Genome assembly with variable order de Bruijn graphs*

Diego Díaz-Domínguez[1], Taku Onodera[2], Simon J. Puglisi[1], and Leena Salmela[1]

[1] Department of Computer Science, University of Helsinki, Finland
{diego.diaz,simon.puglisi,leena.salmela}@helsinki.fi

[2] Department of Computer Science, University of Tokyo, Japan
tk-ono@is.s.u-tokyo.ac.jp

Abstract. The nodes of a de Bruijn graph (DBG) of order k correspond to the set of k-mers occurring in a set of reads and an edge is added between two nodes if there is a $k-1$ length overlap between them. When using a DBG for genome assembly, the choice of k is a delicate issue: if k is too small, the DBG is tangled, making graph traversal ambiguous, whereas choosing k too large makes the DBG disconnected, resulting in more and shorter contigs. The variable order de Bruijn graph (voDBG) has been proposed as a way to avoid fixing a single value of k. A voDBG represents DBGs of all orders in a single data structure and (conceptually) adds edges between the DBGs of different orders to allow increasing and decreasing the order. Whereas for a fixed order DBG unitigs are well defined, no properly defined notion of contig or unitig exists for voDBGs. In this paper we give the first rigorous definition of contigs for voDBGs. We show that voDBG nodes, whose frequency in the input read set is in interval $[\ell, h]$ for some h and $\ell > h/2$, represent an unambiguous set of linear sequences, which we call the set of (ℓ, h)-tigs. By establishing connections between the voDBG and the suffix trie of the input reads, we give an efficient algorithm for enumerating (ℓ, h)-tigs in a voDBG using compressed suffix trees. Our experiments on real and simulated HiFi data show a prototype implementation of our approach has a better or comparable contiguity and accuracy as compared to other DBG based assemblers.

Keywords: genome assembly, de Bruijn graph, variable order de Bruijn graph, suffix trie, suffix tree

* This work was funded by Academy of Finland via grants 308030 and 323233 (to LS) and 339070 (to SJP).

1 Introduction

De Bruijn graph (DBG) based genome assemblers [14,22] first reduce the input reads to a set of k-mers and then produce the assembled genome by reporting paths in the DBG built on these k-mers. While this approach has proved effective and efficient, the order of the DBG, i.e. the value of k, can greatly affect the quality of the assembly. Deciding on a good value of k is not a straightforward task [7]. A small value of k tends to lead to a tangled graph, while a large value of k can lead to a fragmented graph. Furthermore, a different value of k is often optimal for different parts of the genome as the coverage and the repeat content may vary. To counter this, many DBG based assemblers (see e.g. IDBA [21] and SPAdes [2]) use an iterative approach where the value of k is increased in each iteration. However, such an iterative approach increases the runtime of the methods considerably because in each iteration the DBG needs to be rebuilt for the new value of k.

Manifold de Bruijn graphs [17] and variable order de Bruijn graphs (voDBGs) [4] were introduced in an attempt to avoid the need to settle on a single k-mer length at any stage. Manifold de Bruijn graphs generalize DBGs by allowing substrings of arbitrary length as nodes of the graph instead of fixed length k-mers. The manifold de Bruijn graph has not yet been implemented and so remains mostly of theoretical interest. voDBGs [4] represent DBGs of all orders in a single data structure and Boucher et al. [4] provide an implementation by extending the BOSS data structure [5] to represent DBGs of all orders up to a maximum order K. However, while clear definitions of contigs, such as unitigs and omnitigs [27], exist for regular DBGs, no definition of contigs has been presented for voDBG, and consequently, no genome assemblers have been developed based on them.

DNA sequencing instruments have continued to evolve towards producing longer and more accurate reads. Currently, the Pacific Biosciences HiFi reads are 10 kbp to 20 kbp long with an error rate less than 1% [28]. Due to their length and accuracy, the HiFi reads have been central to a range of recent assembly methods. The first HiFi read assemblers, Peregrine [8], HiCanu [20], and hifiasm [6], are based on the overlap-layout-consensus approach and string graphs [19]. More recently, HiFi read assemblers based on DBGs have emerged. A central challenge for these methods has been to efficiently construct and represent DBGs with a very large value of k which is necessary to take advantage of the length of the HiFi reads. MBG [24] solves this challenge by resorting to a sparse DBG which only includes a subset of the k-mers, while (its near namesake) mdBG [10] converts the reads to sequences of minimizers and performs DBG based assembly on the minimizer sequences. LJA [1] constructs a regular DBG but avoids the memory bottleneck of a DBG with large k by directly constructing the compacted DBG and by utilizing disjointigs and Bloom filters in the construction. Because of the need for a DBG of large order, the range of possible values of k is much larger than in short read assembly where the read length limits the value of k. Thus the demand for using several different values of k in different regions of the genome to achieve a contiguous and accurate assembly becomes urgent. To this end, Bankevich et al. [1] introduce the multiplex DBG in LJA. The multiplex DBG is constructed by iteratively increasing the value of k in those parts of the graph where it is deemed possible according to reads threaded through the graph.

We present the first definition of contigs for a voDBG. We consider a frequency restricted subgraph of the voDBG and show that when the frequencies of nodes are limited to an interval $[\ell, h]$, where $\ell > h/2$, the subgraph corresponds to a set of strings that we call (ℓ, h)-tigs. We analyse how the choice of ℓ and h affects the contiguity and accuracy of the assembly and give a method for choosing appropriate values of ℓ and h. Furthermore, we establish connections between the voDBG and the generalized suffix trie of the read set leading to an efficient algorithm for enumerating (ℓ, h)-tigs using a succinct representation of the generalized suffix tree of the reads. We implement our algorithm in the first assembler based on voDBGs and show that on real and simulated HiFi data our approach has a better or comparable contiguity and accuracy as compared to other DBG based assembly approaches but the best overlap based assemblers still produce more contiguous assemblies by tolerating the few remaining errors better.

2 Suffix tries, suffix trees, and variable order de Bruijn graphs

Let the string S be a sequence of characters $S[1]S[2]\dots S[n]$ where each character $S[i]$ is drawn from an alphabet Σ. A string S' is a suffix of S if $S' = S[i]S[i+1]\dots S[n]$ for some $1 \leq i \leq n+1$ and a string S' is a

prefix of S if $S' = S[1]S[2]\dots S[i]$ for some $0 \leq i \leq n$. A string S' is a substring of S if $S' = S[i]S[i+1]\dots S[j]$ for some $0 \leq i \leq j \leq n$. The empty string is a suffix, a prefix and a substring of any string S.

The *suffix trie* of S is the trie of all suffixes of S. The suffix trie therefore has a node for every distinct substring of S. We will refer to the nodes of the suffix trie by the substrings they represent (i.e., their path labels). The nodes corresponding to two patterns S_1 and S_2 occuring in S, respectively, are connected with an edge (S_1, S_2) labeled with character a if $S_2 = S_1 a$.

The *suffix tree* is a compact representation of the suffix trie. The nodes of a suffix tree of S are the branching nodes of a suffix trie of S. The string labeling a path of nonbranching nodes between two branching nodes in the suffix trie becomes the label of the edge between those two nodes in the suffix tree. By storing edge labels as a pair of pointers into S the suffix tree takes linear space.

The *suffix link* of a suffix tree node $S_1 = aS_2$ is an edge pointing to the node S_2. On the other hand, a Weiner link labeled $a \in \Sigma$ for S_1's node is an edge pointing to the node $S_3 = aS_1$. Notice that a suffix tree node can have up to $|\Sigma|$ Weiner links but only one suffix link. Additionally, the Weiner link going to aS_1 is said to be implicit if aS_1 matches only the prefix of a node's label. The generalized suffix tree of a string collection R is the compact trie of all suffixes of all strings in R. We refer the reader to Mäkinen et al. [18] for more information about suffix trees and their compressed representations.

Let us describe some common queries over the suffix tree that will be useful in the following sections:

- $\mathsf{label}(v)$: string labeling the path from the suffix tree's root to the node v
- $\mathsf{size}(v)$: number of leaves under the subtree rooted at v
- $\mathsf{slink}(v)$: node u reached by following the suffix link of v

We next give a definition of the variable order de Bruijn graph [4].

Definition 1 (Variable order de Bruijn graph (voDBG)). *Given a read set R, a variable-order de Bruijn graph has a node for each distinct k-mer S of any length k occurring in R. We add three kinds of edges:*

DBG edges *Given two k-mers S_1 and S_2 of the same length k, we add an edge (S_1, S_2) if the $k-1$ length suffix of S_1 equals the $k-1$ length prefix of S_2. The edge is labelled with $S_2[k]$.*

Extension edges *Given a k-mer S_1 of length k and a $(k+1)$-mer S_2 of length $k+1$, we add an edge (S_1, S_2) if S_1 is the k length prefix of S_2. The edge is labelled with $S_2[k+1]$.*

Contraction edges *Given a k-mer S_1 of length k and a $(k-1)$-mer S_2 of length $k-1$, we add an edge (S_1, S_2) if the $k-1$ length suffix of S_1 is S_2. Contraction edges do not have labels.*

The original definition of voDBGs by Boucher et al. [4] had an upper bound on the length of k-mers, i.e. the nodes of the graph would be the distinct k-mers occurring in R for any $k < K$ for some K. In this paper, we assume that there is no upper bound on the length of k-mers. Such voDBGs have previously also been called infinite order de Bruijn graphs [3].

A voDBG is called *frequency aware* if the graph stores (or is otherwise able to determine) the frequency $f(S)$ of the k-mer S in R for each node S.

We note that the definition of voDBG resembles a generalized suffix trie. Both the nodes of the generalized suffix trie and the nodes of the voDBG correspond to all substrings occurring in the read set. Thus the nodes of voDBG correspond one-to-one to the generalized suffix trie nodes, extension edges correspond one-to-one to the trie edges and contraction edges correspond one-to-one to the suffix links. Furthermore, we see that the DBG edges are "transitive": they correspond to a contraction edge followed by an extension edge. The suffix trie therefore, and its compact representation, the suffix tree, can be used to implement a frequency aware voDBG.

We have defined a DBG using the node centric definition of a DBG where the k-mers are the nodes and an edge is added for all $k-1$ length overlaps between the k-mers. An alternative definition of a DBG defines the edges as the k-mers occurring in the read set and the nodes are then the $k-1$ length prefixes and suffixes of the k-mers. We saw above that for the node centric definition of a DBG, the DBG edges of voDBG correspond to transitive edges composed of a contraction edge followed by an extension edge. Similarly, for the edge centric definition of DBGs, the DBG edges of voDBG correspond to transitive edges composed of an extension edge followed by a contraction edge. From now on, we will only use the node centric definition of a DBG.

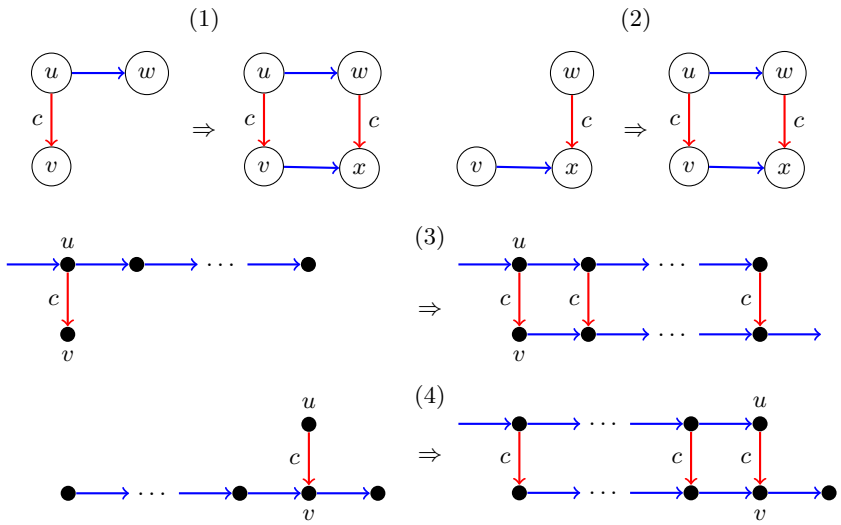

Fig. 1. Lemma 2 and 4. (1) (resp. (2)) is the figurative interpretation of the first (resp. second) part of Lemma 2. (3) (resp. (4)) shows how the part $r_v \geq r_u$ (resp. $l_u \geq l_v$) of Lemma 4 can be derived by successive application of (1) (resp. (2)). Red (resp. blue) edges are extension (resp. contraction) edges.

3 Frequency restricted variable-order de Bruijn graphs

Definition 2 (Frequency restricted variable order de Bruijn graph). *Given a read set R and the frequency interval $[\ell, h]$, a frequency restricted variable order de Bruijn graph $G_{\ell,h}$ has a node for each k-mer S that occurs at least ℓ and at most h times in R. The edges are added as in a variable order de Bruijn graph if both k-mers involved meet the frequency constraints.*

Next we will consider the special case of a frequency restricted variable order de Bruijn graph where $h/2 < \ell < h$. Then the following lemmas hold.

Lemma 1. *In a frequency restricted voDBG where $\ell > h/2$, each node has at most one outgoing and at most one incoming extension edge. Similarly, each node has at most one outgoing and at most one incoming contraction edge.*

Proof. Suppose a $(k+1)$-mer S of length $k+1$ has two incoming extension edges (S_1, S) and (S_2, S). By definition both S_1 and S_2 are k length prefixes of S and thus they must be the same string and are represented by the same node. Therefore S cannot have more than one incoming extension edge.

Suppose that a k-mer S of length k has two outgoing extension edges (S, S_1) and (S, S_2). Thus S is the k length prefix of both S_1 and S_2 and the last characters of S_1 and S_2 must be different. Since S_1 and S_2 are nodes of the frequency restricted variable order de Bruijn graph, they must both have at least ℓ occurrences in R and these must be distinct occurrences since the last characters of S_1 and S_2 differ. Thus the k length prefix of S_1 and S_2, namely S, must have at least $2\ell > h$ occurrences and therefore S is not a node in the frequency restricted variable order de Bruijn graph, contradicting our assumption of the existence of extension edges (S, S_1) and (S, S_2).

The proof of the second part of the lemma, i.e., each node has at most one outgoing and at most one incoming contraction edge, follows a similar argument as above. □

Lemma 2. *If there is an extension edge (u, v) with label c and contraction edge (u, w), there is a node x in $G_{\ell,h}$ s.t. extension edge (v, x) with label c and contraction edge (w, x) exist. Similarly, if there is an extension edge (w, x) with label c and contraction edge (v, x), there is a node u in $G_{\ell,h}$ s.t. extension edge (u, w) with label c and contraction edge (u, w) exist. See also Figure 1.*

Proof. The first part follows from $\ell \leq f(v) \leq f(x) \leq f(w) \leq h$. The second part follows from $\ell \leq f(v) \leq f(u) \leq f(w) \leq h$. □

For nodes u, v of $G_{\ell,h}$, let $u \sim v$ denote the relation that there is a contraction path from u to v or v to u.

Lemma 3. *The relation $\sim$ is an equivalence relation.*

Proof. The symmetry and reflexivity are trivial. The transivity follows from the fact that each node has at most one incoming/outgoing contraction edge (Lemma 1). □

The relation $\sim$ partitions $G_{\ell,h}$ into paths of nodes connected by contraction edges. We call them *contraction paths.*

Lemma 4. *Let $u_{-l_u}, u_{-l_u+1}, \ldots, u_0 = u, u_1, \ldots, u_{r_u}$ and $v_{-l_v}, v_{-l_v+1}, \ldots, v_0 = v, v_1, \ldots, v_{r_v}$ be two distinct contraction paths. If there is an extension edge (u, v) with label c, then $l_u \geq l_v$, $r_v \geq r_u$ and there are extension edges (u_i, v_i) with label c for every $l_v \leq i \leq r_u$.*

Proof. Follows from the successive applications of Lemma 2. See also Figure 1. □

If the condition (and thus, the conclusion) of Lemma 4 is satisfied, we abuse terminology and say there is an extension edge with label c from the former contraction path to the latter. Note that now we are considering the meta-graph where each node is a contraction path in $G_{\ell,h}$.

By Lemma 1 and Lemma 4, a contraction path has at most one outgoing/incoming extension edge and thus, the meta-graph of contraction paths can be partitioned into chains and cycles. (For meta-graph, we use the term "chain" instead of "path" to make it easier to distinguish which graph we are discussing.) The concatenation of the extension edge labels of each such chain/cycle spells out a unique string, with the proviso that, for a cycle, there is a degree of freedom in the head position. We call these *partial (ℓ, h)-tigs.*

Now we give a procedure to spell out partial (ℓ, h)-tigs on $G_{\ell,h}$ instead of the meta-graph of contraction paths. In this procedure, we traverse $G_{\ell,h}$ node-by-node. If a node in a contraction path is visited, we say the path is visited. The procedure is as follows: i) for each node without incoming edge, start from the node and continue following edges while there is an edge to follow; ii) while there is a node that does not belong to any visited contraction path, choose one as the start and continue following edges while there is an edge to a node in an unvisited contraction path. Part i) spells out non-cyclic partial (ℓ, h)-tigs and part ii) spells out cyclic partial (ℓ, h)-tigs. This procedure is not completely deterministic because a node can have two outgoing edges (extension and contraction). Nevertheless, partial (ℓ, h)-tigs are spelled out uniquely because, by Lemma 4, it does not matter where exactly in a contraction path we transition to the next contraction path. For case i), we call the string label of the starting node concatenated by the spelled out partial (ℓ, h)-tig the *(ℓ, h)-tig*; for case ii), the (ℓ, h)-tig is defined to be the same cyclic string as the partial (ℓ, h)-tig.

We can make the traversal rule deterministic by deciding to follow the extension edge if a node has both extension and contraction edge. We call the path in $G_{\ell,h}$ traversed by this particular version of the procedure *canonical.* Figure 2 illustrates the voDBG and one chain of contractions paths.

The above exposition partitions the frequency restricted voDBG $G_{\ell,h}$ by contraction paths. We could equally partition it by *extension paths*, i.e. paths consisting solely of extension edges. This results in a similar meta-graph, where the extension paths form chains and cycles and each chain or cycle spells out one (ℓ, h)-tig. However, because the edge labels are hidden inside the extension paths in this partitioning, spelling the (ℓ, h)-tigs is not as explicit as when $G_{\ell,h}$ is partitioned by contraction edges. In Section 5, we develop an algorithm based on the suffix tree for enumerating the (ℓ, h)-tigs. This algorithm is based on the partitioning of $G_{\ell,h}$ by extension paths because this is more natural for a suffix tree where the extension paths correspond to paths following suffix tree edges only.

4 Determining ℓ and h

To implement our construction, we need to determine values for ℓ and h. Ultimately, we would like to choose our thresholds so that the (ℓ, h)-tigs are as contiguous as possible but do not contain misassemblies. The (ℓ, h)-tigs can be broken due to two reasons: (i) if a region of the genome has such a low coverage that substrings of that region have a frequency below ℓ or (ii) the coverage suddenly drops or rises by more than $h - \ell + 1$.

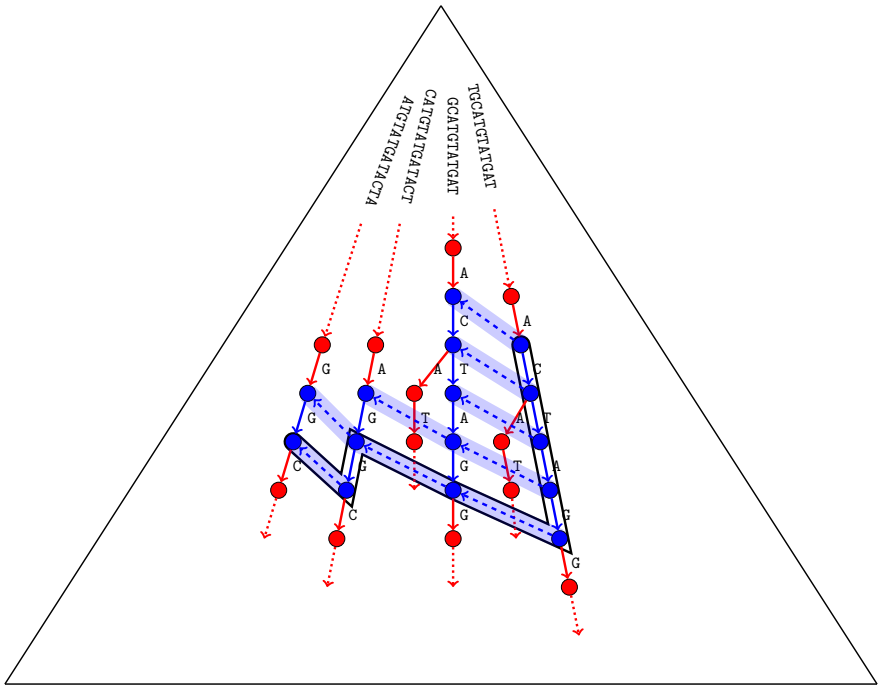

Fig. 2. The voDBG and one chain of contraction paths of the frequency restricted voDBG shown in blue. The organization of the nodes is as in a suffix trie. Extension edges are shown as solid and contraction edges are shown with dashed lines. We omit the DBG edges for clarity. The contraction paths are highlighted in light blue. The blue nodes form a chain of contraction paths and any maximal path through these nodes spells the partial (ℓ, h)-tig, `ACTAGG`. The complete (ℓ, h)-tig is `TGCATGTATGATACTAGG`. The canonical maximal path is outlined in black.

To avoid breaking an (ℓ, h)-tig due to low coverage, ℓ should not be too high. On the other hand, we will soon see that setting ℓ low will make it difficult to avoid misassemblies.

On those parts of the genome where the coverage changes gradually, we can always find k-mers of some length that have extension and contraction edges. However, if the coverage changes suddenly, this may not be the case and thus an (ℓ, h)-tig can be broken due to a large increase or decrease in coverage from one genomic position to the next. Consider a position on the genome where at least $h - \ell + 1$ reads start and further a k-mer S starting at that position with $\ell \leq f(S) \leq h$. The frequency restricted voDBG then cannot have a contraction edge from cS to S for any character c because $f(cS) \leq f(S) - (h - \ell + 1) \leq \ell - 1$ and thus cS is not a node of the frequency restricted voDBG. Similarly consider a k-mer S with $\ell \leq f(S) \leq h$ ending at a position on the genome where at least $h - \ell + 1$ reads end. Now the frequency restricted voDBG cannot have an extension edge from S to Sc for any character c because $f(Sc) \leq f(S) - (h - \ell + 1) \leq \ell - 1$. Thus $h - \ell + 1$ must be chosen so that the probability of $h - \ell + 1$ reads starting or ending at the same position is low. This corresponds to the m balls to n bins problem where the bins correspond to the positions in the genome and the balls correspond to the starting point of reads. According to Raab and Steger [23] if we choose $h - \ell + 1$ as follows:

$$h - \ell + 1 \geq 2 \frac{\log L}{\log \frac{L \log L}{|R|}} \tag{1}$$

where the logarithms are natural logarithms, L is the genome length and $|R|$ is the number of reads then with high probability no position of the genome is a starting point or an ending point of more than $h - \ell + 1$ reads. The above assumes error free reads. Each error in the reads, contributes to a drop and rise in coverage in a similar way as a start and an end of a read. Thus to take into account sequencing errors, we replace $|R|$ in Equation 1 by $|R| + e \cdot C \cdot L$, where e is the error rate and C is the coverage of the read set:

$$h - \ell + 1 \geq 2 \frac{\log L}{\log \frac{L \log L}{|R| + e \cdot C \cdot L}} \tag{2}$$

Let us consider a simple repeat scenario to analyse the probability of misassembling an (ℓ, h)-tig. Let S be a repeat in the genome such that $\ell \leq f(S) \leq h$ and there are two possible extensions of S in the genome: Sc_1 and Sc_2. The string S does not have enough information for us to know which one of the extensions is correct so our construction should not accept either one of them. Thus we want to analyse the probability that $f(Sc_1) \geq \ell$ or $f(Sc_2) \geq \ell$. Note that both of these conditions cannot hold when $\ell > h/2$. Assuming that reads are sampled uniformly at random, i.e. both extensions are equally likely this is

$$P = 2\sum_{i=\ell}^{h} \binom{h}{i} 0.5^i 0.5^{h-i} = 0.5^{h-1} \sum_{i=\ell}^{h} \binom{h}{i}.$$

Using the Chernoff bound, P is bounded above by $2e^{-2\delta^2/h}$ where $\delta = \ell - h/2$. Suppose that we are ready to tolerate a probability ϵ for misassembling a simple repeat, then we must choose ℓ such that

$$\ell \geq \frac{h}{2} + \sqrt{\frac{h}{2} \log\left(\frac{2}{\epsilon}\right)} \tag{3}$$

where the logarithm is the natural logarithm. In other words, ℓ should be a bit larger than $h/2$.

We note that the above analysis of misassemblies is valid only for our simple scenario. For example, it does not apply to a repeat occurring three times in the genome so that twice it is followed by a character c and once by a character $c' \neq c$.

In summary, to avoid breaking the (ℓ, h)-tigs due to drops and rises in coverage and to avoid misassemblies, we combine Equations 2 and 3:

$$\frac{h}{2} + \sqrt{\frac{h}{2} \log\left(\frac{2}{\epsilon}\right)} \leq \ell \leq h - 1 - 2\frac{\log L}{\log \frac{L \log L}{|R| + e \cdot C \cdot L}} \tag{4}$$

Furthermore, ℓ must be an integer and we want ℓ to be small to avoid breaking (ℓ, h)-tigs due to low coverage. Thus we find the smallest h such that Equation 4 is satisfied by an integer ℓ and these are our final values for h and ℓ.

5 Algorithm for extracting (ℓ, h)-tigs

In this section, we will develop an algorithm to enumerate all the (ℓ, h)-tigs for a frequency-restricted voDBG $\mathcal{G}_{\ell,h}$. We will assume that ℓ and h are input parameters and that $\mathcal{G}_{\ell,h}$ was constructed from a string collection R' storing the reads in R along with their reverse complements. The inclusion of the reverse complements is a technical decision as reads are usually strand unspecific.

The general strategy to spell an (ℓ, h)-tig L from $\mathcal{G}_{\ell,h}$ is as follows. First, we select a node v in $\mathcal{G}_{\ell,h}$ with no variable-order incoming nodes and append its associated k-mer sequence into L. Then, we begin a path traversal from v, visiting only extraction and contraction edges. The order in which we switch between contraction and extension edges is arbitrary as it does not affect the final sequence of L (see Lemma 4). However, every time we visit an extension edge, we insert its label $c \in \Sigma$ into L. If during the path traversal, we reach a node u with no variable-order outgoing nodes, we stop the extension of L and mark v (i.e., the first node in the path spelling L) as visited. To get the next (ℓ, h)-tig, we search for an unmarked node with no variable-order incoming nodes and repeat the same procedure. The enumeration of (ℓ, h)-tigs ends when all the nodes with no incoming variable-order edges in $\mathcal{G}_{\ell,h}$ are marked.

We note that the procedure described above will spell for each (ℓ, h)-tig L an equivalent (ℓ, h)-tig $\hat{L}$ whose sequence is the reverse complement of L. In practical Genomics applications, one chooses only one string (L or $\hat{L}$) as both contain the same information. We can avoid spelling $\hat{L}$ with one simple modification: when we reach the last node u in the path for L (i.e., the node with no variable-order outgoing edges), we compute the node $\hat{u}$ labeled with the reverse complement of u's label, and we mark $\hat{u}$ as visited. The inclusion in R' of the reverse complements of R produces a symmetric graph $\mathcal{G}_{\ell,h}$ (see Section 4 of Díaz-Domínguez et al. [9] for further information on this symmetry). This property means that, if u is a node with no variable-order

outgoing edges, then $\hat{u}$ is a node with no variable-order incoming edges. By Lemma 4, we know that the only (ℓ, h)-tig we can spell from $\hat{u}$ is $\hat{L}$. Thus, by marking $\hat{u}$ beforehand, we prevent the construction of $\hat{L}$.

We also note that our algorithm might not spell (ℓ, h)-tigs if the underlying genome for $\mathcal{G}_{\ell,h}$ is circular and the sequencing coverage of R for that genome is always within the range $[\ell, h]$. In this scenario, $\mathcal{G}_{\ell,h}$ will have only one (ℓ, h)-tig, which we can spell by performing a circular walk over the voDBG. Notice that all the nodes in $\mathcal{G}_{\ell,h}$ will have variable-order incoming nodes, meaning that our algorithm will not find any node from which to start the extension of a new (ℓ, h)-tig. Still, the solution to this problem is simple: if our method returns an empty set of strings, we randomly peak a node v from $\mathcal{G}_{\ell,h}$ and mark all the other nodes that are reachable from v via variable-order edges. Subsequently, we perform a new path traversal starting from the farthest node v' we can reach from v by following contraction edges (i.e., the label of v' is the smallest suffix of v's label). We stop the path traversal only when we reach one the previously marked nodes. The way in which we construct the (ℓ, h)-tig L during the traversal remains the same. Notice that, unlike non-circular genomes, the path spelling the (ℓ, h)-tig can be a cycle as the last node we visit can be v'.

6 Implementing our (ℓ, h)-tig algorithm on the suffix tree

We implement our (ℓ, h)-tig algorithm of Section 5 on top of the generalized suffix tree T of R'. Recall from Section 2 that there is a connection between T and the voDBG. We start by describing the nodes in T that map to the nodes in $\mathcal{G}_{\ell,h}$ with no variable-order incoming edges, and the nodes in T that map to the nodes in $\mathcal{G}_{\ell,h}$ with no variable-order outgoing edges. These elements are necessary to enumerate the (ℓ, h)-tigs from T. For simplicity, we will refer to these nodes as left-end nodes and right-end nodes, respectively.

A node v in T maps to a left-end node in $\mathcal{G}_{\ell,h}$ if:

1. $\ell \leq \mathsf{size}(v) \leq h$.
2. $\mathsf{size}(\mathsf{parent}(v)) > h$
3. $\mathsf{size}(v') < \ell$, for each node v' reached from a Weiner link of v.

Equivalently, v maps to a right-end node of $\mathcal{G}_{\ell,h}$ if:

1. $\ell \leq \mathsf{size}(v) \leq h$.
2. $\mathsf{size}(\mathsf{slink}(v)) > h$
3. $\mathsf{size}(v') < \ell$, for each child v' of v

Our algorithm to spell (ℓ, h)-tigs from the generalized suffix tree works as follows. We traverse T in DFS order, and every time we reach an internal node v that maps to a left-end node in $\mathcal{G}_{\ell,h}$, we start the construction of a new (ℓ, h)-tig. For doing so, we first create an empty string L and insert the sequence of $\mathsf{label}(v)$ into it. Then, we descent over the subtree of T rooted at v, always following the child node u whose size is $\ell \leq \mathsf{size}(u) \leq h$. We refer to the decent over the subtree as an *right-expansion step*. Notice that, as we enforce the property $\ell > h/2$, every internal node we visit during the right-expansion step has at most one child that meets the size condition. Additionally, each time we move from a node u' to its child u in the descent, we insert the label of the edge (u', u) into L. When we reach a node u that it is either a leaf in T or all its children have size less than ℓ, we stop the right-expansion step and begin to follow the suffix links $u_1 = \mathsf{slink}(u), u_2 = \mathsf{slink}(u_1), \ldots, u_{i+1} = \mathsf{slink}(u_i)$ until reach a node u_i such that $\ell \leq \mathsf{size}(u_i) \leq h$ and $\mathsf{size}(u_{i+1}) > h$. We refer to the successive visits of suffix links as a *right-contraction step*. Once we reach v_i, we start a new right-expansion step from v_i, and continue looping between expansions and contractions until we reach a node u that maps to a right-end node of $\mathcal{G}_{\ell,h}$. When this situation happens, we return L and search for the suffix tree node $\hat{u}$ labeled with the reverse complement of $\mathsf{label}(u)$ to mark it as visited. As explained in the previous section, the purpose of marking $\hat{u}$ is to prevent the construction of the reverse complement of L. After marking $\bar{v}$, we continue the DFS traversal of T, skipping the subtree of v (i.e., the suffix tree node from which we began the construction of L) and the subtree of any node that was previously marked as visited. Our suffix-tree-based algorithm to enumerate the (ℓ, h)-tigs ends when it completes the DFS traversal of T. If our algorithm does not yield any string (see Section 5 to see why this situation can happen), we pick a node v with $\ell \leq \mathsf{size}(v) \leq h$ at random from T and we use it to start a loop of expansions and contractions, which we stop when we reach a node whose label is suffixed by v's label.

The most common error in PacBio HiFi reads is the misestimation of the homopolymer lengths [28]. We deal with this problem by operating over the run-length compressed reads as in previous work (see e.g. HiCanu [20], MBG [24], and LJA [1]). We divide the encoding of R' in three data structures: a string $R^h[1,n]$ storing the concatenation of the strings in R' separated by a sentinel symbol $ that is lexicographically smaller than any symbol in R'. Besides, R^h replaces each run of a single base by a metasymbol that represents that base. A bit vector $B[1,n]$ encoding in $B[j]=1$ if the value $R^h[j]$ is a metasymbol, and $B[j]=0$ otherwise. We augment B with a data structure of $o(n)$ bits that allows us to answer the operation $\mathsf{rank}_1(B,j)$ in constant time. This query returns the number of 1s up to position j in B. Finally, let n' be the number of metasymbols in R^h. An integer vector $K[1,n']$ storing in $K[j]$ the length of the jth homopolymer in R'. That is, for example a read AAAAGTGGTAAAC becomes $R^h = [\mathtt{A^*GTG^*TA^*C\$}]$, $B=[1,0,0,1,0,1,0]$, $K=[4,2,3]$, where the characters with * are the metasymbols.

We now construct the generalized suffix tree T for R^h and run our algorithm for spelling (ℓ,h)-tigs. The only difference of this version compared to the one that does not compress the homopolymers is in how it inserts the bases to each (ℓ,h)-tig L. Let v be a node in T and let $W=\mathsf{label}(v)$ be its label. We scan W from left to right to append its symbols to L. If $W[j]=c$ is a base, we append c into L. On the other hand, if $W[j]=c^*$ is a metasymbol, we proceed as follows. We visit all the occurrences of $W[j]$ encoded in the leaves of the subtree of T rooted at v to compute an integer list $\mathcal{L}$ storing the lengths in R' of the homopolymer that $W[j]$ represents. Then, we calculate the median m of the values in $\mathcal{L}$, and insert m copies of the base associated with c^*. Our experimental results showed that the median produces more accurate estimates of the homopolymer length than the mean. We proceed similarly during the extension steps: when we insert the label of an edge into L, we check which values are metasymbols, and estimate the length of their associated homopolymers like with $W[j]$. Notice that constructing $\mathcal{L}$ is simple in our encoding: let i be a position in R^h stored in one of the leaves under v's subtree. To get the homopolymer length for $W[j]$ in the occurrence $R^h[i,i+|W|-1]=W$, we compute $K[\mathsf{rank}_1(B,i+j-1)]$.

7 Experiments

Software. We implemented our suffix-tree-based algorithm to spell the (ℓ,h)-tigs of a voDBG $\mathcal{G}_{\ell,h}$ constructed from a collection of run-length-compressed HiFi reads and their reverse complements. This implementation was written in C++ and on top of the SDSL-lite library [11]. We call our tool Ryu (龍), the Japanese word for dragon[3]. We use the compressed suffix tree (CST) proposed by Sadakane et al. [26] to run our algorithm. This data structure is implemented in the class template `cst_sada<>` of the SDSL-lite library. The relevant template parameters that affect the performance of Ryu are the underlying compressed suffix array (CSA) [12] used by the CST, the sampling rates for the suffix array (SA), and inverse suffix array (ISA). In our case, we used the CSA proposed by Sadakane et al [25], with sampling rates of 1/16 and 1/32 for the SA and the ISA (respectively). The source code of Ryu is available at https://github.com/ddiazdom/Ryu.

Data sets. We ran experiments on real Pacific Biosciences HiFi data from the *E. coli* genome and on simulated reads from the *C. elegans* and *D. melanogaster* genomes. We decided not to use real HiFi data for *C .elegans* and *D. melanogaster* as our method does not yet consider the case of polyploid genomes. The *E. coli* dataset was downloaded from the SRA database (accession SRR10971019). The reads for *C. elegans* and *D. melanogaster* were simulated with the script `randomreads.sh` from the BBTools suite [15]. We used the same parameters for both genomes; the PacBio error model (`pacbio=t`), with error rates that ranged from 0.01 to 0.001 (`pbmin=0.001` and `pbmax=0.01`). We set the minimum read length to 9,000 base pairs (`minlength=9000`) and the maximum length to 12,000 base pairs (`maxlength=12000`). Finally, we set the coverage to 45 (`coverage=45`). The details of all the data sets used in the experiments are given in Table 1.

Test Machine. All our experiments were conducted on a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The OS was Linux (Ubuntu 18.04.5 LTS) running kernel 5.4.0-58-generic.

Benchmarks. First, we assembled the datasets with Ryu using different values for ℓ and h. The results for these tests are shown in Table 2. Subsequently, we assembled the datasets by setting ℓ and h according to the recommendations of our analysis in Section 4. We assumed an error rate of 0.1% for all the datasets,

[3] Finnish speakers may see the reason for this name, a combination of the "lo-hi" thresholds that drive the assembly and the "snake"-like movement of the algorithm through the suffix tree.

Table 1. Data sets used in the experiments

Genome	Genome size (Mbp)	Number of reads	Total length of reads (Mbp)	Sequencing Coverage
E. coli	4.63	95,514	1,389.5	299.48
C. elegans	100.28	429,798	4,309.8	42.98
D. melanogaster	143.72	615,968	6,176.8	42.98

accepting a probability of 1% for misassemblies. These assumptions yielded the values $\ell = 24$ and $h = 32$ for *E. coli* and $\ell = 20$ and $h = 26$ for *C. elegans* and *D. melanogaster* (see the rows marked with a $\diamondsuit$ symbol in Table 2). We discarded the contigs with less than 800 base pairs in all the instances. Next, we compared our results against the state-of-the-art assemblers that support HiFi data, HiCanu [20], Hifiasm [6], MBG [24], and LJA [1]. For the comparisons, we selected for each organism the best[4] assembly generated with Ryu (marked with a ♣ symbol in Table 2), but also the one generated with the recommended values for ℓ and h. The results of the comparisons are shown in Table 3. All the software was installed following the instructions of the authors. In the case of Ryu, we used the compiler flags `-O3 -full-enroll-loops -msse4.2 -ffast-math`. We executed the tools using ten threads, except for Ryu, which does not support multithreading yet. The results of MBG depend on two parameters; the kmer size (`-k`) and the windows size (`-w`). We produced different assemblies using distinct values for these parameters, and we picked the instance with the highest N50. In the case of Hifiasm, we used the parameter `-l0` to avoid the purge of duplicates. The authors recommend this parameter for inbred or homozygous genomes. LJA and HiCanu were executed using default parameters. However, the LJA execution ended with a failed assertion on the simulated data sets and thus we were not able to produce results with it on those data sets. We assessed the quality of the assemblies using QUAST [13]. Finally, we remark that, while we have not yet optimized Ryu for speed, its computational bottleneck — suffix array construction — takes under 4 minutes on our test machine for the *E. coli* dataset using the psascan tool [16].

8 Results and discussion

Table 2 shows that Ryu obtains the best results for *E. coli* (in terms of quantity) with $\ell = 17$ and $h = 28$. These values produce five contigs and an assembly of 4.66 Mbp. However, in almost all the instances of *E. coli*, the chromosome is maintained in only one contig. With the recommended values of ℓ and h (namely $\ell = 24$ and $h = 32$), the longest contig misses about 500 Kbp of the genome showing that our analysis is a bit too conservative on this small bacterial genome.

On the larger genomes with simulated reads, *C. elegans* and *D. melanogaster*, Table 2 shows that Ryu obtains the longest contigs with $\ell = 9$ and $h = 17$. Our definition of (ℓ, h)-tigs requires that $\ell > h/2$ and we see that in this case ℓ is set to the minimum possible value. As predicted by our analysis in Section 4, this is not enough to avoid misassemblies and we see in Table 3 that these low values of ℓ indeed produce several misassemblies, whereas with the recommended values for ℓ and h, no misassemblies are produced. The assembler with the poorest performance in all datasets was HiCanu. We believe this problem could have occurred due to an incorrect setting of parameters.

Table 3 shows that Ryu, HiCanu, Hifiasm, and LJA all produce an assembly where the *E. coli* genome is fully included in the largest reported contig, whereas MBG produces a more fragmented assembly. The number of misassemblies and the accuracy as measured by number of differences to the reference genome per 100 Kbp are higher for HiCanu, whereas the results for other assemblers are comparable. We note that the number of misassemblies and differences to the reference genome remains the same for both instances of Ryu. This situation may indicate that the misassemblies are real differences between the *E. coli* reads and the reference genome.

On the simulated *C. elegans* data, hifiasm assembles the six chromosomes into 10 contigs thus producing a near perfect assembly, whereas the other assemblers produce more contigs. Also on the *D. melanogaster* data, hifiasm again produces the fewest contigs. The assembly of HiCanu on these data sets contains only a

[4] The criteria for selecting the best assembly were the length (the closer to the original reference genome the better), then the number of contigs and then the N50 value.

Table 2. Performance of Ryu on different values of ℓ and h. The rows marked with a ♣ symbol are the best assembly instances. The rows marked with a ♢ symbol are the assembly instances generated with the pair $[\ell, h]$ recommended by our analysis of Section 4.

Genome	$[\ell, h]$	Length of assembly (Mbp)	Number of contigs	Longest contig (Kbp)	N50 (Kbp)
	$[9, 16]$	4.74	38	4,653.18	4,653.18
	$[9, 17]$	4.74	41	4,653.18	4,653.18
	$[13, 22]$	4.68	12	4,651.97	4,651.97
E. coli	$[13, 25]$	4.68	12	4,651.97	4,651.97
	♣$[17, 28]$	4.66	5	4,650.09	4,650.09
	$[17, 31]$	4.67	5	4,651.32	4,651.32
	♢$[24, 32]$	4.67	5	4,123.88	4,123.88
	$[9, 16]$	99.52	1,766	841.38	149.516
	♣$[9, 17]$	99.53	1,653	1,089.53	152.308
	$[13, 22]$	98.70	2,674	737.80	90.80
C. elegans	$[13, 25]$	98.87	2,438	737.80	93.43
	$[17, 28]$	97.47	3,830	407.19	56.43
	$[17, 31]$	97.66	3,742	407.19	57.53
	♢$[20, 26]$	96.03	5,055	317.23	42.31
	$[9, 16]$	130.18	5,250	1,008.24	190.88
	♣$[9, 17]$	130.30	4,965	1,008.24	191.03
	$[13, 22]$	126.71	4,935	1,007.79	174.60
D. melanogaster	$[13, 25]$	127.40	4,687	1,007.85	175.66
	$[17, 28]$	124.00	4,548	1,007.62	165.82
	$[17, 31]$	124.66	4,586	1,007.62	165.82
	♢$[20, 26]$	121.61	4,365	1,007.57	140.59

fraction of the genome. The N50 value of the Ryu assembly on both of these data sets is the second highest, both when considering the ℓ and h values producing the most contiguous assembly and the recommended ℓ and h values. With the recommended ℓ and h values, the accuracy of Ryu as measured by misassemblies and differences per 100 Kbp is as good as the accuracy of the other assemblers on the *C. elegans* data and better on the *D. melanogaster* data where Ryu has the lowest number of differences per 100 Kbp and it is the only assembler producing no misassemblies.

9 Conclusion

We have presented the first rigorous definition of contigs for voDBGs, (ℓ, h)-tigs, and an algorithm for enumerating them. We analysed how the values of ℓ and h affect the contiguity and correctness of (ℓ, h)-tigs and the experiments confirmed that choosing the values according to the analysis produces accurate and contiguous (ℓ, h)-tigs. A unique aspect of our analysis of misassemblies is considering missing data, i.e. situations where one copy of a repeat is partially covered by reads, whereas another copy does not have covering reads. These situations often cause misassemblies in genome assembly but still we are not aware of other assemblers analysing these situations and taking them into account in the assembly algorithms.

Our current definition of (ℓ, h)-tigs does not take into account the possible diploid nature of the data. Thus the produced (ℓ, h)-tigs should be considered comparable to unitigs in a regular DBG and further analysis such as bubble popping would be needed on the produced (ℓ, h)-tigs and their connections to implement a full assembler using voDBGs. Nevertheless, experiments with the assembler — Ryu — we have implemented based on our algorithm show that on haploid data the (ℓ, h)-tigs are comparable or better in contiguity and accuracy as compared to contigs produced by DBG assemblers on HiFi data.

Table 3. Comparison of Ryu to full assemblers for HiFi data. The rows marked with a ♣ symbol are the best assembly instances from Table 2. The rows marked with a ♢ symbol are the assembly instances generated with the pair $[\ell, h]$ recommended by our analysis of Section 4.

Genome	Assembler	Length of assembly (Mbp)	N50 (Kbp)	Number of contigs	Longest contig (Kbp)	Number of Misassemblies	Diff. per 100 Kbp
E. coli	♣ Ryu	4.66	4,650.09	5	4,650.09	5	0.09
	♢ Ryu	4.67	4,123.88	5	4.123.88	5	0.09
	HiCanu	4.71	4,669.66	3	4,669.66	17	3.38
	Hifiasm	4.66	4,665.55	1	4,665.55	6	0.09
	LJA	4.64	4,642.46	1	4,642.46	3	0.19
	MBG	4.89	331.53	57	870.52	5	0.18
C. elegans	♣ Ryu	99.53	152.30	1,653	1,089.53	65	0.90
	♢ Ryu	96.03	42.31	5,055	317.23	0	0.08
	HiCanu	10.37	17.39	606	50.19	0	10.69
	Hifiasm	100.21	17,493.31	10	18,370.70	0	0.01
	MBG	96.59	6.31	26,458	53.07	0	0.24
D. melanogaster	♣ Ryu	130.30	191.03	4,965	1,008.24	57	1.65
	♢ Ryu	121.61	140.59	4,365	1,007.57	0	0.03
	HiCanu	34.56	20.13	1,783	78.64	15	11.36
	Hifiasm	137.70	19,748.06	156	21,678.52	12	0.42
	MBG	131.61	7.76	33,351	64.54	4	0.42

References

1. Anton Bankevich, Andrey Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A. Pevzner. LJA: Assembling long and accurate reads using multiplex de Bruijn graphs. bioRxiv, 2021.
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. Prjibelski, 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 and Fabio Cunial. Fully-functional bidirectional Burrows-Wheeler indexes and infinite-order de Bruijn graphs. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 128, pages 10:1–10:15, 2019.
4. Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proc. 25th 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. Haoyu Cheng, Gregory T. Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18:170 – 175, 2021.
7. Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2013.
8. Chen-Shan Chin and Asif Khalak. Human genome assembly in 100 minutes. bioRxiv, 2019.
9. Diego Díaz-Domínguez, Simon J. Puglisi, and Leena Salmela. Computing all-vs-all MEMs in run-length encoded collections of HiFi reads. *arXiv preprint arXiv:2208.14787*, 2022.
10. Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968, 2021.
11. 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.
12. Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
13. Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.
14. Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2(2):291–306, 1995.
15. Joint Genome Institue. BBTools, 2021. Last accessed: 2021-11-04. URL: https://sourceforge.net/projects/bbmap/.

16. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In *Proc. 26th Annual Symposium Combinatorial Pattern Matching (CPM)*, volume 9133, pages 329–342. Springer, 2015.
17. Yu Lin and Pavel A. Pevzner. Manifold de Bruijn graphs. In *Proc. 14th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 296–310, 2014.
18. Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.
19. Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.
20. Sergey Nurk, Brian P. Walenz, Arang Rhie, Mitchell R. Vollger, Glennis A. Logsdon, Robert Grothe, Karen H. Miga, Evan E. Eichler, Adam M. Phillippy, and Sergey Koren. Hicanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome Research*, 30(9):1291–1305, 2020.
21. Yu Peng, Henry C.M. Leung, Siu-Ming Yiu, and Francis Y.L. 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.
22. Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.
23. Martin Raab and Angelika Steger. "Balls into bins" — a simple and tight analysis. In *Proc. 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 159–170, 1998.
24. Mikko Rautiainen and Tobias Marschall. MBG: Minimizer-based sparse de Bruijn graph construction. *Bioinformatics*, 37(16):2476–2478, 2021.
25. Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
26. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
27. Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017.
28. Aaron M. Wenger, Paul Peluso, William J. Rowell, Chang, et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37:1155–1162, 2019.