Quantitative Biology 2020, 8(3): 203–215
https://doi.org/10.1007/s40484-020-0214-5

REVIEW

A survey on *de novo* assembly methods for single-molecular sequencing

Ying Chen, Chuan-Le Xiao*

State Key Laboratory of Ophthalmology, Zhongshan Ophthalmic Center, Sun Yat-Sen University, Guangzhou 510275, China
* Correspondence: xiaochuanle@126.com

Received March 10, 2020; Revised May 17, 2020; Accepted June 13, 2020

***Background*: The single-molecular sequencing (SMS) is under rapid development and generating increasingly long and accurate sequences. *De novo* assembly of genomes from SMS sequences is a critical step for many genomic studies. To scale well with the developing trends of SMS, many *de novo* assemblers for SMS have been released. These assembly workflows can be categorized into two different kinds: the correction-and-assembly strategy and the assembly-and-correction strategy, both of which are gaining more and more attentions.**

***Results*: In this article we make a discussion on the characteristics of errors in SMS sequences. We then review the currently widely applied *de novo* assemblers for SMS sequences. We also describe computational methods relevant to *de novo* assembly, including the alignment methods and the error correction methods. Benchmarks are provided to analyze their performance on different datasets and to provide use guides on applying the computation methods.**

***Conclusion*: We make a detailed review on the latest development of *de novo* assembly and some relevant algorithms for SMS, including their rationales, solutions and results. Besides, we provide use guides on the algorithms based on their benchmark results. Finally we conclude the review by giving some developing trends of third generation sequencing (TGS).**

Keywords: third generation sequencing; single-molecular real-time sequencing; sequence alignment; sequence error correction; *de novo* assembly

Author summary: In this review, we focus on the error characteristics of SMS sequences and challenges for *de novo* assembly of SMS sequences. We then describe the latest *de novo* assembly workflows, including both the correction-and-assembly and the assembly-and-correction assemblers. We also introduce some computation methods that are closely related to the *de novo* assembly, including sequence alignment and error correction methods. Benchmarks are provided to analyze their performance on different datasets and to provide use guides on applying the computation methods. We conclude the review by giving some developing trends of TGS.

INTRODUCTION

Third generation sequencing (TGS), also called single-molecule, real-time (SMRT) sequencing, including PacBio RS II platform developed by Pacific Biosciences (PacBio) and MinION platform developed by Oxford Nanopore, captures sequence information directory in the process of DNA molecule replication. TGS generates average read length in the order of 10 kb, which is much longer than next generation sequencing (NGS) read size (100–500 bp) and significantly improves the integrity and continuity of *de novo* assembly of genomes. Furthermore, the weak effect of classical sequencing bias, such as GC content, allows for resolving complex genome regions and the exploration of structural variants with an unprecedented accuracy and resolution. Many research results based on TGS have been published in different fields, such as *de novo* assembly, haplotype phasing, N6-methyladenine DNA modification, transcriptome and structural variation detection. TGS has promoted the studies of genomics in an impressive way. However, TGS technology has a serious drawback: its sequencing error

rate (11%–15% for PacBio raw reads, 5%–35% for Nanopore raw reads) is much higher than NGS (1%). High error rate and long read length bring great challenges to *de novo* assembly of TGS sequences. Due to the dramatic difference of data characteristics between TGS sequences and NGS sequences, computation methods designed for NGS sequences always do not work for TGS sequences. Furthermore, sequencing errors between PacBio sequences and Nanopore sequences are also remarkably different, which requires us to address them separately. Currently the widely used *de novo* assembly workflows developed for TGS sequences can be categorized into two strategies: "correction-and-assembly" and "assembly-and-correction". The "correction-and-assembly" strategy corrects the sequencing errors in TGS raw reads first, and then assembles genome using corrected reads. On the other hand, the "assembly-and-correction" strategy assembles genome using error-prone TGS raw reads, and then corrects the genome assembly with raw reads. Due to the computational intensive error correction process, the "correction-and-assembly" approach is usually slower. However, directly assembling genome using raw reads with high sequencing error rates may increase assembly errors in genome sequence, especially in genome containing repetitive regions, which then affects the quality of reference genome and causes results bias in downstream analysis. Practices have demonstrated that the "correction-and-assembly" strategy can reconstruct highly continuous and accurate genome assemblies.

In this review we first take a look at the sequencing errors of PacBio sequences and Oxford Nanopore sequences respectively, and then introduce the currently widely used "correction-and-assembly*"* and "assembly-and-correction" *de novo* assembly workflows and several relevant methods, such as alignment methods and error correction methods designed for TGS sequences. We summarize all these computation methods in Table 1. To analyze the performance of the workflows, we give two benchmarks and talk about their assembly results on each dataset. Based on their performance on different datasets, we then list some use guides on applying these computation methods. Finally, we talk about the development prospects of sequencing technologies and conclude this paper.

CHARACTERISTICS OF SMS SEQUENCES

Initially the sequence (also called continuous long read, CLR) generated by the PacBio RS system with the first generation of chemistry (C1 chemistry) produces average lengths only around 1500 bp [1]. Up to now, the PacBio RS II system with the sixth generation of polymerase and the fourth generation of chemistry (P6-C4 chemistry) has improved average read lengths over 10 kb. The maximum read lengths can exceed 60 kb and the N50 can reach up to 20 kb [2]. Error rates of PacBio raw reads are 11%–15%. The large majority of sequencing errors are insertions and deletions (indels).

Due to the random distribution of errors in CLRs, a sufficient sequencing pass yields a circular consensus sequence (CCS) having significantly reduced error rate [3]. For example, an accuracy $> 99\%$ can be achieved with a coverage of 15 passes. However, considering that the total length of a CLR is limited by the life time of the polymerase [4], there exists a tradeoff between the number of sequencing passes and the CCS read length. Recently, the CCS has been optimized to improve the accuracy and length of single-molecule real-time sequencing [5]. The average sequence length has been increased to 13.5 kb and the accuracy is as high as 99.8%. This improvement has been successfully used for variant detection and *de novo* assembly of human genomes.

The lack of bias and the random nature in sequencing errors [6] make it possible and relatively easy to correct PacBio reads with high accuracy. Heuristic gapped alignment methods are able to effectively find overlaps between the template (the raw read to be corrected) and its supporting reads (reads sampled from the same regions of the genome with the template) having difference less than 30%. Based on these overlaps, the template can be corrected using some consensus algorithm. The accuracy of corrected reads can be as high as 97%–99%.

The MinION sequencer developed by Oxford Nanopore Technologies (ONT) generates long reads base on the transit of a DNA molecule through a nanoscopic pore which has been thought of as one of the most promising approaches to detect polymeric molecules since the 90s [7]. Over the past years, ONT released several different MinION chemistry versions (R6.0, R7.0, R7.3, R9 and R9.4) and several updates of the base-calling tools that strongly improved the performance of the MinON device. While the earliest R6.0 chemistry version allowed to generate a total yield in the order of tens of Mb and 2D reads with 4 kb of median length and 40% of the average error rate, the more recent R7.0 and R7.3 improved the throughput to hundreds of Mb and 2D read size and accuracy to 6 kb and 80%–85%, respectively [7]. In October 2016, new flow cells containing R9.4 1D chemistry were released by ONT, which was used to generate the first nanopore high-coverage (30×) human genomes with about 40 flow cells [8]. At the same time, a new protocol was also developed to generate ultra-long reads (5×) whose N50 was longer than 100 kb and the maximum read length can reach up to 882 kb. *De novo* assembly of this 35× raw reads data yielded a contiguous assembly with NG50 as long as 6.4 Mb [8].

Compared to PacBio sequences, errors in Nanopore

reads are much more complex [9,10]. First, error rates of Nanopore reads are much higher and broadly distributed among reads and among subsequences of a read, ranging from 5%–35%. Second, there commonly exist subsequences with extreme high error rates (50%) in Nanopore reads [9]. This means that not only the error rates of two different reads, but also two different subsequences sampled from the same read, can differ greatly. These characteristics of errors make it much more difficult to align, correct and *de novo* assemble Nanopore reads. Furthermore, due to the higher error rates, average identity of reads corrected from Nanopore reads are generally lower than those corrected from PacBio raw reads. For example, the corrected reads of the 35× human raw sequences [8] mentioned above average only 92% identity to the reference; while reads corrected from PacBio raw reads are typically > 99% identical to the reference genome. The decreased accuracy of Nanopore corrected sequences introduces troubles for accurately assembling genomes, especially for complex genomes.

CORRECTION-AND-ASSEMBLY METHODS

The "correction-and-assembly" methods achieve high consensus accurate reads by correcting the noisy raw reads and then assemble genomes with the corrected reads. The assembly process generally consists of four stages. (i) Perform a pairwise mapping on the raw reads to find supporting reads for each template. (ii) Align the supporting reads to the template and correct the template based on these overlaps. (iii) Perform a pairwise mapping on the corrected reads and assemble the genome based on the mapping results. (iv) Polish the genome with various data sources, such as the raw reads and NGS sequences using polishing algorithms such as Nanopolish [11], Pilon [12], smrtlink [13] and Racon [14] to further improve the accuracy of the genome. It is noted that these polishing methods are also used for correcting errors in genomes assembled by the "assembly-and-correction" approaches.

On the early stage, intuitive and automated hybrid workflows are developed for constructing finished and high-quality small bacterial genomes (1–10 Mb) [15–17]. In these hybrid methods, the accurate short reads (from SMRT circular consensus sequencing reads [5] or NGS reads) are used to correct errors in the long SMRT sequencing reads. These methods have been applied successfully to a variety of microbes and eukaryotic organisms. They generally involve at least two different sequencing libraries and at least two types of sequencing runs (and sometimes different sequencing platforms). For more efficient genome assembly, especially for large and complex genomes, a homogeneous workflow that requires only one library and sequencing platform is required to simplify the assembly process and to reduce the sequencing costs. Another shortcoming of the hybrid approach is that the constructed genomes always break in regions with insufficient NGS data coverage (owing to the GC or sequence context biases) [18].

Currently the monohybrid methods that rely solely on TGS data have become the mainstream assembly approaches. These methods avoid the drawbacks in hybrid approaches and have been successfully applied for constructing large genomes (such as the human genome).

Both the hybrid and monohybrid assembly methods involve sequence alignment and error correction as intermediate steps. The sequence alignment for TGS reads is computationally intensive and still remains the most time-consuming step in genome assemblies. In this section, we will first talk about the existing alignment algorithms developed for TGS reads (see the first category of Table 1). We then proceed to introduce the consensus methods that are widely used in the mainstream assembly workflows (see the second category of Table 1). Finally we describe the assembly workflows that have been extensively used for both PacBio sequences and Nanopore sequences.

Alignment methods for TGS sequences

Strictly speaking, the existing alignment methods designed for TGS sequences are all variants of the standard Smith-Waterman algorithm [19]. They employ various heuristics based on one or more characteristics of TGS reads to reduce the searching space and decrease memory usage of the original Smith-Waterman algorithm. Many of them follow BLAST's "seed-and-extension" searching paradigm [20].

BLASR

BLASR (basic local alignment with successive refinement) [21] uses a successive refinement approach to map SMRT sequences to the reference genome in which case the divergence between them are dominated by insertion and deletion sequencing errors. Its core idea is to narrow down the search space from the whole genome positions to a relatively small number of candidate genome intervals where the read are likely to map. To further validate which candidate interval is mostly likely to be mapped, a detailed alignment is performed on each candidate interval and the best one is chosen as its searching results.

BLASR takes a read $r = r_1 r_2 \cdots r_R$ and a genome $g = g_1 g_2 \cdots g_G$ as inputs. The searching process consists of three phases. Step 1: Detecting candidate intervals by clustering short and exact matches. BLASR builds either

Table 1 Computation methods for Third Generation Sequencing

	Methods	Target sequencing platform	Assembly workflow
Alignment methods	BLASR	PacBio	FALCON
	DALIGN	PacBio	FALCON
	Minimap2	PacBio, ONT	
	edlib	PacBio, ONT	Canu
Consensus methods	DAGCon	PacBio	PBcR, MECAT
	FALCON-sense	PacBio, ONT	FALCON, Canu
Correction-and-assembly workflow	FALCON	PacBio	
	Canu	PacBio, ONT	
	MECAT	PacBio	
Assembly-and-correction workflow	Flye	PacBio, ONT	
	Wtdbg2	PacBio, ONT	

a suffix array (SA) or BWT-FM index [22] on the genome according to the time and space requirements, and queries for exact matches of subsequences (of length at least K) between r and g. The BWT-FM index has been used substantially for aligning NGS short reads [23,24] and accelerating the traditional BLAST search [25,26]. It is a very flexible data structure that allows for searching for fixed length matches, maximal exact matches, and matches in which mismatch bases are allowed efficiently. The set of matches (also called anchors) is denoted as A. The anchors in A are then clustered with global chaining [27]. To do so, the anchors are sorted by genome and query positions and the clusters of anchors are found in intervals that roughly the length of the read. For each anchor $a_i \in A$, a set $A_i := \{a_j \in A : 0 \leqslant G(a_j) + l(a_j) \leqslant R\}$ is created, where $G(a_j)$ and $l(a_j)$ are the genome position and length of the anchor a_j respectively. In each A_i, the global chaining algorithm [27] is used for finding a maximal subset of anchors $C_i \subseteq A_i$ that are not overlapping and listed in increasing read and genome positions. For each cluster C_i BLASR assigns it with a frequency weighted score $\Sigma_{a_{j\in} C_i} \log\left(\frac{1}{\text{Freq}(a_j)}\right)$, where $\text{Freq}(a_j)$ is the frequency of subsequence a_j in the genome. After scoring only the top MAXCANDIDATES (10 by default) clusters are retained. Step 2: Mapping r to the candidate intervals using a sparse dynamic programming (SDP) [28]. For candidate interval C_i, let a^f and a^l be the anchors in C_i that have the minimum and maximum genome positions respectively. Suppose the maximum insertion rate of the sequencing device is δ, then the minimal starting genome position of the interval is $s = G(a^f) - (1+\delta)R(a^f)$, and the maximal ending genome position is $f = G(a^l) + (1+\delta)(R - (R(a^l) + l(a^l)))$, here $R(a)$ is the read position of anchor a. The length of this interval is $l_c = f - s$. Similar to step 1, a set of matches are found between r and the candidate interval. The matches used here are fixed on short length K^{SDP} and denoted as set A^{SDP}. The sparse dynamic programming finds the largest subset of anchors $C^{\text{SDP}} \subseteq A^{\text{SDP}}$ that are of increasing read and genome positions. Step 3: The sparse dynamic programming alignment is used as a guide for performing a detailed banded alignment. The SDP in Step 2 only aligns some subsequences in r from C^{SDP} to the candidate interval. To align every base of r to the interval, BLASR triggers a banded dynamic programming. Due to the large indel rate of SMRT sequences, the size of the band used to contain the entire alignment can be prohibitively large. To reduce the searching space, BLASR uses C^{SDP} as a guide where the band follows the layout of the anchors in C^{SDP}. After this step, the read base positions are assigned to reference positions. In BLASR, the authors also provide detailed theory analysis to support their clustering and to compute the mapping quality values.

DALIGN

DALIGN [29] proposes a sensitive and threaded filter for detecting seed points that are likely to have a significant local alignment passing through them and a linear expected time heuristic local alignment algorithm. It is developed for detecting read-to-read overlaps amongst SMRT noisy long reads. Specifically, given two SMRT long noisy read blocks denoted as A and B respectively, DALIGN aims to find local alignments between reads that are sufficiently long and stringent.

The first step of DALIGN is rapid seed detection with a threaded filter. Unlike BLASR, which employs Suffix Array or BWT-FM index to search for variable-length anchors, DALIGN uses a much simpler series of highly optimized sorts to discover k-mer matches in a given diagonal band between two reads. The filter proceeds as follows:

1. Build a k-mer list $\text{List}_A := \{(kmer(A^a, i), a, i)\}_{a,i}$ for block A, where $kmer(R, i)$ is the subsequence $R[i-k+1, i]$.
2. Similarly build a k-mer list List_B for block B.
3. Sort both List_A and List_B in order of hash values of their k-mers.
4. Build a k-mer match list $\text{List}_M := \{(a, b, i, j) : kmer(A^a, i) = kmer(B^b, j)\}$ by a merge sweep of the two sorted k-mer lists.
5. Sort List_M lexicographically on a, b, and i.

Steps 1, 2 and 4 are easily seen to take linear time and space. The most time-consuming step is the sorting process in Steps 3 and 5, which in theory takes $O(L\log L)$ time, where L is the list size. DALIGN optimizes the encoding of the lists in Steps 3 and 5 by squeezing their elements into 64-bit integers. The problem of sorting lists in Steps 3 and 5 now becomes realizing a threadable, memory coherent sort of an array $src[0, ..., N-1]$ of N 64-bit integers. DALIGN implements a radix sort [30] where each 64-bit integer is considered as a vector of $P = 64/B$, B-bit digits $(x_p, x_{p-1}, ..., x_1)$ and B is a free parameter whose typical value is 8. The radix sort then sorts the numbers by stably sorting the array on the first B-bit digit x_1, then on the second x_2, and so on to x_p in P sorting passes. The sorting process asymptotically takes $O(P(N+2^B))$ time. As B and P are fixed small constants the algorithm is effectively $O(N)$ in time complexity.

The second step of DALIGN is rapid local alignment from a seed $\rho = (i, j)$ reported by the filter in Step 1, where i and j are the positions of the k-mer in two reads. The original $O(nd)$ algorithm [31] centers on the idea of computing progressive "waves" of furthest reaching (f.r.) points. Starting from $\rho = (i, j)$ on diagonal $k = i-j$, the goal is to find the longest possible paths starting at k, first with 0-differences, then with 1-differences, 2-differences, and so on. The problem is that the waves become wider and wider as we compute f.r. waves away from diagonal k in each direction. To narrow down the waves, DALIGN uses several strategies to trim the span of a wave by removing the f.r. points that are extremely unlikely to be in the desired local alignment. The key idea is that a desired local alignment should not cover any reasonable segment having an exceedingly low correlation. This is also the key idea behind the X-drop criteria in many heuristic alignment tools such as BLAST. In DALIGN, the first principle for trimming a wave is to remove f.r. points for which the last C columns of the alignment have less than M matches. The second trimming principle is to keep only f.r. points which are within L antidiagonals of the maximal anti-diagonal reached by its wave. The computation of successive waves eventually ends because either (a) the boundary of either sequence is reached, or (b) all the f.r. points fail to meet the regional alignment quality criteria in which case it is assumed that the two reads no longer correlate with each other.

Experiments show that DALIGN is much more sensitive and can be 22 to 39 times faster than BLASR on the PacBio human dataset. Furthermore, it takes only 15,600 core hours for overlapping the 54X PacBio human dataset. Compared to the 404,000 core hours using BLASR when running on the Google "Exacycle" platform, this represents a substantial 25X reduction in computation time.

Minimap and minimap2

Minimap is an alignment free mapping approach for TGS sequences. In addition to pairwise overlapping, minimap also supports read-to-genome and genome-to-genome mapping. The most distinctive features in minimap is that it takes minimizer matches as seeds and skips the detailed alignment step so that it can be very fast compared to other mapping tools.

Minimizers [32] are proposed as a replacement of k-mers for reducing storage requirements and not reducing the searching sensitivity at the same time. Minimap maps a k-mer into a 64-bit integer hash values in three steps. First, choose a hash function ϕ such that $\phi(A) = 0$, $\phi(C) = 1$, $\phi(G) = 2$ and $\phi(T) = 3$. Next, for a k-mer $s = a_1 a_2 \cdots a_k$, define

$$\phi(s) := \phi(a_1) \times 4^{k-1} + \phi(a_2) \times 4^{k-2} + \cdots + \phi(a_k).$$

Finally, the function $\phi' := b \circ \phi$ is used as the hash function that transfers each k-mer to a 64-bit integer hash value. Here b (see Algorithm 2 of [33]) is an invertible integer hash function such that the complex k-mer would have the small hash values while the repetitive k-mers such as a long run of A's have large hash values. A (w,k)-minimizer of a string is the k-mer having smallest hash values in a surrounding window of w consecutive k-mers. Note that there can be more than one minimizer in a window of w consecutive k-mers.

Minimap works in two steps: indexing and mapping. In the indexing stage, it collects minimizers in all target sequences, sorts the minimizers by their hash values using radix sort introduced in DALIGN [29], and builds a hash table for them. Minimap only samples roughly $1/w$ of the total k-mers (note that a minimizer is the k-mer having smallest hash value among w consecutive k-mers) so that its storage requirement is much less than that in DALIGN. Minimap samples the most informative k-mers so that the sensitivity will not reduce significantly. In the mapping stage, minimap collects all the minimizer hits between the query and all the target sequences. A single-linkage clustering is performed to detect the maximal collinear subset of hits by solving a longest increasing sequencing problem (Algorithm 4 of [33]). This subset is output as the final mapping result.

Minimap has been optimized and improved into minimap2 [34] by adopting a more accurate chaining method and supporting recovering base-level alignments by performing DP-based global alignments between adjacent anchors in a chain. Minimap2 supports aligning many more types of sequences, such as accurate short reads, SMRT long noisy reads, full-length noisy Direct RNA or cDNA reads, as well as assembly contigs or closely related full chromosomes of hundreds of megabases in length.

To chain the minimizer hits (anchors), minimap2 sorts them by ending reference position and let $f(i)$ be the maximal chaining score up to the i-th anchor in the list, which is calculated with a dynamic programming:

$$f(i) := \max\{\max_{i>j\geqslant 1}\{f(i)+\alpha(j,i)-\beta(j,i)\}, w_i\},$$

where $\alpha(j,i) := \min\{\min\{y_i-y_j, x_i-x_j\}, w_i\}$ is the number of matching bases between the two anchors and $\beta(j,i)$ is the gap cost that defined by

$$\beta(j,i) := \begin{cases} \infty, \text{ if } y_j > y_i \text{ or } \max\{y_i-y_j, x_i-x_j\} > G \\ \gamma_c((y_i-y_j)-(x_i-x_j)), \text{ otherwise} \end{cases}.$$

In implementation, a gap of length $l>0$ costs $\gamma_c(l) := 0.01\cdot\overline{w}\cdot l + 0.5\cdot\log_2 l$, where $\overline{w}$ is the average anchor length. A direct computation of all the $f(i)$'s obviously takes $O(N^2)$ time. Minimap2 reduces this time to $O(N)$ by introducing a heuristic.

To recover detailed alignments, minimap2 performs DP-based global alignments between adjacent anchors in each chain with a 2-piece affine gap cost [35], which helps to recover longer indels. To further accelerate the global alignment, a SSE implementation based on a difference-based formulation developed in [36] is used here.

Edlib

Edlib [37] is developed for efficiently performing an important category of sequence alignments: the calculation of Levenshtein distance (also called edit distance) which is the minimum number of single-character edits (insertion, deletion or substitution). The calculation of Levenshtein distance is equivalent to the calculation of maximum alignment score using standard dynamic programming under the unit cost scoring system. Edlib [37] in principle is an extended implementation of the original bit-vector dynamic programming algorithm [38]. The original bit-vector algorithm accelerates the dynamic programming alignment algorithm which computes the minimal scoring matrix edit distance of two sequences by packing multiple cells as a bit-vector into one CPU register and thus enables the computation the values of multiple cells at a time. It only supports semi-global alignment where gaps at the start and end of the query sequence are not penalized. Edlib extends the bit-vector algorithm to support both global alignments and semi-global alignments in which gaps at the end of the query are not penalized by the extended banded algorithm which supports all three kinds of alignments.

Another deficiency of the original bit-vector algorithm is that it returns no traceback information (a sequence of operations including substitution, insertion and deletion that performed on one sequence to transform it into the other sequence). The traceback is critical to many sequence analysis such as sequencing error correction and variant detection. Edlib remedies this defect by finding the optimal alignment path for all three supported alignments in linear space with Hirschberg's divide-and-conquer strategy [39].

The consensus algorithms for TGS sequences

The correction of noisy TGS long reads is critical to many genomic studies with high resolution, such as structural variation detection. The currently widely used monohybrid correction methods work by aligning supporting reads to the template sequence. The overlaps are then used to build a multiple sequence alignment, from which a consensus sequence is resolved with some scoring scheme.

DAGCon

DAGCon is developed as part of the hierarchical genome assembly workflow HGAP [6] to correct seeding sequences. It draws the idea from [40] in which multiple sequence alignment is represented as a directed acyclic graph.

DAGCon constructs a directed acyclic graph which is denoted as the sequence alignment graph (SAG) with alignments between supporting reads and the template in five steps. Step 1: Normalize mismatches and degenerated alignment positions in each alignment. It is known that mismatches in one alignment are indistinguishable from an adjacent insertion-deletion pair to the correction algorithm. Furthermore, given a specific gap cost function, there are cases where there is more than one way to place gaps in an alignment. To reduce such potential inconsistency introduced by the two cases mentioned above and to improve the final consensus accuracy, each mismatch is replaced by a insertion-deletion pair and every gap is moved to the right most equivalent position (see Supplementary Fig. S1 in [6] for an example). Step 2: Setup an initial directed acyclic graph from the template. Specifically, an initial backbone graph consists of the nodes v_i with label$(v_i) = R_i$, where R_i is the residue of the template at position i, and the directed

edges $e_{i,i+1} := v_i \rightarrow v_{i+1}$ is built in this step. Step 3: Construct a multigraph from the alignments and merge the edges that are connecting the same nodes. Each alignment is represented as a list $A = \{(q_i, r_i) : q_i, r_i \in A, C, G, T, -\}$ and added to the backbone graph (Algorithm 1 of [6]). Step 4: Merge nodes. To reduce the complexity of the backbone graph, DAGCon merges those nodes that are similar to each other (for example, the nodes having the same label are connected to the same out-node or are connected from the same in-node) while preserving the directed acyclic property of the graph at the same time (Algorithm 4 of [6]). Step 5: Generate consensus sequence from the simplified backbone graph. Each node in the graph is given a score by checking the weight of its out-edges. Each path in the graph is assigned a score by summing up the scores of its nodes. The consensus is constructed by finding the path having the highest path score among all the possible paths with a simple dynamic programming (Algorithms 5 and 6 of [6]).

FALCON-sense

FALCON-sense is the consensus algorithm that is released together with the FALCON assembler [41]. FALCON-sense starts with performing a pairwise mapping with DALIGN [29] on the noisy raw reads. Each template is then processed as follows. First, a variation of the $O(nd)$ algorithm [31] is applied to align the supporting reads to the template where all the differences in the alignments are encoded as insertions and deletions only. For each position in every alignment, an "alignment tag" that encodes both the positions and the bases in the supporting read and the template is created. These tags are then used for constructing an alignment graph in the following way. For each tag, if it does not exist in the graph, a corresponding node is added to the graph. For consecutive tags from an alignment, an edge that is created with *edge Count* equaling to 1 is added to the graph. If the edge already exists in the graph, its corresponding *edge Count* is increased by 1. After processing all the alignment tags we obtain a directed acyclic graph. Every node in the graph has a tag. The *edge Count* in each edge indicates how many reads support the connection between the two nodes. Finally a standard dynamic programming algorithm is applied on the alignment graph to find the highest confidence path. By concatenating the bases of the tags on the best path the consensus sequence is generated. Supplementary Fig. 12 of [41] illustrates this consensus process.

In implementation, FALCON-sense is much simpler, more robust and more efficient than the DAGCon algorithm. It is also integrated in another widely used *de novo* assembly workflow Canu as the error correction method.

The correction-and-assembly workflows

There are three widely used "correction-and-assembly" assemblers, which are described below.

FALCON

The FALCON assembler [41] follows the same workflow of the hierarchical genome assembly pipeline HGAP [18] with many computationally optimized implementations. The most distinctive feature of FALCON is that it also supports phased diploid genome assembly with the FALCON-Unzip algorithm.

The FALCON workflow starts with error correction using the FALCON-sense algorithm (Section "FALCON-sense"). After that the DALIGN [29] algorithm is used for identifying overlaps between all pairs of corrected reads. These overlaps are used to construct a directed string graph [42], which is further simplified with assembly string graph reduction (Supplementary Fig. 14 of [41]) into the final "haplotype-fused assembly graph". After the reduction process, the primary and associated contigs are constructed based on the corrected reads and their overlap relationships. For each primary contig, FALCON collects all the raw reads associated with it and with its associated contigs according to the overlapping information. The raw reads are then aligned to the contigs with BLASR [21] for phasing heterozygous SNPs and identifying the haplotype of each raw read (Supplementary Fig. 15 of [41]).

To construct haplotype-specific contigs (haplotigs), the FALCON-Unzip algorithm constructs a haplotype-specific assembly graph for each contig from its corresponding raw reads and then combines this graph to the fused assembly subgraph that contains the paths of this contig to construct a complete subgraph, which has complete read representation from both haplotypes. See Fig. 1 and Supplementary Fig. 13 of [41] for details. As a result, FALCON-Unzip build a new primary contig and several haplotigs from the original assembly graph of that contig. It assigns each phased read to the primary contig or one of the haploptigs according to the phasing information.

Canu

The PBcR assembly workflow [43] is developed for effectively *de novo* assembling large genomes such as the human genome. It proposes MHAP for overlapping noisy long reads using probabilistic and locality-sensitive hashing. The DAGCon [6] and FalconSense are used

for correcting the errors in the long reads. And the Celera Assembler is used for assembling the corrected long reads.

To reduce the running time of the overlapping step in the assembly pipeline, MHAP (MinHash Alignment Process) uses a dimensionality reduction technique named MinHash sketches for efficient alignment filtering. The MinHash sketch of a read is created by extracting all the k-mers and converting them to fingerprints by H hash functions. As a result, we obtain H hash sets $\Gamma_1, ..., \Gamma_H$. In MHAP, after the first hash set Γ_1, subsequent fingerprints are generated using XOR shift pseudo-random number generators $\Gamma_2, ..., \Gamma_H$. The k-mer that generates the minimum value for each hash is referred to as the min-mer for that hash. The sketch of that sequence is then composed of the ordered set of its H min-mer finerprints that is much smaller than the set of all of the read's k-mers. To find overlaps between two reads, MHAP builds two sketches for them and estimates their Jaccard similarity by computing the fraction of min-mers common to both sketches. If these two sketches are sufficiently similar, the shared min-mers are located in two reads and the median difference in positions is computed to determine the overlap offsets. See Fig. 1 of [43] for an example. The concept of sketch presented here and the concept of minimizer used in minimap and minimap2 share the same idea. They are both reduced representations of the original sequence that are used for decreasing memory usage and accelerating the overlapping process.

The PBcR is later optimized into a more efficient assembler named Canu [44]. Canu implements an updated version of MHAP which uses a two-stage overlap filter, where the first stage identifies read pairs that are likely to share an overlap and the second stage estimates the extent and quality of the overlap. For the first step, a *tf-idf* (term frequency, inverse document frequency) weighting is implemented in MHAP to increase sensitivity of true overlaps and reduce the number of false and repetitive overlaps. For a raw read set Q, let $f_{\max}$ and $f_{\min}$ be the maximum and minimum observed frequency of all the k-mers in Q respectively, and f_q be the frequency of a specific k-mer q in Q. The inverse document frequency function for q, denoted as idf_q, is defined as

$$idf_q := T\left(\log\left(\frac{f_{\max}}{f_q} - a\right)\right).$$

The parameter $a \in [0,1]$ controls how strongly less common k-mers are preferred in relation to the more common ones, and T is a constant used for linearly rescaling the idf value to fall in $[1, f_{\max}]$. A k-mer not existing in Q will be given $idf_{\max}$ as its idf value. The $tf-idf$ value of q is then computed by the formulation $w_q := tf_q \cdot idf_q$, where tf_q is the number of occurrences of q in Q. After computing the *tf-idf* weight w_q for each k-mer q, the original MinHash computation is now replaced by applying w_q hash functions $\Gamma_{i,1}, ..., \Gamma_{i,w_q}$ per entry rather than the single Γ_i. For each sketch entry S_i, the min-mer is chosen as the minimum hash value computed across all the hash functions. Since highly weighted k-mers are hashed more times, this increases the chance that they will be chosen as a min-mer.

In the correction stage, Canu uses two filtering steps to avoid the introduction of overlaps from copy-specific repeat variants. The first is a global filter where each read chooses where it will supply correction evidence. The second is a local filter where each read accepts or rejects the evidence supplied by other reads. The FALCON-sense consensus algorithm [41] is used for generating the corrected reads. Canu also reduces the runtime by an order of magnitude of the Celera Assembler. In the assembly stage, Canu uses a variant of the greedy "best overlap graph" algorithm [45] for constructing a sparse overlap graph. The greedy algorithm loads only the "best" (longest) overlaps for each read end into memory and thus consumes much less memory compared to the string graph formulation [42], which requires loading the full overlap graph into memory. Furthermore, to improve repeat and haplotype separation, Canu introduces a new "Bogart" algorithm to statistically filter repeat-induced overlaps and retrospectively inspect the graph for potential errors.

MECAT

MECAT [46] is developed to effectively distinguish true overlaps and random read pairs that share similar subsequences. Traditional aligners such as BLAST [20] employ the k-mer match methods to filer out random read pairs and quickly find seeding positions for local gapped alignments. However, the repetitive subsequences in reads always find a large number of k-mer matches and results in excessive candidate seeding positions. On the other hand, simply masking these low-complexity regions, or ignoring the k-mers with high frequencies can cause loss of correct overlaps. To address this issue, MECAT proposes a pseudolinear alignment distance difference factors scoring (DDFs) algorithm to quickly filter excessive alignments.

For two k-mer matches whose positions are $s_1 = (q_1, r_1)$ and $s_2 = (q_2, r_2)$ respectively, where q_i is the read position and r_i is the reference position, their distance difference factor is defined as

$$\mathrm{DDF}(s_1, s_2) := |1 - \frac{q_1 - q_2}{r_1 - r_2}|.$$

If $\mathrm{DDF}(s_1, s_2) < \varepsilon$, then this two k-mer matches are said to support each other in positions. Now suppose the

k-mer matches between a read and a target sequence are s_1, s_2, ..., s_n. The DDF score of each match is initialized with zero. For every match pair s_i and s_j, if $\mathrm{DDF}(s_i, s_j) < \varepsilon$, we increase the DDF scores of both matches by one. The k-mer having highest DDF score provides a position to which a significant local alignment is likely to pass through. It is evidently that the DDF scoring process takes $O(n^2)$ time, which is very slow for long SMRT reads. To speed up the scoring process, MECAT breaks it into two steps so that the quadratic mutual scoring only takes place in local regions. In the first step the database is decomposed into consecutive blocks. Each block has the same length of B whose default value is 2000. The scoring starts with randomly selecting a block that having at least b k-mer matches. The mutual DDF scoring is performed on the matches into that block. If the match with the highest score is significant (greater than the threshold), we set it as the seed position. In the second step, the scoring process is extended to its neighbor blocks. For each neighbor block, we calculate the DDF between its k-mer matches and the seed k-mer match. If $\mathrm{DDF} < \varepsilon$, we increase the score of the seed k-mer match by 1. The mutual scoring in Step 1 is conducted in $O(N^2)$ time, and the extension scoring in Step 2 is conducted in $O(N)$ time, where N is the number of k-mer matches. Since the number of k-mer matches in mutual scoring is small, the overall scoring process can be finished in pseudolinear time.

Experiments show that the scores of the seeding positions grow linearly with their alignment lengths. After DDF scoring filtering, the candidate seeding positions are reduced by 50%–70% before extending the seeding positions to local alignments. In the correction stage, MECAT employs DDF scoring to effectively choose supporting reads for each template and then uses an adaptive DAGCon consensus algorithm [18] to correct the templates. Originally the DAGCon algorithm achieves higher precision compared to other correction methods and builds a sequence alignment graph (SAG) for the whole template, which consumes a lot of time and memory. MECAT accelerates the consensus process in two steps. First it identifies bases in the template that are covered mainly by the same bases or gaps from supporting reads. In the corrected read, these bases will remain unchanged if they are covered mainly by the same bases, or will be deleted if they are mainly covered by gaps. Second, the DAGCon algorithm is used for correcting the subsequences between two unchanged bases. MECAT originally uses the assembly module from Canu [44], which is now replaced by fsa, a string graph [42] based assembler.

ASSEMBLY-AND-CORRECTION METHODS

Flye

Flye [47] is developed for accurately resolving genomic repeats in *de novo* genome assembly. Unlike other mainstream assemblers, Flye starts with creating a disjointigs that represent concatenations of multiple disjoint genomic segments and concatenating all error-prone disjointtigs into a single string. From these concatenate results, Flye creates an accurate assembly graph, untangles it with reads and resolves the bridged repeats. Finally, Flye resolves the unbridged repeats with the repeat graph using small differences between repeat copies and outputs accurate contigs formed by paths in the graph.

Wtdbg2

Wtdbg2 [48] proposes a fast pairwise mapping implementation and a layout algorithm based on fuzzy-Bruijn graph (FBG) to speed up the large genome assembly process. To reduce memory usage, the current aligners (such as DALIGN and minimap2) split the input raw reads into small batches and perform pairwise mapping between batches. Wtdbg2 loads all the raw reads into memory and builds a hash table for the k-mers in all raw reads. For two reads that share k-mers, wtdbg2 performs a Smith-Waterman-like dynamic programming between these two reads in terms of bins. Wtdbg2 encodes each read into a consecutive sequence of bins. Each bin consists of 256 bp. The FBG proposed by wtdbg2 extends the basic ideas from *de Bruijn* graph. Each base in FBG is a 256 bp bin and a k-bin in FBG consists of k consecutive bins from reads. After simplifying the FBG, wtdbg2 builds the final consensus with partial order alignment over edge sequences.

SOME BENCHMARKS

To illustrate the efficiency and assembly quality of the assembly workflows described in this review, we present some benchmarks in this section. The benchmark results would also provide clues for use guides to be talked about in the next section. It is noted that there exits an abundance of benchmarks of this sort reported in various references. Therefore we are not going to carry out such tests ourselves but instead of quoting two representative reports and listing the results here for easy reference and discussion.

Benchmarks on PacBio data

We list a simplified version of experimental results reported in [48] (Table 2). The five workflows are used for *de novo* assembling three moderate genomes (100 M) and one large human genome (3 G). More details about the datasets and the experimental setup are found in Table 1 and Supplementary Table 1 of [48]. On the three moderate genomes, Canu achieves the highest assembly quality while it has the worst efficient performance. FALCON, Flye and MECAT share roughly the same behavior in terms of NG50, NG75 and run time. Wtdgb2 is the fastest assembler and is at least one order of magnitude faster than Canu. NG50 and NG75 of assemblies constructed by Wtdbg2 are generally better than those of FALCON, Flye and MECAT. In assembling the 100X human dataset, FALCON has the best assembly quality. The assembly process of Wtdbg2 is dominated by the polishing step, which consumes about 75% of the computation time. The reference genome cover of the Wtgbg2 assembly is significantly smaller than that of Canu and FALCON.

Benchmarks on Nanopore data

The FALCON and MECAT workflows do not work for Nanopore data. Therefore we assess performance of the other three assemblers here. In Table 3 we quote the results from [49] where the three workflows are used for assembling five moderate genomes (100 M). Details about the datasets and experiment setup can be found in Table 1 and Supplementary Table 8 of [49]. As on PacBio datasets, Canu takes the longest time. Wtdbg2 is still the most efficient workflow and can be two orders of magnitude faster than Canu.

On the other hand, due to the great discrepancy between PacBio and Nanopore sequences, the three assemblers behave quite differently on these two kinds of datasets when considering the assembly contiguity. The Flye workflow has the longest N50 on each dataset. Wtdbg2 obtains better N50 on *A. thaliana* and *D. melanogaster* datasets than Canu. While on complex genomes *C. reinhardtii*, *O. sativa* and *S. pennellii*, Canu produces larger N50 than Wtdbg2. This shows the advantage of the Correction-and-Assembly strategies on assembling complex genomes. We also note that Wtgbd2 outputs significantly more contigs than the other two workflows on each dataset.

SOME USE GUIDES

If your task is to carry out a sequence alignment, the preferred alignment tool is generally minimap2 [34], which can meet your most demands of processing. As a general-purpose sequence mapping tool, minimap2

Table 2 Performance evaluation of assemblers on PacBio datasets

Dataset	Metric	CANU	FALCON	Flye	MECAT	Wtdgb2
C. elegans (80X)	Assembly size	106.5 Mb	100.8 Mb	102.0 Mb	102.1 Mb	104.8 Mb
	% reference cover	99.58	99.16	99.29	99.51	99.37
	NG75	1,884,280	935,802	1,275,590	1,424,674	2,255,274
	NG50	2,677,990	1,629,544	1,926,198	2,113,456	3,596,268
	Wall-lock time	9 h 30 m	2 h 6 m	2 h 58 m	3 h 8 m	26 m
A. thaliana (75X)	Assembly size	118.3 Mb	119.3 Mb	116.4 Mb	117.9 Mb	117.8 Mb
	% reference cover	90.53	91.07	90.62	90.91	90.79
	NG75	589,667	6,083,367	3,857,946	1,339,557	6,073,475
	NG50	1,098,921	10,401,798	6,726,569	4,070,235	11,218,688
	Wall-lock time	15 h 50 m	(by PacBio)	3 h 33 m	4 h 31 m	1 h 6 m
D. melanogast (120X)	Assembly size	138.5 Mb		131.2 Mb	134.8 Mb	131.9 Mb
	% reference cover	91.09		89.69	90.26	89.63
	NG75	8,173,560		2,198,937	5,557,762	5,109,865
	NG50	20,301,392		9,318,110	16,078,851	17,035,952
	Wall-lock time	16 h 35 m		5 h 13 m	5 h 3 m	46 m
Human (100X)	Assembly size	2,837 Mb	2,938 Mb			2,712 Mb
	% reference cover	89.33	90.13			86.03
	NG75	3,793,440	7,726,658			4,387,668
	NG50	17,570,750	26,132,317			18,220,221
	Total CPU hours (pre-polish CPU hours)	22,750	68,789			2,506 (632)

Table 3 Performance evaluation of assemblers on Nanopore datasets

Dataset	Workflow	Assembly size	Contigs	NG50 (AP)	CPU hours
A. thaliana	Canu	113408765	288	6,522,919 (28%)	1423
	Flye	126772040	160	12,043,133 (51%)	46.1
	Wtdgb2	115323989	349	9,840,213 (42%)	14.4
D. melanogaster	Canu	146764973	499	3,508,917 (14%)	1548.8
	Flye	138916413	553	10,420,459 (41%)	116.5
	Wtdgb2	138929862	872	6,633,247 (26%)	26.0
C. reinhardtii	Canu	116421921	93	4,563,858 (59%)	18320
	Flye	112517660	52	6,720,472 (86%)	136.3
	Wtdgb2	115667394	344	4,289,786 (55%)	35.4
O. sativa	Canu	383923158	385	5,041,373 (16%)	19568.0
	Flye	380980383	177	8,315,232 (27%)	454.2
	Wtdgb2	394595916	2554	2,432,307 (8%)	154.3
S. pennellii	Canu	961827720	2010	1,663,626 (66%)	21131.5
	Flye	1003210907	2807	1,847,300 (73%)	1687.2
	Wtdgb2	934260260	4986	1,227,952 (49%)	439.0

always runs quite satisfactorily in both efficiency and sensitivity. Minimap2 supports both read-to-read pairwise mapping and reference mapping, and a large spectrum of sequences: DNA and mRNA sequences, PacBio and Nanopore noisy long reads, as well as assembly contigs or closely related full chromosomes of hundreds of megabases in length.

If your study (not necessarily be *de novo* genome assembly) involves raw reads correction as an intermediate step, you do not bother re-implementing the consensus algorithms mentioned above and then proceeding to carry out the overlap-and-correction pipeline. To correct PacBio noisy long reads, either Canu or MECAT is recommended. However if the dataset is large, MECAT is the preferred option, since MECAT has astounding speed benefits and maintains the same level of accuracy assurance as Canu. To correct Nanopore raw reads, the currently available mainstream tool is Canu.

There exist multiple choices of workflows for *de novo* genome assembly with both PacBio and Nanopore sequences. For small and moderate datasets, especially for complex genomes, Canu is able to construct contiguous and accurate assemblies solidly. For large PacBio datasets, if the genome is complex, then MECAT is recommended. Otherwise Flye and Wtdbg2 are also appropriate alternatives. For large Nanopore datasets, we shall first consider using Flye. However if your computing resources is limited you should use Wtdbg2. Note that there are cases in which these workflows can work in a cooperative way. For example, if the quality of contigs assembled by one assembler is stunningly poor, you may wonder if the raw read dataset is corrupt (that is, the data is incomplete or too noisy). To validate your point, you may construct the contigs again with a second fast assembler such as Wtdbg2 or Flye and investigate the quality of the new contigs.

CONCLUSIONS AND DISCUSSIONS

In this survey, we analyze in detail the features of SMS sequences including both the PacBio platform and the Nanopore platform. We review and summarize several important and widely used computation methods that are designed for TGS. We classify these methods into three categories: alignment methods, consensus methods and assembly workflows. The alignment methods and consensus methods are integrated into assembly workflows to carry out intermediate tasks. For each method, we give a general description to its design principle and an in-depth analysis of its core concepts and core ideas. To illustrate the performance of the assembly workflows, we show two benchmark results here and analyze their behavior on each dataset. Based on their design principles and benchmark results, we proceed to provide some general use guides on applying these methods.

Although having made substantial progress, TGS still has defects and the field is still finite that TGS is used in genomic studies. Based on what we have learned, we predict here that in the future TGS will develop towards the following ways. Firstly, as the ultra-long read protocol [8] and the circular consensus sequencing continue to evolve, SMS sequences should keep increasing in both length and accuracy. As results, we get more and more complete and contiguous assembled contigs that cover regions that remain unresolved today. It will be also feasible to perform polyploid assembly of complex genomes. Besides *de novo* assembly, it is expected that TGS will promote studies in other fields such as N6-

methyladenine DNA modification, transcriptome and structural variation detection. Secondly, it is also expected that the third generation sequencing technology also is developed in different areas such as RNA and single-molecule sequencing and to promote the relevant studies with higher resolutions.

We hope that this review can serve as a roadmap for those who are interested in TGS and serve as a foundation based on which more advanced computation methods are developed.

COMPLIANCE WITH ETHICS GUIDELINES

The authors Ying Chen and Chuan-Le Xiao declare that they have no conflict of interests.

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. Brown, S. D., Nagaraju, S., Utturkar, S., De Tissera, S., Segovia, S., Mitchell, W., Land, M. L., Dassanayake, A. and Köpke, M. (2014) Comparison of single-molecule sequencing and hybrid approaches for finishing the genome of *Clostridium autoethanogenum* and analysis of CRISPR systems in industrial relevant Clostridia. Biotechnol. Biofuels, 7, 40
2. Rhoads, A. and Au, K. F. (2015) PacBio sequencing and its applications. Genom. Proteom. Bioinf., 13, 278–289
3. Eid, J., Fehr, A., Gray, J., Luong, K., Lyle, J., Otto, G., Peluso, P., Rank, D., Baybayan, P., Bettman, B., *et al.* (2009) Real-time DNA sequencing from single polymerase molecules. Science, 323, 133–138
4. Detter, J. C., Johnson, S. L., Bishop-Lilly, K. A., Chain, P. S., Gibbons, H. S., Minogue, T. D., Sozhamannan, S., Van Gieson, E. J. and Resnick, I. G. (2014) Nucleic acid sequencing for characterizing infectious and/or novel agents in complex samples. In: Biological Identification, pp. 3–53. Sawston: Woodhead Publishing
5. Wenger, A. M., Peluso, P., Rowell, W. J., Chang, P. C., Hall, R. J., Concepcion, G. T., Ebler, J., Fungtammasan, A., Kolesnikov, A., Olson, N. D., *et al.* (2019) Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. Nat. Biotechnol., 37, 1155–1162
6. 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
7. Magi, A., Semeraro, R., Mingrino, A., Giusti, B. and D'Aurizio, R. (2018) Nanopore sequencing data analysis: state of the art, applications and challenges. Brief. Bioinformatics, 19, 1256–1272
8. Jain, M., Koren, S., Miga, K. H., Quick, J., Rand, A. C., Sasani, T. A., Tyson, J. R., Beggs, A. D., Dilthey, A. T., Fiddes, I. T., *et al.* (2018) Nanopore sequencing and assembly of a human genome with ultra-long reads. Nat. Biotechnol., 36, 338–345
9. Magi, A., Giusti, B. and Tattini, L. (2017) Characterization of MinION nanopore data for resequencing analyses. Brief. Bioinformatics, 18, 940–953
10. Rang, F. J., Kloosterman, W. P. and de Ridder, J. (2018) From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. Genome Biol., 19, 90
11. Loman, N. J., Quick, J. and Simpson, J. T. (2015) A complete bacterial genome assembled *de novo* using only nanopore sequencing data. Nat. Methods, 12, 733–735
12. Walker, B. J., Abeel, T., Shea, T., Priest, M., Abouelliel, A., Sakthikumar, S., Cuomo, C. A., Zeng, Q., Wortman, J., Young, S. K., *et al.* (2014) Pilon: an integrated tool for comprehensive microbial variant detection and genome assembly improvement. PLoS One, 9, e112963
13. Kingan, S. B., Heaton, H., Cudini, J., Lambert, C. C., Baybayan, P., Galvin, B. D., Durbin, R., Korlach, J. and Lawniczak, M. K. N. (2019) A high-quality *de novo* genome assembly from a single mosquito using PacBio sequencing. Genes (Basel), 10, 62
14. Vaser, R., Sović I., Nagarajan, N. and Šikić, M. (2017) Fast and accurate *de novo* genome assembly from long uncorrected reads. Genome Res., 27, 737–746
15. Bashir, A., Klammer, A., Robins, W. P., Chin, C. S., Webster, D., Paxinos, E., Hsu, D., Ashby, M., Wang, S., Peluso, P., *et al.* (2012) A hybrid approach for the automated finishing of bacterial genomes. Nat. Biotechnol., 30, 701–707
16. Koren, S., Schatz, M. C., Walenz, B. P., Martin, J., Howard, J. T., Ganapathy, G., Wang, Z., Rasko, D. A., McCombie, W. R., Jarvis, E. D., *et al.* (2012) Hybrid error correction and *de novo* assembly of single-molecule sequencing reads. Nat. Biotechnol., 30, 693–700
17. Ribeiro, F. J., Przybylski, D., Yin, S., Sharpe, T., Gnerre, S., Abouelleil, A., Berlin, A. M., Montmayeur, A., Shea, T. P., Walker, B. J., *et al.* (2012) Finished bacterial genomes from shotgun sequence data. Genome Res., 22, 2270–2277
18. 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
19. Smith, T. F. and Waterman, M. S. (1981) Identification of common molecular subsequences. J. Mol. Biol., 147, 195–197
20. Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W. and Lipman, D. J. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res., 25, 3389–3402
21. Chaisson, M. J. and Tesler, G. (2012) Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. BMC Bioinformatics, 13, 238
22. Ferragina, P. and Manzini, G. (2000) Opportunistic data structures with applications In: Proceedings the 41st Annual Symposium on Foundations of Computer Science, pp. 390–398. IEEE
23. Li, H. and Durbin, R. (2010) Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics, 26, 589–595

24. Ben, L. and Salzberg, S. L. (2012) Fast gapped-read alignment with Bowtie 2. Nat. Methods 9, 357–359
25. Chen, Y., Ye, W., Zhang, Y. and Xu, Y. (2015) High speed BLASTN: an accelerated MegaBLAST search tool. Nucleic Acids Res., 43, 7762–7768
26. Lam, T. W., Sung, W. K., Tam, S. L., Wong, C. K. and Yiu, S. M. (2008) Compressed indexing and local alignment of DNA. Bioinformatics, 24, 791–797
27. Abouelhoda, M. I. and Ohlebusch, E. A. (2003) Local chaining algorithm and its applications in comparative genomics. In: International Workshop on Algorithms in Bioinformatics, pp. 1–16. Berlin: Springer
28. Eppstein, D., Galil, Z., Giancarlo, R. and Italiano, G. F. (1992) Sparse dynamic programming I: linear cost functions. J. Assoc. Comput. Mach., 39, 519–545
29. Myers, G. (2014) Efficient local alignment discovery amongst noisy long reads In: International Workshop on Algorithms in Bioinformatics, pp. 52–67. Berlin: Springer
30. Cormen, T. H., Leiserson,C. E., Rivest, R. L. and Stein, C. (2009) Introduction to Algorithms (3rd. edition), pp.197–204. Cambridge: MIT Press
31. Myers, E. W. (1986) AnO (ND) difference algorithm and its variations. Algorithmica, 1, 251–266
32. 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
33. Li, H. (2016) Minimap and miniasm: fast mapping and *de novo* assembly for noisy long sequences. Bioinformatics, 32, 2103–2110
34. Li, H. (2018) Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics, 34, 3094–3100
35. Gotoh, O. (1990) Optimal sequence alignment allowing for long gaps. Bull. Math. Biol., 52, 359–373
36. Suzuki, H. and Kasahara, M. (2018) Introducing difference recurrence relations for faster semi-global alignment of long sequences. BMC Bioinformatics, 19, 45
37. Šošic, M. and Šikic, M. (2017) Edlib: a C/C ++ library for fast, exact sequence alignment using edit distance. Bioinformatics, 33, 1394–1395
38. Myers, G. (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. Assoc. Comput. Mach., 46, 395–415
39. Hirschberg, D. S. (1975) A linear space algorithm for computing maximal common subsequences. Commun. ACM, 18, 341–343
40. Lee, C., Grasso, C. and Sharlow, M. F. (2002) Multiple sequence alignment using partial order graphs. Bioinformatics, 18, 452–464
41. 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
42. Myers, E. W. (2005) The fragment assembly string graph. Bioinformatics, 21(suppl_2), ii79–ii85
43. 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
44. 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
45. 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
46. Xiao, C. L., Chen, Y., Xie, S. Q., Chen,K.-N., Wang, Y., Han, Y., Luo F., Xie, Z. (2017) MECAT: fast mapping, error correction, and *de novo* assembly for single-molecule sequencing reads. Nat. Methods, 14, 1072–1074
47. Kolmogorov, M., Yuan, J., Lin, Y. and Pevzner, P. A. (2019) Assembly of long, error-prone reads using repeat graphs. Nat. Biotechnol., 37, 540–546
48. Ruan, J. and Li, H. (2020) Fast and accurate long-read assembly with wtdbg2. Nat. Methods, 17, 155–158
49. Chen, Y., Nie, F., Xie, S.-Q. and Zheng, Y.-F. (2020) Fast and accurate assembly of Nanopore reads via progressive error correction and adaptive read selection. bioRxiv, 930107

