Bioinformatics, 2016, 1–9
doi: 10.1093/bioinformatics/btw470
Advance Access Publication Date: 13 July 2016
Original Paper

OXFORD

Genome analysis

LightAssembler: fast and memory-efficient assembly algorithm for high-throughput sequencing reads

Sara El-Metwally[1,2,*], Magdi Zakaria[2] and TaherHamza[2]

[1]Molecular and Computational Biology, University of Southern California, Los Angeles, CA 90089, USA and [2]Department of Computer Science, Faculty of Computers and Information, Mansoura University, Mansoura 35516, Egypt

*To whom correspondence should be addressed.

Associate Editor: Inanc Birol

Received on April 2, 2016; revised on June 7, 2016; accepted on June 28, 2016

Abstract

Motivation: The deluge of current sequenced data has exceeded Moore's Law, more than doubling every 2 years since the next-generation sequencing (NGS) technologies were invented. Accordingly, we will able to generate more and more data with high speed at fixed cost, but lack the computational resources to store, process and analyze it. With error prone high throughput NGS reads and genomic repeats, the assembly graph contains massive amount of redundant nodes and branching edges. Most assembly pipelines require this large graph to reside in memory to start their workflows, which is intractable for mammalian genomes. Resource-efficient genome assemblers combine both the power of advanced computing techniques and innovative data structures to encode the assembly graph efficiently in a computer memory.

Results: LightAssembler is a lightweight assembly algorithm designed to be executed on a desktop machine. It uses a pair of cache oblivious Bloom filters, one holding a uniform sample of *g*-spaced sequenced *k*-mers and the other holding *k*-mers classified as likely correct, using a simple statistical test. LightAssembler contains a light implementation of the graph traversal and simplification modules that achieves comparable assembly accuracy and contiguity to other competing tools. Our method reduces the memory usage by 50% compared to the resource-efficient assemblers using benchmark datasets from GAGE and Assemblathon projects. While LightAssembler can be considered as a gap-based sequence assembler, different gap sizes result in an almost constant assembly size and genome coverage.

Availability and implementation: https://github.com/SaraEl-Metwally/LightAssembler

Contact: sarah_almetwally4@mans.edu.eg

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

The advent of next-generation sequencing (NGS) technologies has revolutionized the genomic research, but has not been able to provide a complete picture of a sequenced organism, since the relative positions of the billions of fragmented pieces are unknown without a genome assembly, which is a highly ambiguous overlapping puzzle (Nagarajan and Pop, 2013; Pevzner *et al.*, 2001). *De novo* sequence assembly is an initial step towards downstream data analysis such as understanding evolutionary diversity across different species, evidenced by the multitude of data collection projects, including Genome 10K (Koepfli *et al.*, 2015). With the increasing efforts to sequence and assemble the genomes of more organisms, the assembly

problem becomes more complicated and computationally intensive, especially with short inaccurate sequenced reads and genomic repeats (Head *et al.*, 2014).

Next-generation assembly algorithms play around two basic frameworks for efficiently completing their task: namely, De Bruijn and string graphs. In a De Bruijn graph, nodes are the set of distinct k-mers (substrings of length k) extracted from reads and the edges are the $(k-1)$-overlap among them. The string graph is a simplified version of a classical overlap graph, where nodes are the sequenced reads and the non-transitive edges encode their suffix-to-prefix overlaps (El-Metwally *et al.*, 2013, 2014; Myers, 2005; Nagarajan and Pop, 2013).

Many efforts have been made to fit the assembly graph into computer memory by the creation of resource-efficient genome assemblers. The term resource efficiency touches on both memory space and speed (Chikhi *et al.*, 2015). One compressed representation for a string graph is introduced in SGA (Simpson and Durbin, 2012) using FM-index and Burrows–Wheeler transformation of the sequenced reads (Simpson and Durbin, 2010). Recently, an incremental hashing technique combined with a probabilistic data structure (Bloom filter) revisited the string graph representation (Ben-Bassat and Chor, 2014). The early condensed representation of De Bruijn graph is a sparse bit vector (Conway and Bromage, 2011), later implemented in a Gossamer sequence assembler (Conway *et al.*, 2012). This representation is changed in Minia (Chikhi and Rizk, 2013) by introducing the exact representation of De Bruijn graph using the combination of a Bloom filter and a hash table that holds an approximate set of false positive nodes. The hash table is replaced in subsequent versions of Minia by a set of cascading Bloom filters for further space optimization (Salikhov *et al.*, 2014). The Burrows–Wheeler transformation plays another role in the succinct representation of De Bruijn graph (Bowe *et al.*, 2012) by combining FM-index with frequency-based minimizers to reduce its complexity (Chikhi *et al.*, 2015). SparseAssembler (Ye *et al.*, 2012) stores a subsample of k-mers in a hash table with their overlap links, recorded to maintain De Bruijn graph representation. ABySS (Simpson *et al.*, 2009) distributes the assembly graph nodes among different machines to reduce the representation complexity in a computer memory.

Resource-efficient sequence assemblers vary in their assembly results in terms of both accuracy and contiguity measures. Each tool has a set of advantages and disadvantages according to the compromises made to achieve efficiency. Also, different evaluation studies (Bradnam *et al.*, 2013; Earl *et al.*, 2011; Kleftogiannis *et al.*, 2013; Salzberg *et al.*, 2012) generally reported that the assembly algorithms differ in their outputs according to their working scenarios such as the quality of sequenced data and the complexity of the corresponding genome. There is a common conclusion that there is no one tool is best for all scenarios, and that there is still room for improvement in current assembly pipelines.

In this paper, we revisit De Bruijn graph representation and introduce an optimized cache oblivious Bloom filter to the sequence assembly. Our method is inspired by Lighter's idea (Song *et al.*, 2014) to correct the sequenced errors using a pair of Bloom filters and a simple statistical test. Lighter stores a random sample of k-mers in a Bloom filter and uses them with a simple statistical test as seeds to classify the read positions as trusted or untrusted. While Lighter's goal is to use the trust-classified k-mers to correct erroneous ones, our ultimate goal is assembling these k-mers without error correction since they are already classified as trusted nodes (k-mers made by k consecutive trust-classified positions in the sequenced reads are considered to be trusted).

LightAssembler obtains a uniform sample of k-mers by skipping g bases between the k-mers, where g is the gap length and stores them in a Bloom filter. The erroneous bases in a read will produce rare k-mers and are unlikely to survive in the sample compared to the abundant k-mers generated by the correct bases. The trustiness of a read position will be determined by comparing the number of k-mers that cover the position and appear in the sample to a statistically computed threshold. LightAssembler uses the k-mers made by k consecutive trust-classified reads positions as the set of assembly graph traversal nodes, while several assemblers rely on error correction modules to identify and correct the erroneous k-mers before starting the assembly process. The majority of error correction algorithms count the k-mers to determine their confidence and exclude ones with a multiplicity less than a specified threshold, which might result in missing a subset of true k-mers with low abundance. Other assemblers such as Velvet (Zerbino and Birney, 2008) rely on intensive graph simplification modules to resolve the erroneous structures introduced by erroneous bases such as tips and bubbles. Complex assembly pipelines combine both approaches and perform post-processing graph filtering using mate pairs during scaffolding stage.

LightAssembler uses only two passes over the sequenced reads to identify the approximate set of trusted nodes without error correction or intensive graph simplification modules. Also, one of the efficient representations of De Bruijn graph based on a Bloom filter is implemented in Minia and uses k-mer counting module to identify the set of trusted k-mers. Minia's counting algorithm follows a divide and conquer paradigm and utilizes the disk space as secondary memory storage. Our method is able to identify the set of trusted k-mers without utilizing either a counting module or disk-space overhead. We will present our comparable results to the current state-of-the-art sequence assemblers as well as resource-efficient ones using the simulated and benchmarked datasets.

2 Methods

2.1 Pattern-blocked bloom filter

A Bloom filter (Bloom, 1970) is a memory-efficient data structure for storing a given subset of elements $K \subseteq U$. In the assembly context, K is a subset of k-mers and U is the whole set of sequenced k-mers. It supports approximate membership queries on K using a compact representation of its elements (i.e. k-mers). A Bloom filter has a feature of one-sided error, which means if the filter reports yes for an element e then either $e \in K$ or with a small 'false positive' probability $e \notin K$. On the other hand, if the filter reports no then necessarily $e \notin K$. The standard implementation of a Bloom filter is a zero-initialized bit array B with length m and y independent hash functions. To insert an element $e \in K$ in a Bloom filter, the set of indices $H_1(e), H_2(e), \ldots, H_y(e)$ are computed and their corresponding bits in B are set to one, that is $\forall e \in K, B[H_i(e)] = 1$, where $1 \leq i \leq y$. The membership queries are achieved by evaluating the same set of hash functions on an element e and testing their corresponding bits. If all $B[H_i(e)]$ equal 1 then a Bloom filter answers yes and no otherwise. In a Bloom filter, each hash function has equal probability to choose a position in the bit array, so the false-positive rate p_f for the bit array of size m with the number n of inserted elements is:

$$p_f = \left(1 - e^{-y\frac{n}{m}}\right)^y$$

By choosing appropriate Bloom filter parameters m and y, the false positive rate p_f can be adjusted.

The standard Bloom filter implementation is cache-inefficient since each insertion or membership query operation generates at

most y cache misses. The cache-efficient variant of a Bloom filter (Putze *et al.*, 2007) is implemented by a sequence of consecutive blocks, each of size b that can fit into one-cache line. In this implementation, the first hash function is used to choose the block number and the subsequent hash functions are performing in the same chosen block. Therefore, a blocked Bloom filter minimizes the cache misses to one rather than y for each operation. To improve the implementation further in terms of the computation time, the precomputed hash patterns are used to set all y bits at once rather than doing this separately in each block, which is called Pattern-blocked Bloom filter. In a Pattern-blocked Bloom filter, the false-positive rate can be computed using the following equation (Song *et al.*, 2014):

$$p_f = \frac{\sum_{i}^{|B|} \left(\frac{b'_i}{b}\right)^y}{|B|}$$

where $|B| = m - b + 1$, size of a Pattern-blocked Bloom filter in terms of blocks; b, number of bits in one block; b'_i, number of bits set to one in the i-th block; y, number of hash functions.

The false-positive rate is increased in a Pattern-blocked Bloom filter compared to the standard implementation of a Bloom filter and the false positive generally can be managed using large m and y.

LightAssembler uses the Pattern-blocked Bloom filter to hold a set of canonicalized k-mers extracted from the sequenced reads. The canonical k-mer is the minimum lexicographic k-mer of the k-mer itself and its reverse complement.

2.2 LightAssembler framework

The light version of an assembly algorithm can be viewed as the combination of two basic modules, graph construction and graph traversal as depicted in Figure 1.

2.2.1 Graph construction

The graph construction module has two stages, uniform k-mers sampling and trust/untrust k-mers filtering. The input to this module is the set of sequenced reads and the outputs are Bloom filter B and the trusted k-mers file.

*2.2.1.1 Uniform **k**-mers sampling.* The whole set of sequenced k-mers are obtained using a sliding window of length k one base at a time across the input reads. The g-mers or g-spaced k-mers are obtained by skipping g bases between the k-mers. The gap values are in the range $1 \leq g < L - k + 1$, where L is the length of the sequenced read. LightAssembler uses this uniform sampling process to store a sample of k-mers (g-mers) in Bloom filter A. A gap size is chosen by a user or computed by LightAssembler parameters extrapolation module. g-mers in Bloom filter A represent a $1/g$ sample of the nodes diversity in De Bruijn graph of the sequenced genome.

Base-called errors of the NGS technologies are typically identified as unusual events and their corresponding erroneous k-mers are occurring less frequently than the correct k-mers under the assumption of deep uniform coverage (Chaisson *et al.*, 2004; Pevzner *et al.*, 2001; Yang *et al.*, 2013). The most abundant sequenced k-mers will survive in the sample stored in Bloom filter A and we will use them with a simple statistical test as seeds to mark each read position as trusted or untrusted.

*2.2.1.2 Trusted/untrusted **k**-mers filtering.* A trusted k-mer is defined as a k-mer made up by k consecutive trust-classified read positions and it is untrusted otherwise. Each read position is classified as trusted or untrusted based on the following idea with illustration in Figure 2.

Suppose that we have a sequenced read $R = r_1 r_2 r_3 \ldots r_L$, a read position r_i is overlapping by maximum x_i k-mers where x_i is defined as the following:

$$x_i = \begin{cases} i & 1 \leq i < k \\ k & k \leq i \leq L - k + 1 \\ L - i + 1 & L - k + 2 \leq i \leq L \end{cases}$$

If a sequenced base r_i is erroneously called, the overlapped x_i k-mers are all incorrect and occur rarely in the dataset. Accordingly, these incorrect k-mers are unlikely to survive in the sample stored in Bloom filter A. We define a statistically computed threshold for each read position r_i such that if the number of overlapped x_i k-mers appeared in the sample is less than the defined threshold, the read position r_i is classified as untrusted. Otherwise the read position r_i is classified as trusted (Song *et al.*, 2014).

The multiplicity of an incorrect k-mer in the sequenced reads has been modeled previously using Poisson distribution (Chaisson *et al.*, 2009; Lander and Waterman, 1988; Melsted and Halldorsson, 2014; Simpson, 2014). We will assume that the erroneous k-mers will occur at some rate $\lambda_e = c\varepsilon$ in the sequenced dataset and $\lambda'_e = c\varepsilon/g$ in the sample stored in Bloom filter A where c and ε are the expected coverage and error rate of the sequenced reads, respectively.

Suppose M_e is a random variable for the number of times an incorrect k-mer appears in the sample:

$$M_e \sim Pois(\lambda'_e)$$

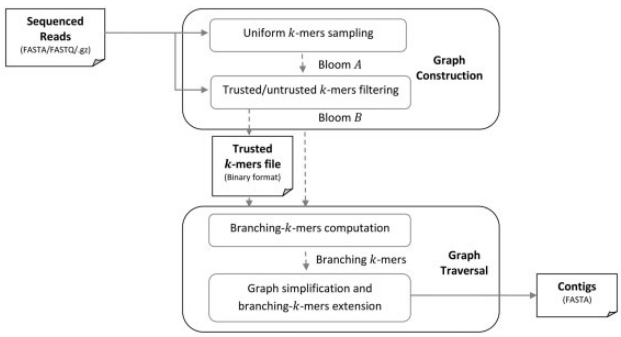

Fig. 1. LightAssembler framework

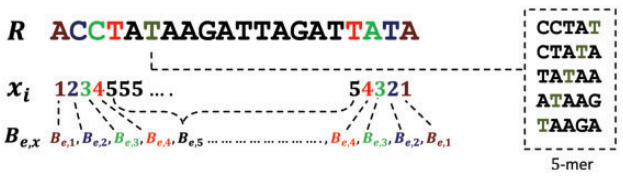

Fig. 2. Overlapped k-mers for read positions. The number of overlapped k-mers, x_i , for each read position r_i considering the reads end effects, where $x_i \in [1, k]$. When read length $L = 20$ and $k = 5$, $x_1 = 1$, $x_2 = 2, x_3 = 3$, $x_4 = 4$, $x_5 = 5, x_6 = 5, \ldots x_{L-k} = 5, x_{16} = 5, x_{17} = 4, x_{18} = 3, x_{19} = 2$ and $x_{20} = 1$. Each random variable B_{e,x_i} is defined for each $x_i \in [1, 5]$, in this example, we have $B_{e,1}, B_{e,2}, B_{e,3}, B_{e,4}$ and $B_{e,5}$

The probability of an incorrect k-mer appears in the sample p' is defined as:

$$p(M_e \geq 1) = 1 - p(M_e < 1)$$
$$= 1 - p(M_e = 0)$$
$$= 1 - e^{-c\varepsilon/g}$$

By considering the false-positive rate p_f of Bloom filter A that stores a sample of k-mers, we redefine p' as:

$$p' = p_f + (1 - p_f)\left(1 - e^{-\frac{c\varepsilon}{g}}\right)$$

Let B_{e,x_i} be a random variable for the number of g-mers appeared in the sample stored in Bloom filter A for an untrusted position r_i and these g-mers are overlapped by $x_i \in [1, k]$:

$$B_{e,x_i} \sim \text{Binom}(x_i, p')$$

We compute a threshold t'_k for each $x_i = k$ as the minimum integer that satisfies the following equations:

$$p(B_{e,k} \leq t_k - 1) \geq 0.9$$
$$t'_k = t_k + \left\lfloor \frac{k}{g} \right\rfloor$$

Then for each $x_i \in [1, k-1]$, we define other t'_{x_i} thresholds as:

$$t'_{x_i} = \left\lceil t'_k \times \frac{x_i}{k} \right\rceil$$

2.2.2 Graph traversal

Since our goal is efficiently using the computational recourses to construct and traverse De Bruijn graph, we found the most optimized traversal algorithm for visiting and marking the graph nodes is implemented in Minia. We modified Minia's traversal algorithm according to LightAssembler graph construction method. There are two steps in the graph traversal module: (i) computing branching-k-mers and (ii) simplifying De Bruijn graph and extending the branching-k-mers (Fig. 3). The inputs to this module are Bloom filter B and the trusted k-mers file and the output is the set of assembled contigs.

2.2.2.1 Branching-k-mers computation. The traversal algorithm starts by computing the set of branching nodes (k-mers have multiple extensions) by querying Bloom filter B for each k-mer in the trusted k-mer file. We use this file to store the set of trusted k-mers to avoid the third iteration over the dataset that might introduce false-positive nodes. The file contains only the minimum set of trusted graph nodes in a binary format. The file is removed after the branching k-mers are computed and stored in a hash table, which serves as a recording structure for the visited branching nodes.

2.2.2.2 Graph simplification and branching-k-mers extensions. Every assembled contig represents a simple path starting from a branching node and ending with a marked branching node. Rather than marking every used node on a simple path that adds an additional space overhead, we only mark the branching nodes as terminal points for every simple path. The number of branching nodes also compared to the complete set of graph nodes is very small and depends on the genome complexity. Storing and marking only the branching nodes saves an additional space in LightAssembler implementation. The assembly graph has dead end paths called tips, LightAssembler removes the tips of length $2k + 1$ bases or shorter.

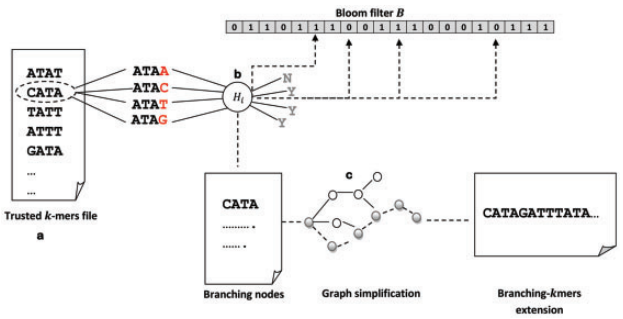

Fig. 3. LightAssembler graph traversal module. The first step in the graph traversal module is computing the set of branching k-mers (k-mers have multiple extensions). (**a**) The successors for each trusted k-mer are computed by appending a nucleotide $nt \in \{A, C, G, T\}$, for example, the successors of a k-mer = *CATA*, where $k = 4$ are $\{ATAA, ATAC, ATAG, ATAT\}$. (**b**) Bloom filter B is queried for each successor to check its presence in the sequenced reads. If the number of existing successors for each trusted k-mer is larger than one, the trusted k-mer is considered as a branching node. Otherwise, it is a simple node. (**c**) Each assembled contig starts from a branching node in the assembly graph, where each node is extended one nucleotide at a time and Bloom filter B is continuously queried for checking the presence of extended k-mers. The assembly graph is simplified by removing the dead end paths and resolving the simple bubbles

LightAssembler also resolves the bubbles that represent the multiple paths started from a branching node by traversing them until the convergence point is reached and accordingly, LightAssembler finds the best consensus sequence that is expressed by all multiple branches. LightAssembler ignores solving the complex bubbles that might solve using the paired-end information encoded in the sequenced libraries.

2.3 LightAssembler usability and scalability

LightAssembler has two parameters that a user must specify, a genome length G and a k-mer size k. A gap size g is an optional parameter, which can be set by a user or LightAssembler invokes the parameters extrapolation module to compute a gap size based on a sequenced coverage c and an error rate ε. A user can utilize our suggested gap starting values presented in **Table 1**, which are computed based on the simulated datasets using different sequenced coverage and error rates. Also, the genome size and the k-mer size can be estimated using stand-alone tools such as KmerGenie (Chikhi and Medvedev, 2014) and KmerStream (Melsted and Halldorsson, 2014). Moreover, LightAssembler is a multithreaded program with an optional parameter t to specify the number of working threads, where the default value is one, for more detail of parallelized implementation see Supplementary 1 Results, section 1.

3 Results

We evaluated the performance of LighAssembler against Minia v2.0.3 (Chikhi and Rizk, 2013), SparseAssembler (Ye *et al.*, 2012) and ABySS v1.5.2 (Simpson *et al.*, 2009) using simulated datasets from the *Escherichia coli* reference genome [GenBank: NC_000913.2] with different attributes listed in Supplementary 1 Table 1. We also compared our results with the same assemblers, including Velvet v1.2.10 (Zerbino and Birney, 2008) using real benchmark datasets from GAGE (Salzberg *et al.*, 2012) and Assemblathon 2 (Bradnam *et al.*, 2013) evaluation studies, the characteristics of benchmark datasets are presented in Supplementary 1 Table 2. LightAssembler, like the other chosen assembly tools, is a

Table 1. Suggested starting values for a gap size parameter g, for various sequenced coverage and error rates

Sequenced coverage c	Error rate $e \leq 0.01$	Error rate $e > 0.01$
25×	3	6
35×	4	8
75×	8	15
140×	15	20
280×	25	33

Table 2. Memory usage (peak resident memory in GB) for simulated datasets with 1% error rate[a]

Assemblers	25×	35×	75×	140×	280×
ABySS	1.24	1.51	2.72	4.34	8.25
Minia	0.29	0.34	0.42	0.67	0.83
SparseAssembler	0.14	0.12	0.18	0.20	0.30
LightAssembler	**0.08**	**0.08**	**0.08**	**0.07**	**0.09**

[a]The best value for each column is shown in bold.

De Bruijn graph-based assembler. LightAssembler, Minia and SparseAssembler are also considered as resource-efficient contig-based assembly tools. They do not utilize the paired-end information encoded in the sequenced libraries to perform scaffolding, while ABySS and Velvet have their own scaffolding modules. In order to make a fair comparison, we evaluated all methods based on their resulted contigs without using paired-end information such as the insert size. Then, we performed scaffold analysis based on our resulted contigs compared to those from other methods using SSPACE v3.0.0 (Boetzer *et al.*, 2011) as one of stand-alone scaffolding tools.

One of the major assembly steps is evaluating assembly results to assess their accuracy and contiguity. When a reference genome is available, the assembly results are evaluated by mapping the assembled contigs or scaffolds back to the reference with the possibility of setting a minimum length threshold for the mapping contigs/scaffolds. The assembly evaluation tools have different reported metrics and even different approaches when a reference genome is absent. GAGE, Assemblathon and QUAST (Gurevich *et al.*, 2013) are the most popular tools. The metrics used to evaluate the assembly results from the competing tools are described in Supplementary 1 Table 3 according to their definition in GAGE and Assemblathon studies. While QUAST has GAGE option to run the assembly evaluation using GAGE standards, we found that for the same minimum contigs threshold length, QUAST NG50 equals to GAGE N50 before performing the contigs correction step (breaking contigs at every misjoin and at every indel longer than 5 bases.). QUAST has a slightly higher N50 contig length compared to those from GAGE and Assemblathon 2. The default minimum threshold for contigs analysis in QUAST is 500 bp, which can be adjusted by the end user. GAGE has a fixed threshold contig length equals to 200 bp, while it is not specified in the Assemblathon's paper their threshold-based analysis. The scaffold-based contiguity analysis is used in Assemblathon 2 to report the contigs statistics by breaking the scaffolds into their corresponding contigs, which increases the N50 contig length compared to the length reported by the contigs-based analysis from other evaluation tools. Also, NG50 reported by the Assemblaton script equals to the N50 length reported by GAGE before doing the contigs correction step. We will use GAGE script to evaluate the assembly results from all competing programs. All conducted computer experiments and the exact command lines used for each tool are described in detail in Supplementary 1.

3.1 Resource requirements

The major goal of LightAssembler is to use the computational resources efficiently, in particular reducing the memory requirements for contigs production. We compared LightAssembler memory usage and the running time with those of ABySS, Minia and SparseAssembler using simulated datasets and with the same assemblers, including Velvet using real benchmark datasets. The experiments were run on a computer running Red Hat Linux with 16 2.4-GHz Xeon E5-2665 cores and 128 GB memory. The maximum memory for our computational resources is 128 GB, which is not sufficient to run Velvet and ABySS for the bird dataset from the Assemblathon project, so we used temporarily another limited-access machine with 1TB memory.

The memory usage peaks for the simulated datasets with low error rate ($e \approx 1\%$) and real benchmark datasets are presented in Tables 2 and 3. For the simulated datasets with 3% error rate, see Supplementary 1 Table 4. LightAssembler has the lowest peak memory usage for all different simulated datasets that vary in the sequencing coverage from 25× to 280×. Also, LightAssembler has the lowest peak for different real datasets that vary in sizes from 2.9 Mbp to 1.23Gbp. The memory usage is almost constant for different sequencing coverage of the same genome and increases slightly when the coverage is 280×. SparseAssembler has the next lowest memory peak, but the memory peak increases greatly when the sequenced coverage is increased and the gap size is decreased. While Minia uses less memory than ABySS and Velvet, it utilizes the disk space to overcome the memory usage limitations, which increases Minia's peak for the virtual memory usage. The assembly results by Minia vary greatly when the disk space is not sufficient to run the k-mer counting module. Velvet uses less memory than ABySS, which is designed to distribute the overhead of memory usage among multiple machines. The disk size of the assembly results for real datasets is reported in Supplementary 1 Table 5.

We also studied the peak memory usage of LightAssembler using different gap values (see Supplementary 1 Results, section 2).

We reported the running time for Velvet, ABySS and LightAssembler using only one thread (see Supplementary 1 Tables 6–8). The latest version of Minia adjusts the number of threads according to the number of the available cores without being able to modify the number of threads to a single-thread mode. SparseAssembler is a single-thread assembler where Velvet, ABySS, Minia and LightAssembler are able to run in the multithreaded mode.

3.2 Simulated datasets

The assembly results for all simulated datasets are presented in Table 4 and Supplementary 1 Table 9 with more detail in Supplementary 2. Also, apart from Minia, SparseAssembler, ABySS and LightAssembler achieve the highest N50 length with low coverage dataset ($c \leq 35x$) and low error rate ($e \approx 1\%$). When the error rate is increased to 3%, Minia and LightAssembler achieve the highest N50 length. LightAssembler outperforms all methods with the high coverage dataset ($c \geq 75x$) and low error rate ($e \approx 1\%$) according to the different evaluation metrics. ABySS utilizes the high coverage to overcome the sequenced errors when the error rate is increased to 3% and achieves the best results. It seems from our simulation that all assemblers have comparable results when the error rate is 1% and the average coverage $c \geq 75x$. LightAssembler outputs the minimum number of contigs across different coverage values when the error rate is 1% and has comparable numbers to

Minia in the experiments with 3% error rate. LightAssembler, SparseAssembler and ABySS have no assembly errors across all different experiments according to the error definition in Supplementary 1 Table 3, while all assemblers have indel (i.e. insertion/deletion) errors with length less than 5 bases (see Supplementary 2). For the simulated datasets, we found that the maximum corrected contig length (max corr.) and the N50 corrected length (N50 corr.) equal to their lengths before doing the correction step (see Supplementary 2). SparseAssembler produces the highest genome coverage for all simulated datasets because of the large assembly size, which increases when the sequenced coverage and error rate of datasets are increased and the gap size is decreased. SparseAssembler has also the largest chaff size, which is the total size of all contigs that are in length less than 200 bp. LightAssembler, ABySS and Minia have the constant genome coverage across different scenarios.

The assembly parameters supplied to all programs are presented in Supplementary 1 Table 10. Minia uses the minimum abundant parameter to filter the erroneous k-mers by implementing a k-mer counting module to remove k-mers with abundance less than a specified threshold. While the k-mers counting module has an additional overhead of the running time and disk usage, Minia's performance drops dramatically when we tried to use all k-mers in the simulated datasets (minimum abundant threshold equals one). LightAssembler and SparseAssembler use the gap size parameter to get a uniform sample of k-mers in the sequenced reads. While LightAssembler uses the sample to filter the whole set of sequenced k-mers, SparseAssembler assembles the sampled k-mers by extending their links. The assembly results changed dramatically with different gap sizes in SparseAssembler, unlike LightAssembler, which was relatively insensitive, starting from a suitable value, for datasets with low error rates (see Supplementary 3). SparseAssembler has a maximum gap size, $g = 25$, We think that this gap size should increase for the high coverage datasets with $c \geq 140x$.

We also studied the effect of varying the gap size g and the k-mer size k on the assembly results of LightAssembler (see Supplementary 1 Results, section 3).

Table 3. Memory usage (peak resident memory in GB) for real benchmark datasets[a]

Assemblers	*S.aureus*	*R.sphae.*	H. chr14	Bird
Velvet[b]	1.88	2.65	19.69	281.9
ABySS	2.60	3.71	28.27	451.24
Minia	0.25	0.34	2.84	32.24
SparseAssembler	0.13	0.23	1.57	13.62
LightAssembler	**0.05**	**0.06**	**0.79**	**6.72**

[a]The best value for each column is shown in bold.

[b]We reported the highest peak memory usage among the two-steps assembly process.

3.3 Real datasets

The assembly parameters supplied to all programs are presented in Supplementary 1 Table 11. The detailed assembly results for each assembler on each dataset are presented in Supplementary 2.

3.3.1 Accuracy *of LightAssembler k-mers classification*

We measured the number of correct classified k-mers by LightAssembler compared to the number of distinct k-mers in reference genomes for real datasets (Table 5). Also, we reported the number of incorrect k-mers that are kept in Bloom filter B. While LightAssembler kept some incorrect k-mers, the number of introduced errors in the final assembled contigs is comparable to Minia and SparseAssembler and is very low compared to Velvet and ABySS. In addition, we studied the effect of varying a gap value on the accuracy of LightAssembler k-mers classification with more detail in Supplementary 1 Results, section 4.

3.3.2 GAGE *human chromosome 14*

Human chromosome 14 is a paired-end dataset from the GAGE evaluation study. We evaluated assembly results using the GAGE evaluation metrics that are computed based on aligning the assembled sequences to the reference of Human chromosome 14 dataset.

The assembly results for all assemblers are presented in Table 6. No assembler performs the best on all combined metrics. Velvet produces the highest N50 length at the expense of introducing more errors compared to the other assemblers. Minia produces the lowest

Table 4. Assembly statistics for simulated datasets with various sequencing depths and 1% error rate[a]

Evaluation metrics	Assemblers	25×	35×	75×	140×	280×
num	ABySS	873	743	610	609	609
	Minia	586	381	248	246	249
	SparseAssembler	4105	4522	16 853	23 418	79 475
	LightAssembler	**474**	**307**	**217**	**221**	**224**
max (bp)	ABySS	**111 690**	**269 704**	326 386	326 386	**326 386**
	Minia	88 437	162 370	326 386	326 386	**326 386**
	SparseAssembler	63 311	171 510	**326 391**	**326 390**	296 443
	LightAssembler	80 023	171 498	326 386	326 386	**326 386**
N50 (bp)	ABySS	**28 615**	**55 252**	59 812	59 812	59 812
	Minia	17 944	39 965	59 812	59 812	59 812
	SparseAssembler	14 759	44 088	**60 168**	**60 170**	57 852
	LightAssembler	26 488	52 505	60 160	60 166	**60 166**
coverage (%)	ABySS	96.28	96.27	96.37	96.37	96.37
	Minia	95.65	95.53	95.52	95.61	95.62
	SparseAssembler	99.11	100.05	120.24	128.82	209.34
	LightAssembler	95.55	95.53	95.57	95.60	95.56

[a]The best value for each column is shown in bold.

Table 5. Accuracy of LightAssembler *k*-mers classification for real datasets

Dataset	Distinct *k*-mers	Correct *k*-mers	Accuracy (%)	Incorrect *k*-mers
S.aureus	2 858 856	2 848 016	99.62	337 600
R.sphae.	4 558 417	4 342 986	95.27	75 338
H. chr14	86 467 655	82 706 223	95.65	16 977 939

Table 6. Contigs statistics for the GAGE human chromosome 14 dataset (ungapped size 88 289 540 bp)[a]

Evaluation metrics	Velvet	ABySS	Minia	Sparse Assembler	Light Assembler
num	**42 939**	190 356	70 469	233 840	64 595
max (bp)	63 828	**65 461**	47 457	55 666	60 825
max corr.	63 834	**65 463**	47 458	49 794	60 823
N50 (bp)	**4318**	3857	3115	3516	3547
N50 corr. (bp)	**4138**	3742	3064	3434	3472
errors	1473	1044	**816**	1001	904
coverage (%)	97.41	109.20	99.42	110.24	98.91

[a]The best value for each row is shown in bold.

number of errors with LightAssembler very close behind. The assembled sequence of ABySS and SparseAssembler is longer than the reference sequence, which increases their genomic coverage compared to other assemblers. Velvet and LightAssembler outputs the minimum number of contigs, while ABySS has the maximum contig length. Overall, LightAssembler performance on human chromosome 14 is comparable to other assemblers for all evaluation metrics.

3.3.3 Assemblathon 2 *bird dataset*

LightAssembler is designated to overcome the intensive memory requirements for assembling large genomes, so we compared LightAssembler results with other assembly tools (Table 7) using one of the vertebrate species (*Melopsittacus undulatus* or simply bird dataset) from Assemblathon 2.

Since there is no reference genome available for the bird dataset, we evaluated different results based on the assembly contiguity metrics. We used GAGE script to compute the N50 length with the minimum threshold set at 200bp for the contigs-based contiguity analysis. SparseAssembler has the highest N50 length, while LightAssembler has the second best value. As we reported previously, ABySS and SparseAssembler have the largest assembly size and the highest number of resulted contigs. Overall, LightAssembler results for the bird dataset are comparable using all contiguity evaluation metrics.

3.3.4 *Importance of error correction and data cleaning*

We used two bacterial genomes (*S.aureus* and *R.sphaeroides*) from GAGE project to compare the performance of all assemblers using uncorrected and corrected sequenced reads. We used the error-free datasets corrected by ALLPATHS-LG (Gnerre *et al.*, 2011) since it has one of the best error correction modules for these datasets as mentioned in GAGE.

The assembly results for all methods are increased dramatically after the error correction step (Table 8), which highlights the importance of data quality on the assembly process. Some assemblers like Velvet have extensive graph simplification modules, which resulted in high N50 length before data cleaning process at the expense of introducing more errors in the finished assembled contigs. The number of errors might be increased for some assemblers after data cleaning due to false-positive or false-negative error correction. Some error correction tools have also trimming processes, which truncate the tails of the sequenced reads at the bases with low quality scores and discard the reads that cannot be corrected. ABySS and SparseAssembler have the largest assembly size, which is reduced effectively after error correction. The large assembly size can mislead the assembly evaluation because it might result from errors in the dataset (tips or dead-ends) or repeated regions when assemblers infer different paths due to heterozygosity. More discussion can be found in Supplementary 1 Results, section 5 and Supplementary 2.

Table 7. Contigs statistics for Assemblathon 2 bird dataset (estimated genome size 1.23 Gbp)[a]

Evaluation metrics	Velvet	ABySS	Minia	Sparse Assembler	Light Assembler
num	**841 486**	4 978 938	1 374 322	1 715 152	1 160 406
max (bp)	138 491	**203 054**	138 470	188 160	180 702
N50 (bp)	3048	1593	2739	**5756**	3674
coverage (%)	87.90	124.32	89.87	99.41	96.09

[a]The best value for each row is shown in bold.

3.3.5 *Scaffold results for* GAGE *human chromosome 14*

Sequence assembly is the combination of two stages: contig assembly followed by the use of paired-end reads or mate pairs to link the contigs further into scaffolds. Typically, contig assembly is the most memory-intensive stage for an assembler compared to the scaffolding stage. Some assemblers have their own built-in scaffolding modules, while others rely on the stand-alone tools to accomplish scaffolding. LightAssembler, Minia and SparseAssembler are contig-based assemblers, which are not designated to consider the paired-end information and perform scaffolding, while Velvet and ABySS have their own scaffolding modules.

We assessed the contiguity and accuracy of the resulted contigs into scaffolds for all assemblers using the SSPACE scaffolding tool. The stand-alone scaffolding tools vary in their assembly results according to the number of correct/incorrect misjoins they made between the contigs as reported recently in one of the evaluation studies (Hunt *et al.*, 2014). They reported SSPACE as one of the best scaffolding tool for their defined assembly evaluation metrics. Also, Human chromosome 14 is one of the benchmark datasets used in their study. We used the contigs resulted by all assembler and GAGE short jump library for Human chromosome 14 and ran SSPACE with the same reported best parameters for this dataset. In order to make a fair comparison, we used the contigs file from Velvet and ABySS without utilizing their own scaffolding modules. The assemblers' contig files are produced using only one sequenced library and without correcting the sequenced errors as mentioned previously.

The scaffolding results are presented in Table 9 with more detail in Supplementary 1 Results, section 6 and Supplementary 2. The maximum scaffold length and the minimum number of scaffolds are resulted from the contigs produced by Velvet, while the lowest number of misjoins is resulted from SparseAssembler and LightAssembler. Velvet also has the highest N50 scaffold length, which is reduced dramatically after breaking scaffolds at every indel and at every misjoin. We also evaluated the scaffolding results of LightAssembler contigs

Table 8. Comparison of assembly statistics using uncorrected and corrected reads[a]

Evaluation metrics	Assemblers	*S.aureus* uncorr.	*S.aureus* corr.	*R.sphae*. uncorr.	*R.sphae*. corr.
num	Velvet	**764**	**601**	**2770**	**1201**
	ABySS	4706	1453	5184	2465
	Minia	1207	745	3777	1497
	SparseAssembler	7843	794	19 126	1263
	LightAssembler	1331	745	6504	1493
max corr. (bp)	Velvet	**42 855**	**91 987**	**26 045**	**40 052**
	ABySS	28 746	52 999	25 757	26 496
	Minia	33 620	72 450	16 174	29 331
	SparseAssembler	25 170	41 189	16 502	40 038
	LightAssembler	29 864	72 450	20 643	29 331
N50 corr. (bp)	Velvet	**11 558**	**15 789**	3234	**8700**
	ABySS	7122	14 636	**3766**	7373
	Minia	7225	14 576	2529	7639
	SparseAssembler	7353	10 439	3068	7402
	LightAssembler	6026	14 576	1220	7639
errors	Velvet	10	13	12	12
	ABySS	4	10	9	**6**
	Minia	**3**	**5**	10	7
	SparseAssembler	4	10	**7**	9
	LightAssembler	4	7	9	8
coverage (%)	Velvet	97.93	98.16	97.88	98.59
	ABySS	103.53	98.86	100.77	98.25
	Minia	98.61	98.32	100.31	98.79
	SparseAssembler	107.03	98.04	111.25	97.92
	LightAssembler	98.78	98.38	99.98	98.88

[a]The best value for each column is shown in bold.

Table 9. Scaffolds statistics for GAGE human chromosome 14 dataset[a]

Evaluation metrics	Velvet	ABySS	Minia	Sparse Assembler	Light Assembler
num	**23 427**	170 168	48 375	212 893	43 841
max (bp)	**341 738**	241 036	232 760	204 173	218 388
N50 (bp)	**42 400**	35 290	26 599	31 535	31 293
N50 corr. (bp)	**38 567**	33 938	25 135	30 019	30 141
misjoins	49	42	42	**27**	28

[a]The best value for each row is shown in bold.

using error corrected version of GAGE short jump library supplied to SSPACE, the N50 scaffold length is increased to 36426 bp. Moreover, the N50 scaffold length is increased to 46582 bp and the N50 contig length is increased to 4171 bp when we supplied the error corrected version of GAGE human chromosome 14 dataset to LightAssembler, more detail in Supplementary 3. The scaffolding results of LightAssembler contigs are comparable to other assemblers for the human chromosome 14 dataset.

4 Discussions

LightAssembler is a light-version of an assembly algorithm that is executed on a desktop machine. It retains the assembly accuracy and contiguity using a pair of Bloom filters, one holding a uniform sample of the sequenced k-mers and the other holding k-mers that are likely correct using a simple statistical test. While LightAssembler is a gap-based assembler, different gap sizes result in an almost constant assembly size and genome coverage with varying in the sequenced coverage. The starting value for the gap size interval is chosen according to the sequenced coverage and error rate of a dataset with the gap value increases when the sequenced coverage and error rate are increased.

We compared LightAssembler results with those from Velvet, ABySS, Minia and SparseAssembler using benchmark datasets from evaluation studies such as GAGE and Assemblathon 2. The assembly results reported in those studies are based on using multiple sequenced libraries (paired-ends and mate pairs) with different insert sizes and the sequenced errors corrected before starting the assembly process. In our paper, we used only one sequenced library for each dataset without error correction to verify the validity of our method without increasing the cost of sequencing process (using more libraries) or using error correction tools. To highlight the importance of these concepts on the assembly process, we studied the effect of the error corrected dataset on increasing the performance of different assemblers. We also discussed the scaffolding results of contigs produced by LightAssembler and other assembly tools using one short jump library from GAGE evaluation study.

The data quality and complexity of the assembled genome rather than the assembler itself play a key role on the assembly results. The accuracy and contiguity of the assembly results are not correlated and the large assembly size can mislead the assembly evaluation. The major goal for resource-efficient contigs-based assemblers such as LightAssembler is reducing the memory usage for contigs production, which is the most memory-intensive stage among different assembly stages. LightAssembler achieved an improvement of a 50% reduction in the memory usage compared to the lowest memory usage reported by the current state-of-the-art assembly tools.

Future improvements to LightAssembler will focus on the exploitation of paired-end information via implementing a built-in scaffolding module. Also, extending LightAssembler approach to metagenomic and single-cell assembly where the coverage is highly non-uniform and the number of sequenced errors and chimeric reads are increased. One possible solution is using different sampling rates, gap values, in the first pass so Bloom filter *A* can be populated with *k*-mers of different abundance from different genomic regions. The statistically computed thresholds in the second pass will be adjusted accordingly. Moreover, LightAssembler is an initial step towards full implementation of a streaming algorithm in the sequence assembly. It is considered as a multi-pass semi-streaming algorithm with low-memory usage for sequence assembly. LightAssembler is an open-source software released under the GNU GPL license.

Acknowledgements

We would like to thank Michael Waterman for hosting SE in his lab and his valuable inputs to this project. Also, we are grateful for Peter Ralph for helpful discussions and feedback.

Funding

This work has been supported by the Egyptian Ministry of Higher Education, the Egyptian Cultural and Educational Bureau - Washington DC (fellowship no. JS-2844) and Google Anita Borg Memorial Scholarship to SE.

Conflict of Interest: none declared.

References

Ben-Bassat,I. and Chor,B. (2014) String graph construction using incremental hashing. *Bioinformatics*, **30**, 3515–3523.

Bloom,B.H. (1970) Space/Time Trade/Offs in hash coding with allowable errors. *Commun. ACM*, **13**, 422. &.

Boetzer,M. *et al.* (2011) Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, **27**, 578–579.

Bowe,A. *et al.* (2012) Succinct de Bruijn Graphs. In: Raphael,B. and Tang,J. (eds.), *Algorithms in Bioinformatics*. Springer, Berlin, Heidelberg, pp. 225–235.

Bradnam,K.R. *et al.* (2013) Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *Gigascience*, **2**, 10.

Chaisson,M. *et al.* (2004) Fragment assembly with short reads. *Bioinformatics*, **20**, 2067–2074.

Chaisson,M.J. *et al.* (2009) De novo fragment assembly with short mate-paired reads: does the read length matter? *Genome Res.*, **19**, 336–346.

Chikhi,R. *et al.* (2015) On the representation of De Bruijn graphs. *J. Comput. Biol*, **22**, 336–352.

Chikhi,R. and Medvedev,P. (2014) Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, **30**, 31–37.

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

Conway,T. *et al.* (2012) Gossamer–a resource-efficient de novo assembler. *Bioinformatics*, **28**, 1937–1938.

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

Earl,D. *et al.* (2011) Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome Res.*, **21**, 2224–2241.

El-Metwally,S. *et al.* (2013) Next-generation sequence assembly: four stages of data processing and computational challenges. *PLoS Comput. Biol.*, **9**, e1003345.

El-Metwally,S. *et al.* (2014) Next Generation Sequencing Technologies and Challenges in Sequence Assembly. *SpringerBriefs in Systems Biology*. Springer-Verlag, New York.

Gnerre,S. *et al.* (2011) High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proc. Natl. Acad. Sci. U. S. A.*, **108**, 1513–1518.

Gurevich,A. *et al.* (2013) QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, **29**, 1072–1075.

Head,S.R. *et al.* (2014) Library construction for next-generation sequencing: overviews and challenges. *Biotechniques*, **56**, 61–66, 68, passim.

Hunt,M. *et al.* (2014) A comprehensive evaluation of assembly scaffolding tools. *Genome Biol.*, **15**, R42.

Kleftogiannis,D. *et al.* (2013) Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures. *PLoS One*, **8**, e75505.

Koepfli,K.P. *et al.* (2015) The Genome 10K Project: a way forward. *Annu. Rev. Anim. Biosci.*, **3**, 57–111.

Lander,E.S. and Waterman,M.S. (1988) Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, **2**, 231–239.

Melsted,P. and Halldorsson,B.V. (2014) KmerStream: streaming algorithms for k-mer abundance estimation. *Bioinformatics*, **30**, 3541–3547.

Myers,E.W. (2005) The fragment assembly string graph. *Bioinformatics*, **21**, ii79–ii85.

Nagarajan,N. and Pop,M. (2013) Sequence assembly demystified. *Nat. Rev. Genet.*, **14**, 157–167.

Pevzner,P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U. S. A.*, **98**, 9748–9753.

Putze,F. *et al.* (2007) Cache-, hash- and space-efficient bloom filters. *Lect. Notes Comput. Sci.*, **4525**, 108–121.

Salikhov,K. *et al.* (2014) Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms Mol. Biol.*, **9**, 2.

Salzberg,S.L. *et al.* (2012) GAGE: a critical evaluation of genome assemblies and assembly algorithms. *Genome Res.*, **22**, 557–567.

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

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

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

Simpson,J.T. *et al.* (2009) ABySS: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

Song,L. *et al.* (2014) Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biol.*, **15**, 509.

Yang,X. *et al.* (2013) A survey of error-correction methods for next-generation sequencing. *Brief Bioinform.*, **14**, 56–66.

Ye,C. *et al.* (2012) Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, **13**, S1.

Zerbino,D.R. and Birney,E. (2008) Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, **18**, 821–829.