REVIEW

Overlap graphs and *de Bruijn* graphs: data structures for *de novo* genome assembly in the big data era

Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, Paola Bonizzoni*

Department of Informatics, Systems and Communications, University of Milan-Bicocca, Milan, Viale Sarca 336, 20126, Italy
* Correspondence: bonizzoni@disco.unimib.it

Background: *De novo* genome assembly relies on two kinds of graphs: *de Bruijn* graphs and overlap graphs. Overlap graphs are the basis for the Celera assembler, while *de Bruijn* graphs have become the dominant technical device in the last decade. Those two kinds of graphs are collectively called assembly graphs.
Results: In this review, we discuss the most recent advances in the problem of constructing, representing and navigating assembly graphs, focusing on very large datasets. We will also explore some computational techniques, such as the Bloom filter, to compactly store graphs while keeping all functionalities intact.
Conclusions: We complete our analysis with a discussion on the algorithmic issues of assembling from long reads (*e.g.*, PacBio and Oxford Nanopore). Finally, we present some of the most relevant open problems in this field.

Keywords: overlap graphs; *de Bruijn* graphs; genome assembly; long reads; string graphs

Author summary: All available assemblers are built upon the notions of *de Bruijn* graphs or overlap graphs. This review discusses the most recent advances in the problem of constructing, representing and navigating those graphs, focusing on very large datasets. Moreover, we present some of the most relevant open problems.

INTRODUCTION

The widespread use of next generation sequencing (NGS) technologies, which is due to their reduced cost, has revitalized the algorithmic research focused on *de novo* assembly, since the huge amount of available data poses new computational challenges. A fundamental tool used for *de novo* assembly is a graph representation of the relationships between the portions of the genome, called *reads*, sharing a common prefix and suffix. A commonly used representation is the *de Bruijn graph*, which is based on the notion of k-mers — a length k substring of a read. In the context of genome assembly, a *de Bruijn* graph is a graph whose vertices are the k-mers of the reads, and each edge connects two k-mers that share a common prefix and suffix of length $k-1$. Another graph representation is the *overlap graph*, where we have a vertex for each read, and

two vertices r_1 and r_2 are connected by an edge (r_1, r_2) when a suffix of r_1 is equal to a prefix of r_2. Those two kinds of graphs are called *assembly graphs*. Clearly, overlap graphs are more informative (with respect to the input reads) than *de Bruijn* graphs, but they usually require more computational resources to be built and stored. On the other hand, the simpler structure of *de Bruijn* graphs can allow some more sophisticated arguments to be applied.

Both representations share the idea that a genome assembly corresponds to a path in the graph: for this reason, the step following the construction of such a graph is the extraction of relevant paths. Under ideal conditions, such as the absence of errors and repeats, we can reconstruct only one relevant path in such graph (that is, there is only one possible assembly). Unfortunately, errors and repeats are common, hence there may be several paths

in the graph, each corresponding to a possible assembly. For this reason, all widely used assemblers compute and output, from a graph representation of reads, a set of strings called *contigs* — putative portions of the complete genome.

While contig assembly is a notion that goes back 20 years, surprisingly few efforts have been spent to produce a deep theoretical and practical investigation of how to characterize the substrings that can be extracted from an assembly graph. In this direction, the notion of *omnitigs* [1] — the set of contigs appearing in all possible paths of the assembly graph — deserves further investigation.

In the literature, both *de Bruijn* and overlap graphs have been considered for designing genome assemblers. The construction of such graphs is one of the most resource-intensive steps of all assembly algorithms: thus it poses some major scalability issues. The main goal of this survey is to discuss the most important differences between those two approaches to genome assembly, especially regarding the influence of the recent results on compressed data structures, which have introduced in bioinformatics some sophisticated, but computationally efficient, algorithms. Mainly, the Burrows-Wheeler Transform [2] and the FM-index [3] are two powerful tools for indexing huge collections of sequences, such as reads, with the goal of quickly computing information related to the substrings of the reads (*e.g.*, Longest Common Prefix array and the maximal overlap between reads).

This paper surveys the combinatorial and algorithmic aspects of assembly graphs. While there are several surveys on computational aspects of genome assembly, to the best of our knowledge no survey details the computational aspects of assembly graphs. In fact, [4] is a wide-ranging discussion of genome assembly, while [5] and [6] focuses on second generation (short read) data (Illumina's SOLEXA and Applied Biosystem's SOLID) and deal with the general aspects of the assembly pipeline. Another survey [7] focuses on quality assessment and defines an ad-hoc measure for the trade-off between contig lengths and accuracy. Assemblers of Sanger and short read data, from five categories (greedy, OLC, *de Bruijn* graphs, seed-extend, and branch-and-bound) are considered and compared. [8] compares the two approaches based on OLC and *de Bruijn* graphs in relation to second-generation reads and in terms of memory occupation and construction efficiency. An evaluation pipeline is the main focus of [9], which presents and compares ten long read assemblers (third-generation reads) in terms of contiguity, completeness and accuracy. Finally, the main topic of [10] is the scaffolding of contigs by means of several types of external data (such as long-range linking data, physical mapping, long reads, subcloning, paired-end reads, and chromosomal contact

data). The goal of our survey are the theoretical aspects of the two main assembly approaches — OLC and *de Bruijn* graph — paying particular attention to the construction and the "cleaning" of the graph structures and to the support data structures (such as BWT, FM-index and Bloom Filter) in relation both to second-generation (short) and third-generation (long) reads. Moreover, this survey proposes a list of open problems which are of some interest.

We will introduce the formal definitions of overlap graphs and *de Bruijn* graphs from a mathematical point of view, and we will discuss how those graphs can be built, managed, and analyzed efficiently. We point out that efficiency is essential when assembling long genomes, such as human genomes, or when large datasets are involved — as usually is the case for short reads. Moreover, we will also discuss the main differences between assembly graphs obtained from short reads and from long reads. Finally, the last section concludes our paper with a discussion of the most relevant open problems of the field.

PRELIMINARIES

Since reads are essentially substrings of a longer and unknown genome, we need to summarize the data structures that are most widely used to index and query texts. One of the most prominent of these data structures is the Burrows-Wheeler Transform (BWT), which is adopted by the majority of the methods working on NGS data.

The BWT [2] of a set R of reads is a permutation B of their symbols such that $B[i]$ is the symbol preceding the i-th smallest element in the lexicographically ordered set of all suffixes of the reads in R.

To each read r_i in R, a sentinel symbol $\$_i$ that is not in the alphabet Σ of the reads and that lexicographically precedes all symbols in Σ, is appended. Moreover, $\$_i < \$_j$ if the read r_i precedes read r_j in R. In Figure 1 an example of the BWT is presented for a set of two reads. For simplicity the same sentinel $\$$ has been appended to both reads.

Some other widely adopted data structures, which are strictly related to the BWT, are the Generalized Suffix Array (GSA), the Longest Common Prefix (LCP) array, and the FM-index. The GSA [11] of the set R is the array Sa where the element $S_a[i]$ points to the suffix of a read in R that is the i-th smallest element in the lexicographically ordered set of all suffixes of the strings in R. The LCP array [12] is the array L such that $L[i]$ is equal to the length of the longest prefix shared by the suffixes pointed to by $S_a[i]$ and $S_a[i-1]$.

Notice that the i-th symbol $B[i]$ of the BWT is the symbol preceding the suffix pointed to by $S_a[i]$. All suffixes with a common prefix Q appear consecutively in

BWT	Suffixes
t	$
a	$
c	a$
t	acgt$
$	acgtacgt$
c	ca$
t	cca$
a	cgt$
a	cgtacgt$
$	ggtcca$
c	gt$
c	gtacgt$
g	gtcca$
g	t$
g	tacgt$
g	tcca$

Figure 1. Example of the BWT (first column) for the two reads *acgtacgt* and *ggtcca*. The second column lists the suffixes in lexicographical order.

an interval $[b, e)$ on S_a, which is called a Q-interval. Since S_a and B are closely related, also $[b, e)$ on B is called a Q-interval. The backward σ-extension of a Q-interval is the σQ-interval (that is, the interval of the suffixes sharing the common prefix σQ), and can be efficiently computed by means of the FM-index functions C and Occ [3], where $C(\sigma)$ is the number of occurrences in B of symbols that are alphabetically smaller than σ and $Occ(\sigma, i)$ is the number of occurrences of σ in the prefix of the first $i - 1$ symbols of the BWT B.

The forward σ-extension of a Q-interval $[b, e)$ is the $Q\sigma$-interval $[b', e')$, that is, the interval of the suffixes sharing $Q\sigma$ as a prefix. The interval $[b', e')$ is contained in $[b, e)$, and can be efficiently computed by applying the concept of *bidirectional BWT* and *linked intervals* as proposed in [13,14]. Let B^r be the BWT of the set R^r of the reversed reads of R. Then, the Q-interval $[b, e)$ on B and the $rev(Q)$-interval $[b_r, e_r)$ on B^r are *linked intervals*, where $rev(Q)$ is the reverse of Q. Two linked intervals have clearly the same width $b - e = b^r - e^r$. The forward σ-extension of $[b, e)$ (that is, the $Q\sigma$-interval on B) and the backward σ-extension of $[b_r, e_r)$ (that is, the $\sigma rev(Q)$-interval on B^r) are linked intervals. Moreover, observe that for a string Q of just one symbol, $Q = rev(Q)$ and $[b, e) = [b_r, e_r)$. Based on these concepts, the formulas presented in [13] allow to compute efficiently the simultaneous backward σ-extension of $[b_r, e_r)$ and forward σ-extension of $[b, e)$ using only the FM-index functions C and Occ of the BWT B without computing the BWT B^r of the reversed reads. The above case leads to

the well known notion of *bidirectional FM-index* and there is a wide literature on the use of such index to perform pattern search in an amount of time that scales linearly in the size of the pattern P. Indeed, both backward and forward σ-extensions have constant time cost, once functions C and Occ and the BWT B are given.

The main interest for using the BWT and the FM-index functions to analyze NGS data is due to the fact that they lead to a succinct data representation of genomic reads [14,15] and they are quite flexible in providing efficient methods for several crucial tasks in bioinformatics, such as the alignment of reads [16,17]. Most notably, the BWT and FM-index allow to obtain a linear time algorithm to perform a task that previously required quadratic time, that is the operation of computing all prefix-suffix overlaps among reads, avoiding a costly all-against-all comparison among reads [14].

Once the BWT and FM-index of the collection of reads are given, the computation of overlaps is performed by reading in one step all the reads and incrementally computing Q-intervals related to common substrings Q of the reads, that may be prefixes and suffixes of some reads, *i.e.*, overlaps shared by reads. Distinct Q-intervals are obtained by reading suffixes of the reads of increasing length and then iteratively performing backward extensions. The LCP and GSA arrays support the test of verifying when a Q-interval corresponds to a string Q that is an overlap among some reads.

In the literature there are several approaches to construct the BWT of a large collection of short DNA reads. For example BCR and BCRext [18] construct the BWT in external memory and both are based on an iterative procedure, where each iteration j computes the BWT permutation of the symbols preceding the suffixes of length at most j by simulating the insertion, in the BWT that has been computed in the previous iteration, of the suffixes of length exactly equal to j. At each iteration the partial BWT is partitioned into segments $B_j (\sigma)$ (external files) such that $B_j (\sigma)$ is the BWT segment related to the suffixes starting with symbol σ. At the end of the iterations, these segments contain the BWT of the input collection. BCR and BCRext are implemented in the suite BEETL for building and manipulating the BWT of a large collection of strings. Another tool to build the BWT is ropeBWT2 [19] that is suitable for processing long reads as well as short reads. The approach is similar to BCR/BCRext since it works by incrementally inserting sequences into a partial BWT, but it is tailored to be fully in-memory.

More recently, alternative approaches to build a BWT of a collection of reads that are implemented in external memory and with a parallel approach have been introduced [20].

Another important data structure is the Bloom filter

[21]. It allows to store a dictionary, that is a set of elements with membership queries—we can ask whether an element belongs to the set. The usual implementation of Bloom filter consists of a vector of m bits, initialized with zeros, and h hash functions providing h array positions. To store an element, h array positions are set to 1; while the presence of an element is tested by verifying that all the bits at those positions are 1.

Notice that it is easy to extend Bloom filters to solve a problem that is more sophisticated than membership. For example, counting Bloom filters [22] allow to determine the number of occurrences of an element in a multiset— which is the typical use for assemblers.

While Bloom filters are easy to implement, very fast, and they use only a small amount of memory, they introduce two drawbacks: (i) they are probabilistic, so different executions might give different results, and (ii) they introduce some collisions, that is the number of occurrences of some elements is overestimated.

A quick and effective way to determine if two documents are duplicate is the MinHash sketch [23], which essentially consists of estimating the Jaccard coefficient—given two sets A and B, the ratio $\frac{|A \cap B|}{A \cup B}$—by choosing a hash function h and a random permutation p of the universe set and saying that $A = B$ if $\mathrm{argmin}_{x \in A}\{h(p(x))\} = \mathrm{argmin}_{y \in B}\{h(p(y))\}$.

A related idea is minimizers [24]. In this case, two sets A and B of strings are considered identical if the lexicographically minimum string in A is equal to the minimum in B. The typical usage is to test whether two strings a and b share a common substring, by having a sliding window in a and one in b and extracting all fixed-length substrings within each window. If a and b share common substrings, we can find the corresponding windows inside the substring, and the two sets of fixed-length substrings are the same—which is checked by looking at the minimum substring in each set.

ANATOMY OF A DNA ASSEMBLY PIPELINE

Many tools for assembling DNA fragments were proposed in the last 30 years, each one implementing different strategies to achieve its goal. Although assemblers differ in performance, results, and method, almost every pipeline follows the same steps, namely reads correction, assembly graph computation, graph cleaning, contig computation, and scaffolding.

Depending on the sequencing technology used to produce the input reads, each step might overcome different challenges. For example, errors produced by NGS machines and 3rd generation sequencing machines differ in rate, distribution, and type, thus read correction tools shall account for them.

Clearly, read correction tools aim to remove sequencing errors from the reads. Depending on the input sequences, read correction tools may build a consensus sequence between the reads or exploit the shared sequences between the reads to correct them [25].

The second step of DNA assembly pipelines aims to build a graph representing overlap relations between reads or substrings of reads. To this purpose, two types of graphs are used in the literature: overlap graphs and *de Bruijn* graphs. The main difference between the two is that the former represents either exact or approximate overlaps between the reads, whereas the latter represents only perfect overlap between substrings of fixed length of the reads. Selecting one representation over the other is not a trivial task and is application dependent. Indeed, building overlap graphs is usually much more computationally expensive than building *de Bruijn* graphs but, on the other hand, *de Bruijn* graphs are difficult to use with long reads due to the high error rate.

The third step, graph cleaning, aims to remove errors from the graph that reads correction tools were not able to remove. There are three main categories of graph errors that require cleaning, namely tips (short dead-end paths diverging from the main path), bubbles (parallel short paths that start and end in the same two nodes), and bulges (short low-coverage paths that create alternate paths between two nodes).

After this last step, DNA assembly pipelines build contigs. Contigs are contiguous length of genomic sequence in which the order of bases is known to a high confidence level [26]. In this step residual bubbles in the graph are usually considered heterozygous loci and, depending on the application, are either maintained or removed.

The last step is scaffolding; in this step contigs are sorted exploiting some additional information (*e.g.*, mate-pair information), the distances between contigs are estimated, and scaffolds (*i.e.*, successions of contigs and gaps) are produced. Gaps in scaffolds represent portions of the genome that the assembly pipeline was not able to infer and are represented by the letter N in the sequence.

Assembly graphs

We will consider two different types of assembly graphs: (1) those stemming from the Overlap-Layout-Consensus (OLC) approach—the overlap graph and the related string graph—and (2) the *de Bruijn* graph.

The OLC is an approach to assemble a genome from a set of strings (also called *reads*) extracted from the genome, and it is based on the construction of the so-called *overlap graph* whose vertices are the input reads and arcs are the overlap relations between reads, and the reconstructed sequence is obtained by a visit of the paths

of the graph. Sometimes we are interested in the string graph, that is obtained from the overlap graph after removing all redundant arcs. This approach is basically composed of three steps: (i) computing the overlaps between the reads, (ii) laying out the overlap information on a graph, and (iii) inferring the consensus sequence. Observe that both the overlap graph and the string graph were initially proposed by Myers [27] before the advent of NGS technologies, and they allow to assemble overlapping sequences into larger contigs. The computation of the overlap graph is time and space consuming and is considered the bottleneck of an OLC pipeline [28].

A practical advantage of overlap/string graphs over *de Bruijn* graphs is that they can immediately disambiguate short repeats that *de Bruijn* graphs might resolve only at later stages.

Given a set R of reads, the *overlap graph* G_O [27] is the directed graph (R, E) whose vertices are the strings in R, and the pair (r_i, r_j) of string r_i and string r_j is an arc in E if a suffix of r_i is equal to a prefix of r_j, or in other words they have an *overlap*. The suffix of r_j exceeding the overlap is usually called *extension* $e_{i,j}$ of r_i with r_j and is used as the *label* of the arc (r_i, r_j) (see Figure 2 for an example on a set of five reads). Observe that also the prefix of r_i before the overlap can be used as the label of the arc (r_i, r_j).

A path $(r_{i_1}, \ldots, r_{i_k})$ in the overlap graph represents a string w that is obtained by assembling the reads of the path. More precisely, such string is the concatenation $w = r_1 e_{i_1,i_2} \ e_{i_2,i_3} \cdots e_{i_{k-1}, i_k}$ [29]. In particular, an arc (r_i, r_j) is a path with only two vertices representing the string $s = r_i e_i j$. An arc (r_i, r_j) is called *reducible* if there exists a path from r_i to r_j, including some other vertex, representing the same string of (r_i, r_j). Reducible arcs are not helpful in assembling reads, and they can be removed from the overlap graph, hence obtaining the string graph. We can now introduce the notion of *de Bruijn* graph.

Given a set of reads R, a *node-centric de Bruijn* graph

(dBG) (V, E) of order k of the set R, is a graph whose vertices are all the distinct substrings of length k of R (called *k-mers* in the bioinformatics literature and also *k-grams* in the computer science community), and the pair (u, v) of *k*-mers is an arc of E if the length $(k-1)$ suffix of u is equal to the length $(k-1)$ prefix of v. The label of the arc (u, v) is the last character of the node v. The *edge-centric* definition of a *de Bruijn* graph requires the additional condition that the $(k+1)$-mer, obtained by concatenating u and the last character of v (or the first character of u and v), occurs in some read of R. Clearly, from an edge-centric graph of order k we can derive a node-centric *de Bruijn* graph of order k, but not the opposite. Note that a *node-centricde Bruijn* graph may induce incorrect assemblies of the original sequence. In Figure 3 an example of *de Bruijn* graph for two reads is depicted.

ASSEMBLING SHORT READS

In this section we will discuss how string graphs or *de Bruijn* graphs are (1) constructed, (2) stored from a methodological point of view, and then (3) how each graph is exploited to complete the assembly.

Since string graphs are completely generic (for instance there is no upper bound on the number of arcs incoming or outgoing from a vertex), there are not huge differences between storing string graphs and storing general graphs. For this reason, we will not discuss how to store string graphs. In fact, the computation of an implicit representation of an overlap graph is indeed one of the simplest steps that can be done quite efficiently by a single scan of the collection of reads [30].

Symmetrically, the procedure to build a *de Bruijn* graph is quite standard: scan all reads to determine the set of *k*-mers, eventually dropping *k*-mers that are too infrequent (likely due to errors) or too frequent (likely belonging to repeats). For this reason, we will not discuss how to build *de Bruijn* graphs. Although building *de Bruijn* graph is
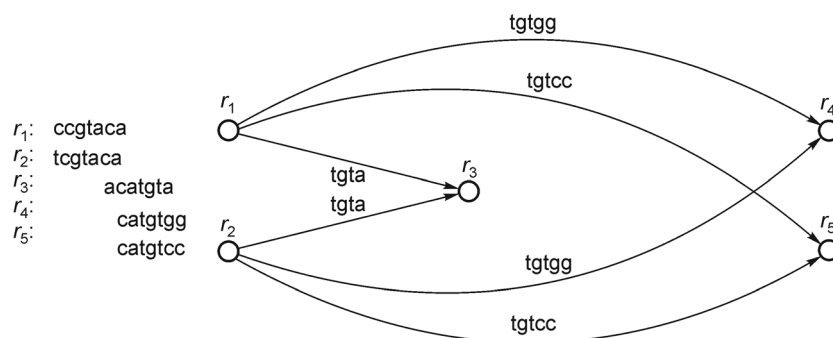

r_1: ccgtaca
r_2: tcgtaca
r_3: acatgta
r_4: catgtgg
r_5: catgtcc

Figure 2. **Example of the overlap graph for five reads** r_1, r_2, r_3, r_4, r_5. Each edge (r_i, r_j) is labelled by the *extension* $e_{i,j}$.
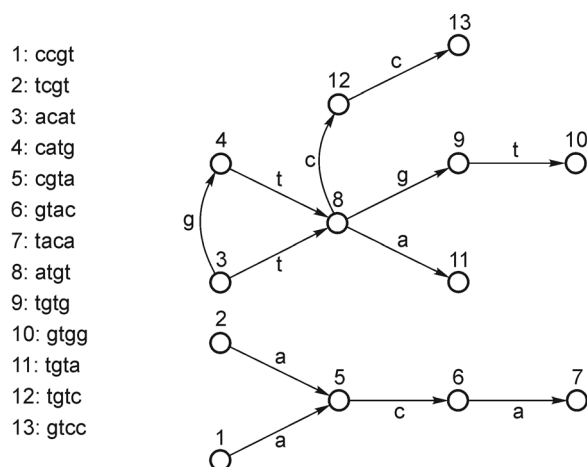
1: ccgt
2: tcgt
3: acat
4: catg
5: cgta
6: gtac
7: taca
8: atgt
9: tgtg
10: gtgg
11: tgta
12: tgtc
13: gtcc

Figure 3. Example of the ***de Bruijn*** **graph for** ***k* = 3 of the two reads** ***ccgtac*** **and** ***catgtg***. The nodes are the sixteen *k*-mers reported on the left. Each arc is labelled by the last character of its second node.

relatively easy in theory, extracting meaningful information from it might be challenging. In this setting, the information we want to retrieve from the graph is a set of contig, *i.e.,* paths describing substrings of the sequenced genome. The main obstacles in this task are errors in the input reads and repeated regions in the genome that create short dead-ended paths and entangle the graph, yielding to a structure that is difficult to analyze. Removing dead-ended short paths from the graph is quite straightforward (see Section of "Anatomy of a DNA assembly pipeline") whereas dealing with repeated regions is usually a much more difficult problem. Many different methods were proposed in the literature to deal with repeated regions of the genome and, surely, many more will be proposed in the next years. For example, IDBA [31] solves this problem by iterating the construction and analysis of the *de Bruijn* graph for increasing values of *k*, Paired *de Bruijn* Graphs [32] includes mate pair information in the structure of the graph, Pathset graphs [33] exploit the mate pair information to estimate the distance between edges in the assembly graph, SEQuel [34] almost completely avoids this problem by using an extension of the *de Bruijn* graph (called positional *de Bruijn* Graph) that incorporate the approximate positions of the reads in the sequenced genome, and AlignGraph [35] distinguishes paths for repetitive regions by incorporating mate pair information and alignment position in the nodes of the graph.

Although this task is extremely important in bioinformatics, we will not provide a deeper description of the methods since this survey's main focus is on methods to construct, represent, and navigate assembly graphs.

Constructing string graphs

This section is devoted to presenting some of the tools for building the string graph of a set of reads. The most used tool is SGA (String Graph Assembler) [14,15]. SGA is based on the Burrows-Wheeler Transform and the FM-index of the set of reads. Other tools are Fermi [36] and Readjoiner [28]. Fermi is inspired by SGA and is tailored for variant calling. Readjoiner is based on an efficient computation of a subset of exact suffix-prefix matches of the overlap graph. The first external-memory tool to build a string graph is LSG (Light String Graph) [30] and is based on the BWT and the FM-index of the set of reads. Observe that SGA, Fermi and Readjoiner are SGAs since they have also an assembly phase, while LSG only computes the string graph, but can use SGA's assembly procedure.

Readjoiner [28] is based on an efficient computation of a subset of exact suffix-prefix matches (overlaps) and, by subsequent rounds of suffix sorting, scanning, and filtering, outputs the irreducible edges of the graph. Readjoiner preprocesses the input set by removing the reads which are a prefix or suffix of some other read. This is performed via radix sort on strings, which is applied to the input set expanded with the reverse-and-complemented reads. There are the following two steps: First, determining all proper suffixes which may be involved in a suffix-prefix match (SPM-suffixes), *i.e.*, all read suffixes of length at least l_m (l_m is a parameter equal to the minimum overlap length) sharing a prefix of length $\geq k$ with some read in the input set — $k \leqslant l_m$ is a parameter chosen on the basis of a time/space trade-off. Then, from the sorted set of SPM-suffixes, the algorithm computes the suffix-prefix matches (overlaps) and outputs the irreducible edges of the graph, by subsequent rounds of suffix sorting, scanning, and filtering.

Notice that Readjoiner uses some heuristics in building the string graph, which is only an approximation of the string graph defined in Section of "Preliminaries".

SGA [14,15] constructs the FM-index of the input set *R* of reads, together with the bidirectional BWT [13], and uses those structures to efficiently compute the arcs of the string graph. Indeed, SGA uses the FM-index of the input reads and of their reversed version. The overlaps between the reads are computed in the following way. Each read *r* in the input set *R* is processed in three steps: (1) all the suffixes of *r* are considered by increasing length and, by performing backward extensions on the BWT of *R*, all the reads having overlap with *r* are found (this step is to find overlaps between reads on the same strand); (2) the set $\overline{r}$ of reverse-and-complemented reads is obtained. Then all suffixes of $\overline{r}$ are considered by increasing length and, by performing backward extensions on the BWT of *R*, all the

reads overlapping with $\overline{r}$ are found (the goal is to find overlaps between reads on opposite strands sharing a prefix); (3) the set $\widetilde{r}$ of complemented reads is obtained. Then all suffixes of $\widetilde{r}$ are considered by increasing length and, by performing backward extensions on the BWT of the reversed reads of R, all the reads overlapping with $\widetilde{r}$ are found (the goal is to find overlaps between reads on opposite strands sharing a suffix). The bidirectional BWT is used to detect and output directly the irreducible arcs of the string graph.

The pipeline of SGA has a preprocessing step for filtering and trimming the input reads with multiple low-quality or ambiguous base calls [15]. A preliminary FM-index is built in order to correct errors using k-mer frequencies. Then the string graph is computed. After that, SGA determines contigs from the string graph and then links them into scaffolds by using paired-end reads.

SGA allows to use Ropebwt2 [19] and SA-IS to index the input reads. Ropebwt2 [19] (an in-memory implementation of BCR [18]) works well with short strings ($\leqslant$ 200 bp), while SA-IS, implementing the suffix array construction algorithm based on the induced sorting [37], is slower but works for longer sequences.

LSG [30] is based on the BWT and FM-index of the input reads. Its goal is to minimize the amount of data maintained in Random Access Memory (RAM). LSG is the first disk-based exact algorithm to construct a string graph that does not require to have the whole original dataset in main memory. LSG uses a slightly modified version of BEETL as a preliminary indexing step on the input reads in order to compute in external memory the Generalized Suffix Array, the BWT and the LCP array of the input reads. The LSG algorithm is based on two steps: (1) computing the overlap graph, and (2) removing the reducible (transitive) arcs. In the first step a single synchronous scan of the BWT, the Generalized Suffix Array, and the LCP array allows to compute all overlaps that are of length at least l_m (where l_m is the minimum length of an overlap), and to represent them as intervals of the BWT (also called BWT intervals). The second step starts from those intervals and, by a sequence of backward extensions, computes the labels of the overlap graph and, at the same time, produces only the irreducible arcs of the string graph. The backward extensions of BWT intervals are computed by using an external-memory approach similar to the one presented in [38].

A variant of LSG is FSG [39] which queries the FM-index of the reads to construct the string graph, without the need of additional expensive data structures. A main characteristics of FSG is that it performs only linear scans, resulting in a cache-efficient implementation.

Fermi [36] is inspired by SGA and tailored for variant calling. A main data structure of Fermi is a single bidirectional FM-index, the so-called FMD-index which is used to represent both DNA strands inside a unique structure. The FMD-index is the FM-index of the bidirectional collection of the input reads, interpreted as the collection of the reads and their reverse-and-complement. Notice that the FMD-index merges into a unique data structure the two FM-index structures of SGA. The indexing phase is based on the SA-IS algorithm [37].

Using string graphs

In the ideal case, that is when reads do not have errors, once we have the string graph, the assembled genome can be obtained by building a traversal of the graph that touches each vertex exactly once — a Hamiltonian path. Unfortunately, to find a Hamiltonian path is an NP-hard problem [40], at least in the worst case. When we also do not have repeated regions, the string graph can have some additional properties (they are acyclic) that make it much easier to find a Hamiltonian path, but this is too strong of a condition, therefore we need some heuristics to obtain the assembly from the string graph.

With the above ideas in mind, we now show how these string graph tools of the previous section perform in practice, by comparing their running times and memory as part of an entire pipeline aimed at assembling the NA12878 sample of the 1000 Genomes Project read group "20FUK", which is composed of approximately 875 million reads, each of 101 bp in length. SGA required 1,112 minutes and 26 GB of RAM for the indexing phase, and 4,145 minutes and 43 GB of RAM for the string graph construction phase. LSG required 9,540 minutes and 52 GB of RAM for the indexing phase, and 9,444 minutes and less than 1 GB of RAM for the string graph construction phase. LSG and SGA share the assembly phase which required 1,637 minutes and 63 GB of RAM. On the other hand, Readjoiner completed the entire pipeline in 713 minutes and 42 GB of RAM.

Representing a *de Bruijn* graph

Bloom filters and probabilistic representation

Conway and Bromage succinct data structure. In [41] the authors propose the first memory efficient representation of a dBG. This method does not store the labels of the nodes explicitly, since they can be inferred from the edges. The approach works as follows. First, for each node u, a vector of four bits (0 and 1) is created, such that there exists one bit for each symbol σ in the DNA alphabet $\Sigma = \{a, c, g, t\}$. Such bit is set to 1 if the string obtained by adding σ at the end of the suffix of u of length $k-1$ is a k-mer, otherwise it is set to 0. All those vectors are concatenated into a (unique) vector CB of bits

according to the lexicographic order of the nodes (*k*-mers): the identification of the segment related to a given *k*-mer is performed on the basis of its rank. The vector *CB* represents the edges of the dBG and, together with a vector of integers that records, for each edge, the number of occurrences of the string joining its two nodes, is a succinct data structure which allows an efficient navigation of the graph using the so-called *rank* and *select* queries. Since *CB* is sparse, it can be efficiently compressed using state-of-the-art approaches. This structure leads to a memory occupation close to the information-theoretical lower bound and experimental results show that 28.5 bits per edge are required. As stated by the authors, the main bottleneck of this approach is sorting the nodes (or *k*-mers).

Minia. To further reduce the memory usage, Minia [42] exploits the following main ideas: (1) use Bloom filters to define *probabilistic de Bruijn* graphs, (2) store in main memory a data structure for restricted queries on the *de Bruijn* graph, such as specifying for each node the list of neighbors, and then use disk memory for other queries such as listing the nodes.

For each *k*-mer in the *de Bruijn* graph, the corresponding bits in the Bloom filter are set to 1 and a traversal of the graph is achieved by testing all the possible nodes that can be reached by the current node. Clearly, since this approach uses a probabilistic data structure, spurious nodes can be found in the Bloom filter. Nevertheless, it is easy to show that only false positive nodes can be added to the graph. To avoid this problem, the authors propose to store an additional data structure that contains all the false positive nodes that can be reached by nodes in the *de Bruijn* graph, and they provide a constant-memory algorithm to compute them.

Using Bloom filters to store *k*-mers is extremely efficient and the authors show that this data structure leads to a memory occupation of 13 – 14 bits per edge.

Noting that the union of the false positive nodes is in turn a set, the authors of [43] proposed to use Bloom filters in a "cascading" way, storing in turn false positives of each set in a smaller Bloom filter and directly storing the false positives only when this set is adequately small. Experimental analysis shows that using only 4 Bloom filters in a cascading fashion leads to using between 30% and 40% less memory.

ABySS 2.0. ABySS 2.0 [44], just as with Minia, uses a Bloom filter representation of the *de Bruijn* graph. Its innovative unitig assembly stage, which is performed on the Bloom filter, allows to reduce the memory requirements by an order of magnitude, while maintaining results comparable with existing assemblers. This aspect permits the assembly of large genomes on a single machine. The unitig phase extends (by using the same graph traversal of Minia) *solid* reads, consisting entirely of *solid k*-mers

(that is, *k*-mers with an occurrence above a user-specified threshold), left and right in order to create unitigs. Only the nodes of the *de Bruijn* graph are stored in the Bloom filter and (during the graph traversal) all four possible neighbors of the current *k*-mer need to be queried. An additional Bloom filter is employed to record *k*-mers included in previous unitigs in order to avoid duplicate unitigs: a solid read is extended if there is at least one *k*-mer that is not present in the additional Bloom filter.

BWT based

DBGFM. A significant reduction in space is provided by the approach in [45] that aims to efficiently enumerate all the maximal simple paths in the dBG without loading the whole graph in memory. DBGFM basically consists of an FM-index storing the simple paths used to answer membership and neighborhood queries. In [45] the authors provide a novel low memory algorithm for the enumeration of simple paths and a partitioning strategy based on the usage of minimizers sorted by frequency, showing that the dBG of the human genome can be stored using 4.76 bits per base, providing good query performance.

BOSS. A different approach to store *de Bruijn* graphs has been proposed by Bowe *et al.* [46] with the introduction of the BOSS data structure. The main idea of BOSS is to store and compress only the labels of the edges of a dBG using an FM-index, borrowing ideas from indexes for texts. More precisely, the *de Bruijn* graph is stored as an array *W* of $|E|$ characters and a bit vector *L* of size $|E|$. The array *W* lists the edge labels according to the lexicographic order of the reverse *k*-mers associated to the source node of the edges; clearly all labels of edges outgoing from the same node will be consecutive in *W*. To mark the consecutive labels related to the same source node, the bit 1 in vector *L* is used to distinguish groups of labels leaving distinct source nodes. Similarly to the FM-index, BOSS allows to efficiently search patterns and *k*-mers in the dBG and, just as for the BWT, *W* and *L* can be easily compressed.

The original paper describing this approach allows to store only a *de Bruijn* graph for a fixed *k*; later works improved this result allowing to store variable order dBGs [47,48] (*i.e.*, all the dBGs of order between 1 and *k*) and colored dBGs [49–51] (*i.e.*, dBGs where each vertex is associated to one or multiple colors).

Although BOSS and its extensions are extremely efficient in space, their construction is not trivial for huge graphs with billions of nodes. Indeed, efficiently sorting *k*-mers and edges labels is a difficult task that might lead to the saturation of the available RAM. To overcome these limitations, recent works [52] proposed new algorithms for the construction of BOSS that first

build the data structure for small partitions of the graph and then merge them together, without saturating the main memory.

Using *de Bruijn* graphs

Repeats that are longer than k, but definitely shorter than a read, affect *de Bruijn* graphs but not string graphs. In fact if a repeat is such that there exists a read spanning the repeat (a so called *bridged* repeat) such that the two portions of the read preceding and following the read are unique, then the string graph is the same as the one when the repeat is not there, while the two *de Bruijn* graphs are clearly different.

There are two main advantages of *de Bruijn* graphs: (1) repeats (both long and short) are more easily detectable, since they correspond to specific subgraphs (called *bubbles*), and (2) given a *de Bruijn* graph, genome assemblies correspond to traversals of the *de Bruijn* graphs where each arc is visited exactly once, that is an Eulerian tour [29]: the latter problem can be easily solved in linear time [53].

Since dBGs for mammalian-sized genomes tend to be very large (*i.e.*, if k is equal to 25, the dBG of the human genome has 4.8 billion nodes [41]), it is fundamental to store them in some very efficient data structure. Notice that a simple representation of such graph needs, for each node, 56 bytes to store the k-mer string (assuming $k \leqslant 32$) and the four possible successor nodes. Therefore a total memory of 268 GB is necessary for the human genome (having 4.8 billion nodes). Moreover, assuming that each node is stored in a hash table (otherwise we cannot find each node), an extra 70 GB is needed [41]. Such a huge memory usage is a concern for several genome assemblers. Indeed, the notion of *compacted de Bruijn* graph is used to reduce the space: this is the graph obtained by replacing all its maximal non branching paths from a vertex u to a vertex v with a single edge (u, v), but the compacted graph does not have some of the combinatorial properties of the usual *de Bruijn* graph — recall that a *path* in a graph is a sequence of vertices where only the first and the last are repeated and it is *non branching* if its vertices have indegree and outdegree one except for the last and the first vertex.

ASSEMBLING LONG READS

The single-molecule sequencing technologies Pacific Biosciences (PacBio, SMRT) and Oxford Nanopore Technologies (ONT) are able to produce long reads of 10–100 kb, but have an error rate much larger (10%–15%) than that of Illumina reads (smaller than 1%). Therefore, long reads can simplify repeat detection and

analysis, but require different strategies to build overlap and *de Bruijn* graphs.

For overlap graphs, an actual overlap between two reads consists in a similar — but far from identical — prefix-suffix pair. For *de Bruijn* graphs, the high error rate produces spurious k-mers, disgregating the structure of the graph. Therefore the main challenge that all tools in this section have to face is either to quickly compute all inexact overlaps, or to avoid the disgregation of the graph. A typical initial step is read correction. To that purpose, most assemblers use a strategy based on the analysis of k-mers included in the reads. Another strategy is based on hybrid approaches, where both short and long reads are exploited.

Overlap detection in the OLC assembly methodology for large genomes is still a major bottleneck in this context. Whereas a linear time algorithm for building an overlap graph exists for the case of exact overlap, when we are interested in an overlap with errors, we are still far from optimal algorithms. A more efficient and accurate approach would be to compute the approximate prefix-suffix relation among reads. In this direction, according to our knowledge, the best current algorithmic solution is Canu [54] which uses MHAP [55].

Unfortunately, the current methods essentially have to resort to all-against-all read comparison — at least in the worst case — which is computationally expensive, hence limiting the applicability of the method.

Some ingredients are common to the various approaches that are employed to compute the overlaps, mainly the idea of finding seeds, that is exact short substrings. However these seeds tend to be very short due to the presence of errors, hence they might not give the desired degree of precision. To overcome this problem, some methodologies use local alignment procedures. Those methods differ mainly in the data structures used for finding seeds. An overview of leading read-to-read overlap detection methods is given in [56], where a comparison of performances is also provided. With the improvements of long read sequencing technologies, we expect an increase in the coverage in the near future — while storage is not currently an issue for long reads datasets, it is likely to become a factor.

In this section we describe a number of tools for assembling long reads, summarizing them in Table 1.

Wtdbg2. Wtdbg2 [57] is a long-read assembler for mammalian genomes (large genomes) that is tens of times faster than other long-read assemblers with little compromise on the quality results.

Wtdbg2 basically follows the OLC paradigm: it performs first a fast all-vs-all read alignment and then a layout based on a fuzzy *de Bruijn* graph. More in detail, it splits the reads in chunks of 256 bp (bins) and builds a

hash table having as keys the *k*-mers, occurring at least twice and at most 1,000 times, and as values their positions in the read bins. Then, the input reads are considered, from the longest to the shortest, and all-vs-all read alignments are computed by using the hash table and by applying a dynamic programming (DP) algorithm; the read binning limits the matrix dimension of the DP. Finally, Wtdbg2 applies a layout algorithm based on a fuzzy *de Bruijn* graph, which extends the definition of *de Bruijn* graph in order to tackle noisy reads (avoiding an error correction step). In this graph, a vertex is a *k*-bin, that is a sequence of *k* consecutive bins, and the edges are determined by the alignments computed by the previous step and takes into account the inaccuracy of the input reads.

HINGE. HINGE [58] applies an OLC approach, therefore it builds an overlap graph. In the first phase it obtains the pairwise overlap information by using DALIGNER [59] (and at the same time it discards chimeric reads, that is, reads resulting from sequencing errors and composed of different parts of the genome), but its main novelty consists of its layout phase. In fact, repeats are enriched with *hinges*, that is short substrings that immediately precede or follow the repeat. An analysis of hinges is coupled with the best-overlap graph strategy [60] to construct an overlap graph that aims at reaching the error robustness of the OLC approach and the repeat resolution capability of *de Bruijn* graphs.

ABruijn. The ABruijn assembler [61] is based on a generalization of the *de Bruijn* graph called the *A-Bruijn* graph. The *A-Bruijn* graph $AB(R, C)$ for a set R of reads is constructed just as the *de Bruijn* graph, however C can be any substring-free collection of strings (referred to as a collection of *solid strings*). The *de Bruijn* graph $DB(R, k)$ is identical to the *A-Bruijn* graph $AB(\Sigma^{k-1})$, where Σ^{k-1} is the set of all $(k-1)$-mers in the alphabet Σ of R.

The ABruijn assembler starts with the selection of solid strings. Let a (k, t)-mer be a *k*-mer that appears at least *t* times in the set R of reads. Parameter *t* is then selected by computing the number ϕ of *k*-mers with frequencies exceeding *t*, and selecting a maximum *t* such that ϕ exceeds the estimated genome length. The *A-Bruijn* graph is then constructed based on this collection of (k, t)-mers as the collection of solid strings.

A path in this *A-Bruijn* graph corresponds to an error-prone draft genome (or contig). Since paths are error-prone, two paths in this graph are said to *overlap* if they have a common *jump*-subpath (given a parameter *jump*) of a given minimum span, as computed by a dynamic programming algorithm. Then each path is extended by merging two overlapping paths, provided that the extension is supported by a minimum number of other

paths — the idea being that spurious reads will have low support. Another problem is when a growing path ends in a long repeat. This is overcome by computing a support graph, similar to the strategy used in exSPAnder [62], but slightly more complex, since the *A-Bruijn* graph does not always reveal the local repeat structure like in the case of the *de Bruijn* graph.

After the resulting draft genome is constructed above, the ABruijn assembler has further functionality for correcting errors in this draft genome, for iteratively refining it, etc.

Falcon. The FALCON assembler [63] follows the OLC approach, building a string graph by using DALIGNER [59] to obtain overlaps between the reads. An overlap filtering step is applied to remove contained reads or reads that appear to be from high copy-repetitive regions. The string graph is built with a slight variation of the method described in [27].

The layout and the consensus phases are tweaked to infer both haplotypes, according to the *hierarchical genome assembly process* (HGAP) [64]. FALCON begins by using reads to construct a string graph that contains sets of "haplotype-fused" contigs as well as bubbles representing divergent regions between homologous sequences [65]. Then, FALCON-Unzip identifies read haplotypes using phasing information from heterozygous positions that it identifies. Phased reads are then used to assemble haplotigs and primary contigs (backbone contigs for both haplotypes) that form the final diploid assembly with phased single-nucleotide polymorphisms (SNPs) and structural variants (SVs).

Canu. Canu [54] builds upon the Celera assembler [66], hence following its OLC approach. To avoid the expensive all-against-all comparison for the overlap phase, Canu first selects a pair of reads that might overlap, then those pairs are analyzed to determine if the overlap is actually good enough.

The first part is especially innovative, since it is based on the tf-idf measure [67] that has originated in Information Retrieval to find which words are most interesting in a set of documents. In Canu, that measure is used to find which *k*-mers are most useful to determine if two reads actually overlap — *e.g.*, *k*-mers of low complexity or exceedingly frequent are not useful.

Miniasm. Miniasm [68] builds an overlap graph by mapping all pairs of reads with Minimap [68] — a fast aligner inspired from MHAP [55] and DALIGNER [59]. The main idea is to use the MinHash sketch [23] to have a small, fixed-size and efficient way to compare two sets of *k*-mers. Moreover, it uses minimizers [24] to further reduce the memory needed without an excessive impact on the quality of the overlap detection.

OPEN PROBLEMS AND FUTURE DIRECTIONS

The construction of assembly graphs is a source of inspiration to new challenges and problems for the design of novel data structures and algorithms in bioinformatics. In this section, we discuss the algorithmic aspects of building assembly graphs, in general, but also in the context of when the input is either short or long reads. In the following we list some of the main computational challenges arising from the use of short or long reads in the assembly process.

Since short read technologies have existed for decades before the advent of long reads, the techniques for assembling short reads are much more developed in this first context. In particular, since short read datasets contain too many reads for doing an all-against-all comparison (typically employed in the OLC framework), the assembly based on k-mers is a popular approach in this context — especially given the very small, often less than 1%, error rate of short read sequencing technologies, such as Illumina. On the contrary, since long read technologies such as PacBio and ONT, have error rates of 10%–15%, k-mers are often not reliable enough, hence all approaches tend to take the OLC approach (see Table 1) — especially given that long read datasets have much fewer reads (lower coverage) than short read datasets. This is with the exception of ABruijn [61], which is an interesting approach that leverages the advantage of both k-mers and OLC approaches for long reads.

Open problem 1: approximate prefix-suffix relation on a set of reads

Given a collection of reads, provide a better algorithm and an efficient implementation to compute approximate prefix-suffix relation among reads. To the best of our knowledge, most recent advances on this topic are in [69]. Since the main limitation of long reads is the large computational costs of computing the overlap of reads, a high priority challenge is to reduce the complexity of

implementing the all-against-all paradigm. More precisely, such complexity could be reduced for example by avoiding the comparison of all-against-all reads by applying clustering strategies to reduce the data set or by extending to long reads techniques such as the linear time computation over short reads of the prefix-suffix relation by the FM-index. Indeed, the FM-index has allowed to lower the quadratic complexity bound required by the all-against-all comparison of reads.

Open problem 2: overlap graph on a set of long reads by indexing of reads

Given a collection of long reads, provide an algorithm and an efficient implementation to compute the overlap graph for long reads after error correction, or compute an approximate overlap. The approach discussed in [70] does not seem to be practical for long reads. Exploiting the FM-index on long reads is quite challenging, due to the high error rate which make it not that palatable for computing the approximate overlap of reads.

On the other hand, the computation of the overlap between long reads could take advantage of some known techniques for indexing k-mers. In this direction, BASE [71], for example, is a tool exploring the indexing of reads by a bidirectional BWT for discovering seeds that are common to reads. Though BASE has been experimented only on short reads, the authors claim their method can be extended to long reads.

Open problem 3: memory efficient representation of a *de Bruijn* graph

Finding a memory efficient representation of de Bruijn graphs. Concerning the memory usage, the number of k-mers generated from reads increases with the length of reads and the coverage, and RAM usage is an issue for *de Bruijn* graph assemblers. Indeed, approximately $2(k + 1)G$ Gigabytes of RAM are required to store a k-mer table of a genome of size G [72]. As reported in [72], using *de Bruijn* graphs can be problematic with high

Table 1   List of tools on long reads

Tool	Main strategy	Overlap computation	Error-aware	Repeat-aware
Wtdbg2	OLC	Pairwise alignments (k-mers + dynamic programming)	Y	N
HINGE	OLC	Pairwise alignments (DAligner)	Y	Y
ABruijn	k-mers	k-mers (based on selection of solid strings)	Y	Y
Falcon	OLC	Pairwise alignments (DAligner)	Y	Y
Canu	OLC	Pairwise alignments (tf-idf measure)	Y	Y
Miniasm	OLC	Pairwise alignments (MinHash sketch)	N	N

This table summarizes the main properties of the tools for assembling from long reads that we detail in this article. Note that error- and repeat-aware denote whether the tool does something to handle, respectively, the errors and repeats which are present in the input long reads. For example, Wtdbg2 handles errors by using a fuzzy *de Bruijn* graph, while HINGE handles repeats by enriching them with so-called *hinges*.

coverage or highly repeated genomes — in the first case the use of RAM memory can become prohibitive for large genomes, while in the second case we need to analyze the graph to disambiguate repeats. The first limitation is partially overcome by (1) using string/overlap graph based assemblers of short reads, since, according to [73], the usage of RAM memory is an order of magnitude lower than the one required by *de Bruijn* graphs at the same coverage, and (2) using space-efficient representations of *de Bruijn* graphs that allow to compute and store them on middle-end machines at the cost of increasing the query time [42]. The use of the BWT and of the FM-index to index *de Bruijn* graphs has opened new research directions aiming to reduce space and time usage in the construction as well as in the operations on the graphs. The same ideas have inspired some techniques to index and construct pan-genome graphs, such as variation graphs [74], and could be further investigated, especially in the context of *de novo* assembly. To overcome the problem of repetitive regions in the genome, it seems that long reads are the most promising direction.

Open problem 4: colored *de Bruijn* graphs

Using and efficiently representing colored de Bruijn graphs. Another promising research direction is colored *de Bruijn* graphs, which is a new topic with several applications, especially in metagenomics and microbial pan-genomes. While some specific data structures for colored *de Bruijn* graphs have been recently developed [75,76], their use in assemblers is not yet fully explored. Still, they offer novel ideas that are quite promising.

Open problem 5: variable order *de Bruijn* graphs

Variable order de Bruijn graphs and string/overlap graphs in capturing variable length overlaps. The accuracy of the assembly results is another aspect to be taken into consideration. *De Bruijn* graph assemblers are used in the most popular short read assemblers, since they provide accurate assembly, and moreover, the computational costs have been constantly reduced in the most recent implementations. In the previous sections we have discussed novel data structures that lead to a theoretical improvement of space and time representation of *de Bruijn* graphs. Some of the previous discussed methods have already been implemented in a *de novo* assembler, while others still wait for an implementation and a practical analysis of their potential. Mainly, Minia [42] is a complete assembly tool, while DBGFM has been tested by integrating the *de Bruijn* construction phase into ABySS [77]. These implementations perform well in terms of space and time. However, it would be useful to investigate deeper whether those methods could produce good quality assemblies besides their theoretical results.

Notice that overlap graphs correspond to variable order *de Bruijn* graphs. Thus in order to fill the gap between the accuracy of *de Bruijn* graphs assemblers and overlap graph assemblers, it would be necessary to better understand the relationship between *de Bruijn* graphs of variable order and overlap graphs. A first step in this direction is the notion of A-Bruijn graph introduced by ABrujin [61] — such graph can be seen as a type of variable-order dBG based on a so-called set of solid strings.

Mainly, the question if variable order *de Bruijn* graphs have the same capability of string graphs in capturing variable length overlaps or overlap graphs contain more information than *de Bruijn* graphs, is a question that deserves to be properly investigated.

The recent results on the construction of the extended BWT via merging partial BWTs originally suggested in [78] have led also to efficient algorithms for merging *de Bruijn* graphs [52]. In this case, the main open problem is to investigate merging variable order *de Bruijn* graphs.

Open problem 6: large scale assembly with long reads

Using long reads in order to assemble large genomes. Concerning the assembly of long reads, current tools have been mainly used to assemble either single chromosomes with high precision (*e.g.*, Canu [54]) or small organisms — mainly bacteria and viruses. Still, it is an open problem the routine application of assembly tools for long reads to human genomes.

Open problem 7: building a pan-genome graph

Assembling a graph representing several genomes. Some recent papers have demonstrated how to efficiently construct the assembly graph for the whole genome sequence setting [79–81] for short reads. The fastest algorithm to date was able to process seven whole mammalian genomes in under eight hours [82]. This makes feasible to attack another problem: pan-genome assembly [83], that is the construction of the assembly graph of several individual genomes.

An interesting approach that can be a direction to follow is TwoPaCo [79]: a novel algorithm for constructing *de Bruijn* graphs from whole genome sequences. The authors of [79] demonstrate how to construct the graph for 100 human genomes in less then a day, and eight primates in less than two hours, on a typical server-grade machine.

COMPLIANCE WITH ETHICS GUIDELINES

This article is a review article and does not contain any studies with human or animal subjects performed by any of the authors.

REFERENCES

1. Tomescu, A.I., Medvedev, P. (2016) Safe and Complete Contig Assembly Via Omnitigs. In: Research in Computational Molecular Biology, Singh, M. (eds.), RECOMB 2016. Lecture Notes in Computer Science. vol 9649. Springer, Cham

2. Burrows, M. and Wheeler, D. J. (1994) A block-sorting lossless data compression algorithm. CiteSeer

3. Ferragina, P. and Manzini, G. (2005) Indexing compressed text. J. Assoc. Comput. Mach., 52, 552–581

4. Baker, M. (2012) *De novo* genome assembly: what every biologist should know. Nat. Methods, 9, 333–337

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

6. Paszkiewicz, K. and Studholme, D. J. (2010) *De novo* assembly of short sequence reads. Brief. Bioinform., 11, 457–472

7. Narzisi, G. and Mishra, B. (2011) Comparing *de novo* genome assembly: the long and short of it. PLoS One, 6, e19175

8. Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., *et al.* (2012) Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and *de-bruijn*-graph. Brief. Funct. Genomics, 11, 25–37

9. Jayakumar, V. and Sakakibara, Y. (2019) Comprehensive evaluation of non-hybrid genome assembly tools for third-generation PacBio long-read sequence data. Brief. Bioinform., 20, 866–876

10. Ghurye, J. and Pop, M. (2019) Modern technologies and algorithms for scaffolding assembled genomes. PLOS Comput. Biol., 15, e1006994

11. Shi, F. (1996) Suffix arrays for multiple strings: A method for on-line multiple string searches. In: Concurrency and Parallelism, Programming, Networking, and Security, Jaffar, J., Yap, R.H.C. (eds.), ASIAN 1996. Lecture Notes in Computer Science, vol 1179. Springer, Berlin, Heidelberg

12. Manber, U. and Myers, G. (1993) Suffix arrays: A new method for on-line string searches. SIAM J. Comput., 22, 935–948

13. Lam, T. W., Li, R., Tam, A., Wong, S., Wu, E. and Yiu, S. M. (2009) High throughput short read alignment via bi-directional BWT. In Bioinformatics and Biomedicine (BIBM '09), pp. 31–36, Washington, DC

14. Simpson, J. T. and Durbin, R. (2010) Efficient construction of an assembly string graph using the FM-index. Bioinformatics, 26, i367–i373

15. Simpson, J. T. and Durbin, R. (2012) Efficient *de novo* assembly of large genomes using compressed data structures. Genome Res., 22, 549–556

16. Li, H. and Durbin, R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics, 25, 1754–1760

17. Langmead, B., Trapnell, C., Pop, M. and Salzberg, S. L. (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol., 10, R25

18. Cox, A. J., Garofalo, F., Rosone, G. and Sciortino, M. (2016) Lightweight LCP construction for very large collections of strings. J. Discrete Algorithms, 37, 17–33

19. Li, H. (2014) Fast construction of FM-index for long sequence reads. Bioinformatics, 30, 3274–3275

20. Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M. and Rizzi, R. (2019) Multithread multistring Burrows-Wheeler transform and longest common prefix array. J. Comput. Biol., 26

21. Bloom, B. H. (1970) Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13, 422–426

22. Fan, L., Cao, P., Almeida, J. and Broder, A. Z. (2000) Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw., 8, 281–293

23. Broder, A. Z. (1997) On the resemblance and containment of documents. In Proceedings of Compression and Complexity of SEQUENCES 1997, pp.21–29. IEEE

24. Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M. and Yorke, J. A. (2004) Reducing storage requirements for biological sequence comparison. Bioinformatics, 20, 3363–3369

25. Salmela, L.and Rivals, E. (2014) LoRDEC: accurate and efficient long read error correction. Bioinformatics, 30, 3506–3514

26. Nordberg, H., Cantor, M., Dusheyko, S., Hua, S., Poliakov, A., Shabalov, I., Smirnova, T., Grigoriev, I. V. and Dubchak. I. (2013) The genome portal of the Department of Energy Joint Genome Institute: 2014 updates. Nucleic Acids Research, 42, D26–D31

27. Myers, E. W. (2005) The fragment assembly string graph. Bioinformatics, 21, i79–i85

28. Gonnella, G. and Kurtz, S. (2012) Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC Bioinformatics, 13, 82

29. Pevzner, P. A., Tang, H. and Waterman, M. S. (2001) An Eulerian path approach to DNA fragment assembly. Proc. Natl. Acad. Sci. USA, 98, 9748–9753

30. Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M. and Rizzi, R. (2016) LSG: An external-memory tool to compute string graphs for next-generation sequencing data assembly. J. Comput. Biol., 23, 137–149

31. Peng, Y. and Leung, H. C. M., Yiu, S. M. and Chin, F. Y. L. (2010) IDBA—a practical iterative *de Bruijn* graph *de novo* assembler. In Research in Computational Molecular Biology, Bonnie B., (ed.), pp. 426–440, Springer Berlin Heidelberg

32. Medvedev, P., Pham, S., Chaisson, M., Tesler, G. and Pevzner, P. (2011) Paired *de Bruijn* graphs: A novel approach for incorporating mate pair information into genome assemblers. In Proceedings of the 15th Annual International Conference on Research in Computational Molecular Biology, RECOMB'11, pp. 238–251, Springer-Verlag

33. Pham, S. K., Antipov, D., Sirotkin, A., Tesler, G., Pevzner, P. A. and Alekseyev, M. A. (2012) Pathset graphs: A novel approach for comprehensive utilization of paired reads in genome assembly. In Research in Computational Molecular Biology, Chor, B., (ed.), pp. 200–212, Springer Berlin Heidelberg

34. Ronen, R., Boucher, C., Chtsaz, H. and Pevzner, P. (2012) SEQuel: improving the accuracy of genome assemblies. Bioinformatics, 28,

i188–i196

35. Bao, E., Jiang, T., and Girke, T. (2014) AlignGraph: algorithm for secondary *de novo* genome assembly guided by closely related references. Bioinformatics, 30, i319–i328

36. Li, H. (2012) Exploring single-sample SNP and INDEL calling with whole-genome *de novo* assembly. Bioinformatics, 28, 1838–1844

37. Nong, G., Zhang, S. and Chan, W. H. (2011) Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput., 60, 1471–1484

38. Bauer, M., Cox, A. and Rosone, G. (2013) Lightweight algorithms for constructing and inverting the BWT of string collections. Theor. Comput. Sci., 483, 134–148

39. Bonizzoni, P., Della Vedova, G. , Pirola, Y., Previtali, M. and Rizzi, R. (2017) FSG: Fast String Graph construction for *de novo* assembly. J. Comput. Biol., 24, 953–968

40. Garey, M. R. and Johnson, D. S. (1979) Computer and Intractability: A Guide to the Theory of NP-completeness. Freeman, W. H., (ed.), San Francisco

41. Conway, T. C. and Bromage, A. J. (2011) Succinct data structures for assembling large genomes. Bioinformatics, 27, 479–486

42. Chikhi, R. and Rizk, G. (2013) Space-efficient and exact *de Bruijn* graph representation based on a Bloom filter. Algorithms Mol. Biol., 8, 22

43. Salikhov, K., Sacomoto, G. and Kucherov, G. (2014) Using cascading Bloom filters to improve the memory usage for *de Bruijn* graphs. Algorithms Mol. Biol., 9, 2

44. Jackman, S. D., Vandervalk, B. P., Mohamadi, H., Chu, J., Yeo, S., Hammond, S. A., Jahesh, G., Khan, H., Coombe, L., Warren, R. L., *et al.* (2017) ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. Genome Res., 27, 768–777

45. Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T. and Medvedev, P. (2015) On the representation of *de Bruijn* graphs. J. Comput. Biol., 22, 336–352

46. Bowe, A., Onodera, T., Sadakane, K. and Shibuya, T. (2012) Succinct *de Bruijn* graphs. In: Algorithms in Bioinformatics, volume 7534 of Lecture Notes in Computer Science, Raphael, B. and Tang, J. (eds.), pp. 225–235. Springer Berlin Heidelberg

47. Boucher, C., Bowe, A., Gagie, T., Puglisi, S. J. and Sadakane, K. (2015) Variable-order *de Bruijn* graphs. In Data Compression Conference (DCC), pp. 383–392

48. Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M. and Puglisi, S. J. (2018) Bidirectional variable-order *de Bruijn* graphs. Int. J. Found. Comput. Sci., 29, 1279–1295

49. Muggli, M. D., Bowe, A., Noyes, N. R.Morley, P.S., Belk, K. E., Raymond, R., Gagie, T., Puglisi, S. J., Boucher, C. (2017) Succinct colored *de Bruijn* graphs. Bioinformatics, 33, 3181–3187

50. Almodaresi, F., Pandey, P. and Patro, R. (2017) Rainbowfish: a succinct colored *de Bruijn* graph representation. In 17th International Workshop on Algorithms in Bioinformatics (WABI 2017).

51. Almodaresi, F., Sarkar, H., Srivastava, A. and Patro, R. (2018) A space and time-efficient index for the compacted colored *de Bruijn* graph. Bioinformatics, 34, i169–i177

52. Muggli, M. D., Alipanahi, B., Boucher, C. (2019) Building large updatable colored *de Bruijn* graphs via merging. bioRxiv, 229641

53. Neil, C.Jones,Pavel A Pevzner, and Pavel Pevzner. (2004) An introduction to bioinformatics algorithms. MIT Press

54. Koren, S., Walenz, B. P., Berlin, K., Miller, J. R., Bergman, N. H. and Phillippy, A. M. (2017) Canu: scalable and accurate long-read assembly via adaptive *k*-mer weighting and repeat separation. Genome Res., 27, 722–736

55. Berlin, K., Koren, S., Chin, C.-S., Drake, J. P., Landolin, J. M. and Phillippy, A. M. (2015) Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. Nat. Biotechnol., 33, 623–630

56. Chu, J., Mohamadi, H., Warren, R. L., Yang, C. and Birol, I. (2017) Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. Bioinformatics, 33, 1261–1270

57. Ruan, J. and Li, H. (2019) Fast and accurate long-read assembly with wtdbg2. bioRxiv, 530972

58. Kamath, G. M., Shomorony, I., Xia, F., Courtade, T. A. and Tse, D. N. (2017) HINGE: long-read assembly achieves optimal repeat resolution. Genome Res., 27, 747–756

59. Myers, G. (2014) Efficient local alignment discovery amongst noisy long reads. In International Workshop on Algorithms in Bioinformatics, pp. 52–67. Springer

60. Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B. P., Brownley, A., Johnson, J., Li, K., Mobarry, C. and Sutton, G. (2008) Aggressive assembly of pyrosequencing reads with mates. Bioinformatics, 24, 2818–2824

61. Lin, Y., Yuan, J., Kolmogorov, M., Shen, M. W., Chaisson, M. and Pevzner, P. A. (2016) Assembly of long error-prone reads using *de Bruijn* graphs. Proc. Natl. Acad. Sci. USA, 113, E8396–E8405

62. Prjibelski, A. D.,Vasilinetc, I., Bankevich, A., Gurevich, A., Krivosheeva, T., Nurk, S., Pham, S., Korobeynikov, A., Lapidus, A., and Pevzner. P.A. (2014) Exspander: a universal repeat resolver for DNA fragment assembly. Bioinformatics, 30, i293–i301

63. Chin, C.-S., Peluso, P., Sedlazeck, F. J., Nattestad, M., Concepcion, G. T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R., Morales-Cruz, A., *et al.* (2016) Phased diploid genome assembly with single-molecule real-time sequencing. Nat. Methods, 13, 1050–1054

64. Chin, C. S., Alexander, D. H., Marks, P., Klammer, A. A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E. E., *et al.* (2013) Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. Nat. Methods, 10, 563–569

65. Fasulo, D., Halpern, A., Dew, I., and Mobarry. C. (2002) Efficiently detecting polymorphisms during the fragment assembly process. Bioinformatics, 18, S294S302

66. Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., Remington, K. A., *et al.* (2000) A whole-genome assembly of Drosophila. Science, 287, 2196–2204

67. Berger, A. and Lafferty, J. (2017) Information retrieval as statistical translation. ACM SIGIR Forum, 51, 219–226

68. Li, H. (2016) Minimap and miniasm: fast mapping and *de novo*

assembly for noisy long sequences. Bioinformatics, 32, 2103–2110

69. Välimäki, N., Ladra, S. and Mäkinen, V. (2012) Approximate all-pairs suffix/prefix overlaps. Inf. Comput., 213, 49–58

70. Egidi, L., Louza, F. A., Manzini, G. and Telles, G. (2018) External memory BWT and LCP computation for sequence collections with applications. In 18th International Workshop on Algorithms in Bioinformatics, WABI 2018, Helsinki, Finland, volume 113, pp. 1–14

71. Liu, B., Liu, C.-M., Li, D., Li, Y., Ting, H.-F., Yiu, S. M., Luo, R. and Lam, T. W. (2016) BASE: a practical de novo assembler for large genomes using long NGS reads. BMC Genomics, 17, 499

72. Sohn, J. and Nam, J.-W. (2016) The present and future of de novo whole-genome assembly. Brief. Bioinform., bbw096

73. Simpson, J. T. (2014) Exploring genome characteristics and sequence quality without a reference. Bioinformatics, 30, 1228–1235

74. Garrison, E., Sirén, J., Novak, A. M., Hickey, G., Eizenga, J. M., Dawson, E. T., Jones, W., Garg, S., Markello, C., Lin, M. F., et al. (2018) Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat. Biotechnol., 36, 875–879

75. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P. and McVean, G. (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs. Nat. Genet., 44, 226–232

76. Pandey, P., Almodaresi, F., Bender, M. A., Ferdman, M., Johnson, R. and Patro, R. (2018) Mantis: a fast, small, and exact large-scale sequence-search index. Cell Syst., 7, 201–207.e4

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

78. Holt, J. and McMillan, L. (2014) Merging of multi-string BWTs with applications. Bioinformatics, 30, 3524–3531

79. Minkin, I., Pham, S. and Medvedev, P. (2017) TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. Bioinformatics, 33, 4024–4032

80. Marcus, S., Lee, H. and Schatz, M. C. (2014) SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics, 30, 3476–3483

81. Cazaux, B., Lecroq, T. and Rivals, E. (2014) From indexing data structures to de Bruijn graphs. In Combinatorial Pattern Matching, pp. 89–99. Springer

82. Baier, U., Beller, T. and Ohlebusch, E. (2016) Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics, 32, 497–504

83. Marschall, T., Marz, M., Abeel, T., Dijkstra, L., Dutilh, B. E., Ghaffaari, A., Kersey, P., Kloosterman, W., Makinen, V., Novak, A., et al. (2016) Computational pan-genomics: status, promises and challenges. Brief. Bioinform., 19, 118–135