixed dBG (black line) using the same k, both with m = 50.

dBG indexes were built using the bcalm tool [7]. It is clear that the ability to vary the value of k to compute overlapping sequences as we spell the contigs, also called maximal paths (MP) in our algorithm, yields an assembly of much higher quality. Figure A.6 gives further data on the assembly. We did not manage yet to run the assembly on the competing VO-BOSS index [4], but Figure A.7 compares their size, showing that rBOSS is more than 20% smaller on average. We plan to also compare assembly times for the final version.

5 Conclusions and Further Work

We have introduced rBOSS, a succinct representation for vo-dBGs (of degree up to k) that avoids the $O(\log k)$-bit penalty factor of previous representations thanks to the use of a new structure we call the overlap tree. This enables the use of k values sufficiently large so as to simulate the full overlap graph, which is an essential tool for genome assembly and other bioinformatic analyses. Our index, for example, can assemble the contigs of 185 MB of 150-base reads, with $k = 100$, in less than 15 minutes and within 105 MB.

Our index builds fast, yet using significant space (in our experiment, 6 minutes and 2.5 GB). Future work includes reducing the construction space, even at some increase in construction time. We also aim to reduce the space of $T'$, the most space-demanding component of our index. Preliminary experiments show that the topology of $T'$ is highly repetitive, and that it can be about halved with a grammar-compressed representation [25].
References

1. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

2. Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prijibelski, et al. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

3. Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. 46th Annual Symposium on the Theory of Computing (STOC)*, pages 148–193, 2014.

4. 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, 2015.

5. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012.

6. Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525, 2016.

7. Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):201–208, 2016.

8. David Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

9. Nicolaas Govert De Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

10. Gennady Denisov, Brian Walenz, Aaron L Halpern, Jason Miller, Nelson Axelrod, Samuel Levy, and Granger Sutton. Consensus generation and variant detection by Celera Assembler. *Bioinformatics*, 24(8):1035–1040, 2008.

11. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

12. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

13. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

14. Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226, 2012.

15. Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear Work Suffix Array Construction. *Journal of the ACM*, 53(6):918–936, 2006.

16. Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

17. Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

18. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

19. Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah 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.
Simulating the DNA string graph in succinct space

20 Heng Li. wgsim - Read simulator for next generation sequencing. *Bioinformatics*, 28:593–594, 2012.

21 V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

22 Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, 2005.

23 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

24 Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

25 Gonzalo Navarro and Alberto Ordóñez. Faster Compressed Suffix Trees for Repetitive Collections. *ACM Journal of Experimental Algorithmics*, 21(1):article 1.8, 2016.

26 Gonzalo Navarro and Kunihiko Sadakane. Fully-Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

27 Daisuke Okanohara and Kunihiko Sadakane. A Linear-Time Burrows-Wheeler Transform using Induced Sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009.

28 Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA-A Practical Iterative de Bruijn Graph De Novo Assembler. In *Proc. 14th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 426–440, 2010.

29 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

30 Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):367–373, 2010.

31 Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

32 Daniel Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(3):821–829, 2008.

33 Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.
A) **Figures**

**Figure A.1** Example of the computation of the forward and backward overlaps for sequence $R_3 = R[3]$ of the example of Figure A. A) Sequence $R_2$ is a forward overlap for $R_3$ and sequences $R_4$ and $R_5$ are backward overlaps. B) The upper matrix represents the range in $M$ that includes the solid node that represents $R_3$ (row in bold), and its contained nodes (grayed row) and the lower matrix is the range of $M$ that includes the solid node of the reverse complement of $R_3$, $R_3^r$ (row in bold and italic), and its contained nodes (gray rows in italic). Gray symbols to the right of every $R_i$ are backward overlaps. B) The upper matrix represents the range in $M$ that includes the solid node that represents $R_3$ (row in bold), and its contained nodes (grayed row) and the lower matrix is the range of $M$ that includes the solid node of the reverse complement of $R_3$, $R_3^r$ (row in bold and italic), and its contained nodes (gray rows in italic). Gray symbols to the right of every $R_i$ are backward overlaps.

B) ![Diagram of overlap computation](image)

**Figure A.2** Different cases in which the weights of the irreductible overlaps can detect patterns in the data. A) Sequence $R_1$ (DBG node $v$) has only one irreductible overlap, with $R_5$ (DBG node $v''$), and there are 3 transitive overlaps between them ($R_2$, $R_3$ and $R_4$ in gray). In this case, there is enough evidence (transitive overlaps) to infer that the string formed by the union of $R_1$ and $R_5$ exists in the input DNA. B) sequence $R_1$ has 2 irreductible overlaps, with $R_4$ and $R_6$ (DBG nodes $v'$ and $v''$, respectively), but the number of unique transitive overlaps between $R_1$ and $R_4$ is zero ($weight(v \oplus v') = 0$), so the most probable option is that $R_4$ contains a sequencing error (italic underlined symbol). C) $R_1$ has two irreductible overlaps, with $R_5$ and $R_7$, and both overlaps have a weight of 2. In this circumstance, it is more probable that $R_1$ belongs to a repeated region or it is next to a genetic variation, if the reads $R_5$ and $R_7$ come from different individuals.
Simulating the DNA string graph in succinct space

**Figure A.3** Computation of weighted forward overlaps as described in Section 3. A) Sequence $R_1$, represented by dBG node $v$ (bold row in $M$), and its irreductible overlaps, $R_2$ and $R_3$ (asterisks), represented by dBG nodes $v'$ and $v''$ respectively. The transitive overlaps that weight every irreductible overlap are shown with arrows to the right of $M$. The dashed box in the upper left corner of $M$ is $\text{lab} = \text{label}(L_v[|L_v|])$. The left side of B) is the range $[i..j]$ in $M$ resulting from searching the range of $(k-1)$-length strings suffixed by $\text{lab}^\text{rc}$. The right side of B) is the subtree in $T'$ induced by the dBG nodes in $[i..j]$. Dashed leaves in the subtree are those that are (probably) irreductible overlaps of $v$ that was originally added to $Y$ but then discarded because none of the branches starting in its outgoing edges spell the reverse complement of some $l \in L_v$. Gray leaves are the ones that contribute to the weight of $y'$. The element $y'$ and its weighting nodes are also represented in the range $[i..j]$ as gray rows. Subranges of $[i..j]$ whose outgoing branches lead to weighted irreductible overlaps of $v$ are shown in curly brackets.

**Figure A.4** Statistics about the construction of rBOSS. In all the plots, the $x$-axis represents the different values used for $k$, and every shape represents a particular value of $m$. The $y$-axis in A) is the mean elapsed time; in B) it is the memory peak achieved during the construction.
**Figure A.5** Stacked barplot with the percentage that each substructure uses in \( rBOSS \). The x-axis shows the value of \( k \) and the numbers on top of the plot are the \( m \) values.

**B Tables**

| \( k \) | \( m \) | dBG nodes | solid nodes | linker nodes | edges | tree nodes | tree int nodes |
|---|---|---|---|---|---|---|---|
| 50 | 20 | 48.86 | 9.13 | 39.73 | 50.92 | 78.78 | 29.92 |
| 50 | 30 | 48.86 | 9.13 | 39.73 | 50.92 | 68.47 | 19.61 |
| 50 | 40 | 48.86 | 9.13 | 39.73 | 50.92 | 58.15 | 9.29 |
| 70 | 20 | 69.52 | 9.14 | 60.38 | 71.58 | 120.09 | 50.57 |
| 70 | 30 | 69.52 | 9.14 | 60.38 | 71.58 | 109.78 | 40.26 |
| 70 | 40 | 69.52 | 9.14 | 60.38 | 71.58 | 99.46 | 29.94 |
| 90 | 20 | 90.18 | 9.14 | 81.04 | 92.24 | 161.40 | 71.22 |
| 90 | 30 | 90.18 | 9.14 | 81.04 | 92.24 | 151.08 | 60.91 |
| 90 | 40 | 90.18 | 9.14 | 81.04 | 92.24 | 140.76 | 50.59 |
| 110 | 20 | 110.79 | 9.09 | 101.69 | 112.84 | 202.61 | 91.82 |
| 110 | 30 | 110.79 | 9.09 | 101.69 | 112.84 | 192.30 | 81.51 |
| 110 | 40 | 110.79 | 9.09 | 101.69 | 112.84 | 181.98 | 71.19 |

**Table A1** Statistics about the different instances of the dBG graphs generated in the experiments. Except for the first and second column, all the values are expressed in millions. Columns one and two are the values used for \( k \) and \( m \), respectively, in the \( rBOSS \) index. Column three contains the total number of dBG nodes at order \( k \). Column four show the number of solid nodes and column five the number of linker nodes. Column six is the total number of edges in the dBG (number of symbols in \( E \)). Column seven is the total number of nodes in the overlap tree and column eight is the number of internal nodes in the overlap tree.

**C Pseudocodes**

**D Building \( rBOSS \)**

To build our data structure, we first form the string \( R = R_1 \$ R_1^{rev} \ldots R_n \$ R_n^{rev} \# \) over the alphabet \( \Sigma \cup \{\$, \#\} \), with size \( n’ = |R| \), and that represents the concatenation of the reads in \( R^* \). In \( R \), symbol \# is the least in lexicographical order. Next, we build the \( SA \), \( BWT \) and \( LCP \)
Simulating the DNA string graph in succinct space

Figure A.6 Other results of the assembly experiments. In all the figures, the x-axis are the values for $k$. In figures A), B), and C), the shapes are the values of $m$. A) Maximum length achieved for a contig in each $r$BOSS index. B) Mean size for contigs in every $r$BOSS index. C) Difference between the memory peak and the size of the index, that is, the memory used exclusively for assembly. D) Comparison of the longest contig in $r$BOSS (gray line) versus the longest unitig of a fixed dBG (black line) using the same $k$, both with $m = 50$.

Algorithm 1 Build set $L$ with the linker nodes contained by $v$

1: procedure nextcontained($v, m$) \[\triangleright v\text{ is a vo-dBG node and } m\text{ is the minimum suffix size}\]
2: $d \leftarrow v.order - 1$
3: while $d \geq m$ do
4: $u \leftarrow$ shorter($v, d$)
5: if $u > v$ then
6: if isLinker($v^j$) and $|\text{label}(v^j)| = d$ then
7: return $u$
8: $v \leftarrow u$
9: $d \leftarrow d - 1$
10: return 0 \[\triangleright\text{ dummy node}\]

11: procedure buildC($v, m$)
12: $L \leftarrow \emptyset$
13: $c \leftarrow$ nextcontained($v, m$)
14: while $c > 0$ do
15: $L \leftarrow L \cup \{c[1]\}$
16: $c \leftarrow$ nextcontained($c, m$)
17: return $L$

arrays for $\overline{R}$, the reversal of $R$. We use $\overline{R}$ instead of $R$ because the BWT ($E$ in $BOSS$) contains the symbols to the left of every suffix (node labels in $BOSS$), but we actually need the symbol to the right when we call forward. After building these arrays, we modify $LCP$ to simulate the padding of the dummy symbols: for every $LCP[i]$, we compute the distance $d$ between $SA[i]$ and the position in $\overline{R}$ of the next occurrence of symbol $\$$. If
Index size comparison between rBOSS and VO-BOSS. The x-axis gives the values of k and the y-axis is the size of the index in MB. All the rBOSS indexes were built with $m = 30$.

**Algorithm 2** Function nextcontained implemented with the topology of $T'$

1: procedure nextcontained($v$)  \(\triangleright v\) is a vo-dBG node at order $K - 1$
2: \(t \leftarrow \text{leafselect}(T', v)\)  \(\triangleright t\) is a node in $T'$ mapping $v$
3: \(t' \leftarrow \text{parent}(T', t)\)
4: \(\text{if } \text{firstchild}(T', t') = t \text{ then} \)  \(\triangleright t\) is already the leftmost sibling
5: \(t' \leftarrow \text{parent}(T', t')\)
6: \(l \leftarrow \text{lchild}(T', t')\)
7: return \(\text{leafrank}(T', l)\)  \(\triangleright\) vo-dBG node mapping $l$

$d < k - 1$ and $d \leq \text{LCP}[i]$, then we set $\text{LCP}[i] = k - 1$. To compute $d$ in constant time, we can generate a bitmap $D$, with rank and select support (see Section D.1), that marks in $R$ the position of every $. Thus, $d$ is computed as $\text{select}_1(D, \text{rank}_1(D, \text{SA}[i] + 1) - \text{SA}[i])$.

The next step is to build $E$. To this end, we traverse $\text{BWT}[i]$ for increasing $i$ as long as $\text{LCP}[i] \geq k - 1$, and in the process we mark in a bitmap $S[1..\sigma]$ the symbols seen so far. When $\text{LCP}[i] < k - 1$, we append the marked symbols of $S$ to $E$, and also append the same number of bits to $B$, all zeros except the last in each step. We then reset $S$ and restart the traversal of $\text{BWT}$.

The final step is to build $T'$. We use an algorithm [3] to build the topology in $BP$ of a tree, modified to discard on the fly the unnecessary nodes. We first compute the virtual suffix tree $\text{ST}$ from $\text{LCP}$[1], and then traverse it in preorder. For each node $v$, we write an opening parenthesis if it satisfies the restrictions, then we recursively traverse its children, and finally write a closing parenthesis if $v$ satisfied the conditions.

The $\text{SA}$ for $R$ can be built in linear time [15, 18, 17], and so can $\text{BWT}$, [27] $\text{LCP}$ [16], and the virtual $\text{ST}$ [11]. Our modifications are obviously linear-time, and therefore the rBOSS structure can be built in linear time as well.

**D.1 Rank and select data structures**

*Rank* and *select* dictionaries are fundamental in most succinct data structures. Given a sequence $B[1..n]$ of elements over the alphabet $\Sigma = [1..\sigma]$, $B.\text{rank}_b(i)$ with $i \in [1..n]$ and $b \in \Sigma$, returns the number of times the element $b$ occurs in $B[1..i]$, while $B.\text{select}_b(i)$
returns the position of the $i$th occurrence of $b$ in $B$. For binary alphabets, $B$ can be represented in $n + o(n)$ bits so that `rank` and `select` are solved in constant time \[8\]. When $B$ has $m \ll n$ 1s, a compressed representation using $m \log \frac{n}{m} + O(m) + o(n)$ bits, still solving the operations in constant time, is of interest \[29\]. This space is $o(n)$ if $m = o(n)$.

### E  Genome assembly

In this section we briefly describe how to use rBOSS to assemble a genome. We define some concepts first.

> **Lemma 7.** A solid node $v$ is right-extensible (RE) (respectively left-extensible (LE)) if (i) it is a non-$s$-node with outdegree 1 (respectively, a non-$p$-node with indegree 1) or (ii) it is an $s$-node with outdegree $\leq 1$ (respectively $p$-node with indegree $\leq 1$) and following its outgoing edge (if outdegree is 1) and the outgoing edges of every $l \in L_v$ (respectively its incoming edge, if indegree is 1, and the incoming edges of its backward overlaps) leads to a unique solid node $v'$ in at most $(k - 1) - m$ forward operations.

> **Theorem 8.** Computing if a solid node $v$ is RE has $O(k + |L_v| (k - m) \log \sigma)$ worst case time complexity. Computing if $v$ is LE also has $O(k + |L_v| (k - m) \log \sigma)$ time complexity if we augment rBOSS with $s \log s$ bits, where $s$ is the number of solid nodes.

**Proof.** When $v$ is not an $s$-node, testing if it is RE reduces to checking its outdegree. The other case, when $v$ is a $s$-node, is harder. First compute $L_v = \text{buildL}(v)$, and then perform a set of operations in batches over the elements of $L_v$, as follows. Regard $L_v$ as a queue. If $L_v[1]$, the linker node that represents the greatest suffix of $v$, has outdegree 0, then remove it from $L_v$. After that, check that each $L_v[i]$, with $i \in [1..|L_v|]$, has outdegree 1 and that all the outgoing edges are labeled with the same symbol. If some $L_v[i]$ has outdegree $> 1$ or two or more different symbols are seen in the outgoing edges of $L_v$, then return false. If all outgoing edges in $L_v$ have the same symbol $a$, then perform $L_v[i] = \text{forward}(L_v[i], a)$ for every $i$. Repeat the process until $L_v$ becomes empty, if that happens, then return true.

Computing LE is exactly the same process, but first we have to compute $v^c$, the reverse complement of $v$. If we use $s \log s$ extra bits to store the permutation with the reverse complements of the solid nodes, then we can compute $v^c$ in $O(1)$.

We also define the concept of right and left maximal paths.

> **Lemma 9.** A right-maximal path (respectively, left-maximal) over rBOSS is a path where all the nodes are RE (respectively, LE) except the rightmost (respectively, leftmost) node. A maximal-path is the concatenation of a left-maximal and a right-maximal path.

### E.1 Marking non-extensible nodes

Computing whether a solid node $v$ is extensible during a graph traversal can be expensive (Lemma \[5\]), especially if the traversal is exhaustive. The amount of computation can be reduced, however, by computing beforehand which nodes are non-extensible and marking them in a bitmap $N$ of size $s$. Notice that only a small fraction of the nodes will be non-extensible, so $N$ is highly compressible.

There are four cases in which $v$ is non-extensible; (i) it has outdegree $> 1$, (ii) there are two or more different outgoing symbols in $L_v$, (iii) the outgoing symbol in $v$ differs from the symbol in $L_v$, or (iv) the computation of the forward overlaps of $v$ yields two or more
different irreductible overlaps (figure x depicts all these cases). To detect non-extensible nodes, we use the topology of $T'$ instead of directly calling the function $foverlaps$.

We descend on $T'$ in DFS, and every time we reach a leaf $t$ that is the leftmost child of its parent, we append its edge symbols into a sequence $U$ and the leaf rank of $t$ to an array $I$, one append per edge. We also keep track of the different symbols appended into $U$ so far. If after consuming $t$ there are two or more different symbols in $U$, we scan $U$ from right to left until finding the first position $i$ such that $U[i] \neq U[i+1]$. Then, we mark as non-extensible all the solid nodes that contain any prefix of $l\text{label}(I[i + 1])$ that in turns contains any prefix of $l\text{label}(I[i])$ of length $\geq m$. The rationale is that any solid node $v$ containing $I[i + 1]$ will also contain $I[i]$ (we now this fact because the DFS order). The problem, however, is that the outgoing symbols of $I[i]$ and $I[i + 1]$ differ. Thus, $v$ is a case (ii) non-extensible node. It can still happen that one of the prefixes of $I[i + 1]$ is contained by a solid node $v'$, and if it does, then it might happen that $v'$ also contains a prefix of $I[i]$. In such case, $v'$ is a case (iv) non-extensible node, because the elements of $L_v'$ will lead to $I[i]$ and $I[i + 1]$, which are known to differ in their outgoing edges. To be sure, we must go backwards in $I[i + 1]$ marking the solid nodes that contain prefixes of $I[i + 1]$, and we stop when $I[i + 1]$ does not contain any prefix of $I[i]$ of size $\geq m$.

When a solid node $v$ is reached during the DFS traversal, we first have to check if it was already marked. If it is still unmarked, then we check if it has outdegree more than two ($v$ is a case i), or if it has outdegree 1, but its outgoing symbol differ from the outgoing symbols in $L_v$ ($v$ is case iii).

E.2 Spelling maximal paths

Once rBOSS and the bitmap $N$ are built, the process of spelling maximal paths can be implemented as an stream algorithm, which is very space-efficient. For every non-extensible node $v$ compute the set $O_v = \text{overlaps}(v)$. We start a forward traversal from each $v, o \in O_v$ and continue until reaching a non-extensible node. During the process, append the edge symbols to a vector $F$. After finishing, compute $o_{rc}^c$, start a forward traversal from it and continue until reaching the next non-extensible node. As with $o$, also append the outgoing edges to a vector $R$. The final string spelled by the maximal path will be $R_{rc} \cdot \text{label}(o) \cdot F$. If in either of both traversals, forward or backward, an extensible solid node with outdegree 0 is reached, then call $\text{nextcontained}$ and continue trough its edges.