BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 24 2014, pages 3515–3523
doi:10.1093/bioinformatics/btu578

Sequence analysis

Advance Access publication September 2, 2014

String graph construction using incremental hashing

Ilan Ben-Bassat* and Benny Chor*

School of Computer Science, Tel-Aviv University

Associate Editor: Gunnar Ratsch

ABSTRACT

Motivation: New sequencing technologies generate larger amount of short reads data at decreasing cost. *De novo* sequence assembly is the problem of combining these reads back to the original genome sequence, without relying on a reference genome. This presents algorithmic and computational challenges, especially for long and repetitive genome sequences. Most existing approaches to the assembly problem operate in the framework of de Bruijn graphs. Yet, a number of recent works use the paradigm of *string graph*, using a variety of methods for storing and processing suffixes and prefixes, like suffix arrays, the Burrows–Wheeler transform or the FM index. Our work is motivated by a search for new approaches to constructing the string graph, using alternative yet simple data structures and algorithmic concepts.

Results: We introduce a novel hash-based method for constructing the string graph. We use incremental hashing, and specifically a modification of the Karp–Rabin fingerprint, and Bloom filters. Using these probabilistic methods might create false-positive and false-negative edges during the algorithm's execution, but these are all detected and corrected. The advantages of the proposed approach over existing methods are its simplicity and the incorporation of established probabilistic techniques in the context of *de novo* genome sequencing. Our preliminary implementation is favorably comparable with the first string graph construction of Simpson and Durbin (2010) (but not with subsequent improvements). Further research and optimizations will hopefully enable the algorithm to be incorporated, with noticeable performance improvement, in state-of-the-art string graph-based assemblers.

Availability and implementation: A beta version of all source code used in this work can be downloaded from http://www.cs.tau.ac.il/~bchor/StringGraph/

Contact: ilanbb@gmail.com or benny@cs.tau.ac.il

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

Received on April 6, 2014; revised on July 2, 2014; accepted on August 14, 2014

1 INTRODUCTION

De novo sequence assembly, namely, reconstructing an unknown genome sequence from a set of overlapping sequence reads, is an important problem in bioinformatics. There has been an extensive research in this area in the past two decades, yielding several efficient methods for the task of sequence assembly. Yet, next-generation sequencing technologies pose challenges to current assemblers. Read sets produced by modern sequencing machines contain up to hundreds of millions short reads. As a consequence, there is an ongoing need for new assembly approaches, which should provide a significant decrease both in memory consumption and running time.

The first assembly paradigm to be commonly used was the 'overlap-layout-consensus' (OLC) one, used in several assemblers (such as Celera and TIGR). The first stage of these assemblers aims to construct an overlap graph representing the sequence reads. In this graph, reads are represented as nodes, and two nodes are connected by an edge if and only if the corresponding reads overlap. Constructing this graph is one of the biggest problems of the OLC paradigm, as this stage is both time and memory intensive. One possible solution for this problem was recently introduced in the LEAP assembler, which uses compact data structures to represent the overlap graph (Dinh and Rajasekaran, 2011).

The de Bruijn paradigm, which was first proposed by Pevzner *et al.* (2001), is more space efficient compared with the OLC paradigm, as it merges reads from different instances of a repeat. In this approach, reads are broken into k-mers, which serve as the nodes in a de Bruijn graph. Thus, reads that come from the same repeat (but from different locations in the genome) share the same path in the de Bruijn graph. However, this approach might increase the ambiguity of assembling short repeats. Several short read assemblers have implemented this approach, e.g. Euler (Pevzner *et al.*, 2001), Velvet (Zerbino and Birney, 2008) and Abyss (Simpson *et al.*, 2009).

In recent years, several improvements of the de Bruijn method have led to a significant decrease in the memory consumption needed for de Bruijn assemblers: Li *et al.* (2010) saved considerable memory usage by not recording read locations and paired-end information; Conway and Bromage (2011) used sparse bit arrays to store an implicit representation of the de Bruijn graph. Ye *et al.* (2012) took advantage of the redundancy in the reads set and constructed roughly an equivalent de Bruijn graph by storing only one out of g nodes (where $10 \leq g \leq 25$). An efficient assembly (both time and memory wise) of a human genome was reported by Cikhi and Rizk (2012), who used Bloom filters to represent the de Bruijn graph, as well as additional data structures for avoiding false-positive nodes. These results were further improved by the usage of cascading Bloom filters (Salikhov *et al.*, 2013).

A different framework is based on the string graph, where the edges of the overlap graph are partitioned to two different types: irreducible edges, which are retained, and transitive edges, which are removed. The notion of transitive edges removal was first introduced in Myers (1995), while the term string graph was first defined in Myers (2005). It was used in the Celera assembler (Myers *et al.*, 2000) and the Edena assembler (Hernandez

*To whom correspondence should be addressed.

et al., 2008). Simpson and Durbin (2012) developed the String Graph Assembler (SGA) further, implementing a new algorithm that outputs the string graph directly (without the need to first construct the overlap graph and only then to remove transitive edges). The string graph approach has the advantage of repeats sharing the same path, without the need to break the reads into k-mers (as in the de Bruijn graph approach).

The ReadJoiner (RJ) assembly (Gonnella and Kurtz, 2012) improved on the SGA assembler in terms of time and memory complexity. This was achieved by first producing a relevant subset of all overlaps between read pairs (using matches between smaller strings as a filter), and then outputting the set of irreducible edges by applying a traversal algorithm on a graph representing the sorted set of candidate overlaps.

In this article, we present a different approach for the construction of the string graph, which resemble the SGA algorithm of Simpson and Durbin (2010), but relies on different theory. We apply hash functions to efficiently store, access and process prefixes and suffixes of reads. This method relies on computing hash values modulo a large prime for all prefixes and suffixes of the reads. Our algorithm deals solely with these hash values, except during a verification process, performed on the reads themselves. The probabilistic method might introduce false-positive and false-negative results, and methods to overcome them are also detailed. The algorithm is relatively easy to implement, as it simplifies the task of identifying irreducible edges, by using probabilistic techniques, such as incremental (rolling) hash and Bloom filters. To the best of our knowledge, this is the first incorporation of these two techniques together in the genome assembly context. We hope and expect these probabilistic techniques will lead to a simplification and improved performance of state-of-the-art assemblers as well. Right now, our initial results improve on the first version of the SGA string graph construction method described in Simpson and Durbin (2010). Further optimizations may substantially reduce the required computational resources, as was the case with the latest version of the SGA assembler. At this point, the method is only a proof of a new concept, and not a complete assembler. However, it can be smoothly combined with other assemblers that are based on the string graph representation of reads.

2 BACKGROUND AND NOTATIONS

2.1 Incremental hash functions of strings

Let $S = s_1 s_2 \dots s_\ell$ be a string of ℓ symbols over the alphabet $\Sigma = \{A, C, G, T\}$. $S[i]$ denotes the i-th symbol of S, and $S[i, j]$ denotes the substring of S that spans $s_i \dots s_j$. A substring of the form $S[1, k]$ is called a prefix of length k of S, whereas a substring of the form $S[k, \ell]$ is called a suffix of length $\ell - k + 1$ of S. Let R be a set of n reads, all of length ℓ. For every read r, its reverse complement, denoted by $\bar{r}$, is the sequence $\overline{r[\ell]} \dots \overline{r[1]}$, where $\bar{b}$ stands for the Watson–Crick complement of base b.

Given a large prime, $m > 0$, there are hash functions mapping strings over Σ into a set of integers $\{0, \dots m-1\}$, such that the probability of two different strings $S_1 \neq S_2$ being mapped to the same value (a collision) is low. We thus introduce the following notation: $F_p(r, k), F_s(r, k)$, denote the results of applying the hash function F to the k long prefix and suffix of read r, respectively. Furthermore, the hash functions we use are *incremental*. They can be incrementally computed on strings [also referred to in the literature as recursive or rolling hash functions (D. Lemire and O. Kaser, submitted for publication)].

Let $r \in \Sigma^*$ be a string, and $b \in \Sigma$ be a single character (one base, in our case). $F : \Sigma^* \to \{0, \dots m-1\}$ is an incremental hash (IH) function if there is a function $H : \{0, \dots m-1\} \times \Sigma \to \{0, \dots m-1\}$ such that for every r, b as above, $F(rb) = H(F(r), b)$ (extension to the right), and a (different) function $G : \{0, \dots m-1\} \times \Sigma \to \{0, \dots m-1\}$ such that for every r, b as above, $F(br) = G(F(r), b)$ (extension to the left). If the hash value of the current string is f, then, for example, f^G denotes the hash value of the string extended by a base G (to the right). We emphasize that the incremental property above applies to strings r of *any length*. The computational complexity of computing the hash of an extension by one letter, given $F(r)$, is thus independent of the length of r. As a consequence, F hashes all prefixes (or suffixes) of a given string in time linear in the length of the string.

A number of IH functions are known, most of which use univariate polynomials over finite fields. We are using a modified version of the Karp–Rabin (KR) fingerprint (Karp and Rabin, 1987). Given a read r of length d, a base (integer) $B > 4$ and a modulus, m, we define the hash value of the entire read r as $F(r) = \sum_{i=1}^{d} \varphi(r[i]) B^{d-i} \bmod m$, where $\varphi : \Sigma \to \{0, \dots, B-1\}$ is a one-to-one function that assigns a different numeric value to every base. For example, if φ satisfies $\varphi(A) = 1, \varphi(G) = 2, \varphi(C) = 3, \varphi(T) = 4$, and furthermore $B = 5$ and $m = 503$, then the hash value of the read AAGTC equals 320. All operations are done modulo m, and hence, the hash values are bounded by m. We can extend both to the left and to the right:

$$F(rb) = B \cdot F(r) + \varphi(b) \bmod m, \tag{1}$$

whereas for s of length d,

$$F(bs) = B^d \cdot \varphi(b) + F(s) \bmod m \tag{2}$$

(we retain the previously computed $B^{d-1} \bmod m$, so computing $B^d \bmod m$ takes one multiplication modulo m). This way, if $B \cdot (m-1)$ fits within one computer word, then every operation can genuinely be considered to be done in a constant time, and computational problems, such as value overflow, do not occur. We will take m to be a prime number in the appropriate range, e.g. $2^{29} - 3$ or $2^{61} - 1$, for 32-bit and 64-bit machines, respectively. Strictly speaking, the prime modulus in KR algorithm should be chosen at random. But because our reads are not chosen by an adversary, using a fixed prime modulo works well in practice.

2.2 Overlaps and string graphs

If the length k suffix of read r equals the length k prefix of read r', we say that r, r' have an overlap of length k (the roles of prefix and suffix can be reversed). As reads come from the two strands of DNA, they have an associated orientation. If the two reads have different orientations, we say that they overlap if there is an overlap (as above) between one read and the reverse complement of the other.

Let R be a set of reads, and $\sigma > 0$ a positive integer. The *overlap graph* of R with respect to the parameter σ is the graph $G = (V, E)$, where every read $r \in R$ is a node (member of V), and there is an undirected edge $e = (r_i, r_j)$ if and only if reads r_i , r_j overlap by at least σ bases.

The string graph (Myers, 2005) is defined in a similar manner, but with several important modifications. To begin with, if read r_i or its reverse complement is contained within read r_j, then r_i is not represented as a vertex in the string graph. Furthermore, to capture the double-stranded nature of the DNA sequences, the edges in the string graph are bi-directional. Even though every read in the sequence data is read in the same direction (from 5′ to 3′), it is not known to which strand the read belongs. The graph should, therefore, encode overlaps with reverse complement reads as well. This goal is achieved by assigning an orientation at the two tips of each edge. Every edge represents an overlap, and has four attributes: two types and two labels.

The overlap types capture the orientation of the reads. Let O be an overlap between reads r and s. If O involves the 5′ tip of r (either its prefix or the suffix of the reverse complement of r), the overlap attribute $type_{rs}$ is P. Otherwise, it is S. $type_{sr}$ is defined in the same manner. We visualize the type by the direction of the arrow incident to the corresponding vertex: overlap with attribute $type_{rs} = S$ is denoted by using an arrow pointing out of r, whereas an overlap with property $type_{rs} = P$ is denoted by using an arrow pointing to vertex r. An edge could have all four possible orientations (PP, PS, SP, SS). The overlap labels are the unmatched portions of the reads, so that, for example, the concatenation of read r and $label_{rs}$ corresponds to the assembly of r and s. When traversing the string graph, a path is valid (represents a valid assembly of reads) only if for every vertex in the path, the arrows entering and exiting the vertex are pointed at opposite directions. Otherwise, the assembly represented by the path includes a read that is assigned different orientation every time that it is involved in an overlap.

An edge of the overlap graph is called *transitive* if it represents an overlap that can be deduced from a pair of two other overlaps. Otherwise, the edge is called *irreducible*. The string graph contains only irreducible edges, and all transitive ones are removed from it. Figures 1 and 2 show a few examples of transitive edges and irreducible edges. In the latter figure, edge labels are also shown. Referring to Figure 1, each corresponding label of the irreducible edge from r to s would be a prefix of the corresponding label of the transitive edge from r to t.

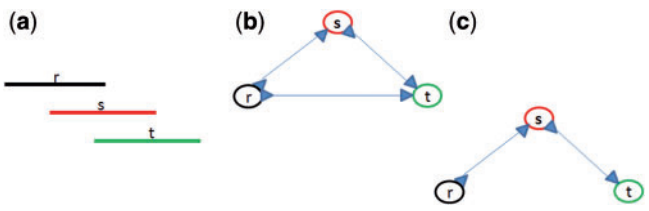

Fig. 1. Read overlaps representation in string graph. (**a**) A set R of three overlapping reads: r, s and t. (**b**) Overlap graph with bidirectional edges. (**c**) String graph representation of R (without the corresponding labels). The transitive edge between r and t is omitted from the graph

2.3 Bloom filters

A Bloom filter (Bloom, 1970) is a probabilistic, hash-based, space efficient implementation of a set data structure, supporting the insert and membership query operations.

A Bloom filter is implemented as an array of s bits, all of which are set to 0 at initialization, and a set of h hash functions, each mapping an element to one of the s array cells, uniformly at random. Insertion of an element to the set is done by computing all h hash values of that element, and setting the h corresponding array cells to 1. If the Bloom filter is expected to store up to n elements, this induces a false-positive probability, `fp`, on membership queries. This is the probability that an answer to a membership query is positive, even though the element was not inserted. It is also termed false positive rate.

A detailed description of the Bloom filter and the relations between its various parameters is deferred to the Supplementary Materials.

2.4 Key terms

Having described notions from several computer science domains that are used in our work, we now present a few terms that are needed for understanding our algorithm. From now on, overlaps will correspond to overlaps by at least σ bases.

Overlapping Prefix (ol-prefix): Given a read s, an *ol-prefix* of s is a prefix of another read that overlaps s. It is composed of two parts: the overlapping part (which equals a suffix of s) and the (possibly empty) extending part.

Ol-prefixes set: Given a read s, an *ol-prefixes set* of s is a collection of *hash values* of all ol-prefixes of s that share the same extending part. The hash value is calculated for the entire relevant prefix (both the overlapping part and the extending part). However, we define the *length* of an ol-prefixes set as the number of bases in the extending part alone (which is the same for all ol-prefixes in the set). Each ol-prefix set is uniquely defined by its extending part.

Ol-prefixes tree: The extending parts of all ol-prefixes of a read s naturally form a tree. The ol-prefixes tree is formed by replacing each extension by the corresponding ol-prefixes set. The root of the tree is the length 0 ol-prefixes set of s (namely, hash values of

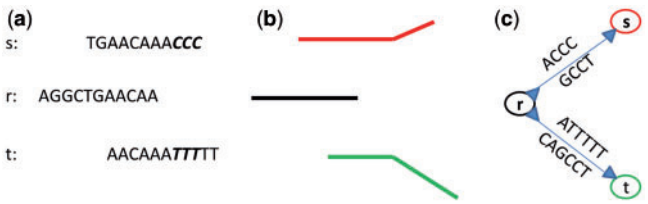

Fig. 2. Read overlaps representation in the string graph: a case where all overlaps are irreducible (no transitive edges). (**a**) Reads r, s and t are aligned according to their bases. The alignment contains a triplet of bases in reads s and t that are a mismatch and marked as bold (CCC and TTT, respectively). (**b**) A graphical representation of the reads alignment: the difference in the reads starting locations is visually depicted via the different starting positions of the lines intervals. The location where the mismatch starts is visually depicted via the breaking point of the first and last lines. (**c**) The corresponding string graph, where all edges are irreducible, and corresponding labels in both directions are shown

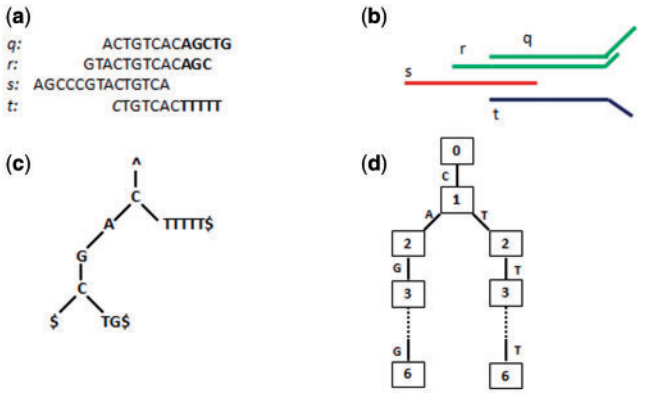

Fig. 3. Illustration of an ol-prefixes tree of read *s*. (**a**) Textual representation of reads *q*, *r*, *s* and *t*. Mismatches in the alignment are in bold face. (**b**) Graphical representation of the alignment of reads *q*, *r*, *s* and *t*. (**c**) Partially compressed tree representation of the extending parts of all ol-prefixes of read *s*. The symbol \$ represents string termination, and the symbol ∧ represents the start of a string. (**d**) The 'skeleton' of the ol-prefixes tree of read *s*. Each node corresponds to an ol-prefixes set. The ol-prefixes set length is specified in the node, and the extending bases appear on the edges

all suffixes of *s* that overlap another read). Nodes at level i correspond to length i ol-prefixes sets of *s*. An edge from level i to level i + 1 in this tree corresponds to one extending base. A path from the root to any node in this tree corresponds to the ol-prefixes set, which is represented by the last node of the path. See Figure 3.

3 APPROACH

We describe a probabilistic IH-based algorithm, for directly identifying the set of all irreducible edges, given a set of reads. Reads are processed as an ordered set of prefixes and suffixes. All string operations are translated to comparisons on the corresponding hash values, except when verifying detected edges. For simplicity reasons, our analysis makes two assumptions:

(1) Data are free of sequence errors.

(2) All reads have the same length.

We remark that the first assumption can be dealt with using existing error correction algorithms, whereas the second can be handled by a simple modification of our algorithm.

Algorithm overview:

The algorithm has three phases:

(1) **Indexing phase:** creates an index of hash values of all reads and their long enough prefixes.

After this step, we process every read *s* to find the set of all its right ol-prefixes sets (corresponding to extensions of *s* to the right). It suffices to consider right extensions, as a left extension of a read corresponds to a right extension of some other read. We perform the following two steps with regard to every *s*:

(2) **Overlaps phase:** using the index generated in the previous step, the algorithm finds, for every read *s*, the set of hash values of prefixes of reads r_i that overlap a suffix of *s* (see Fig. 4).

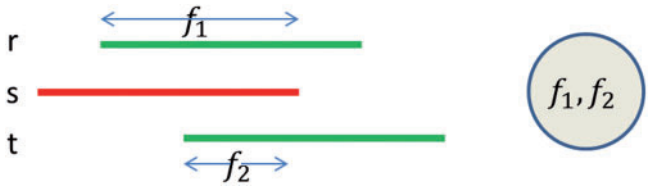

Fig. 4. Overlap phase: two extensions to the right of read *s*. The associated hash values of the two suffixes form the root of the ol-prefixes tree of *s*

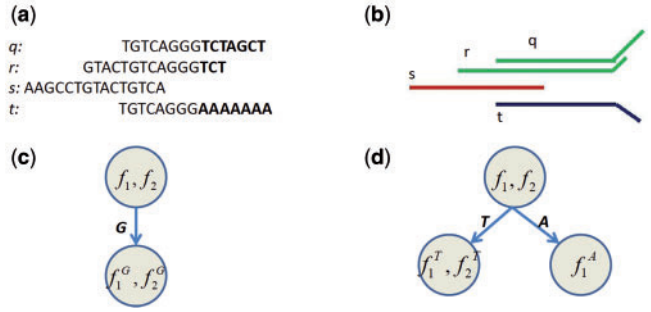

Fig. 5. Overlaps extension phase (regarding read s). (**a**) Textual representation of reads q, r, s and t mismatches in the alignment are in bold. (**b**) Graphical representation of the alignment of reads q, r, s and t. (**c**) The first step in the extension phase. Tree node contains two hash values: f_1 corresponds to the overlaps of q and t with s, while f_2 corresponds to the overlap of r and s. The tree node has only one child, as all overlapping reads perfectly align when extended only one base to the right (by base G). (**d**) The fourth step in the extension phase. At this point, two hash values can be extended by the base T (due to reads q and r) and one hash value can be extended by A (due to read t)

(3) **Overlaps extension phase:** in this phase, we construct the ol-prefixes tree for each read, *s*. Every path from the root to a node (which corresponds to an ol-prefixes set), is elongated, nucleotide by nucleotide, till either an irreducible edge is found or the path becomes too long (see Fig. 5).

Verification: because of the probabilistic manner of using hash functions modulo a large prime, the set of irreducible edges that is reported at the end of the overlaps extension phase might contain false-positive and false-negative edges. This problem is rectified in a verification process. This procedure relies on a data structure, called Signatures, which enables us to recover the reads themselves from their hash values.

We point out that every read, *s*, is processed in the overlaps and overlaps extension phase with respect to two instances of each of the data structures: one for the original reads and one for their reverse complement versions. This enables us to detect overlaps between reads from opposite strands as well.

4 METHODS

4.1 The algorithm

4.1.1 Indexing phase We go over all reads and process them one at a time. To index a read, we incrementally compute the hash values of all its proper prefixes (those longer than σ), and the hash value of the entire

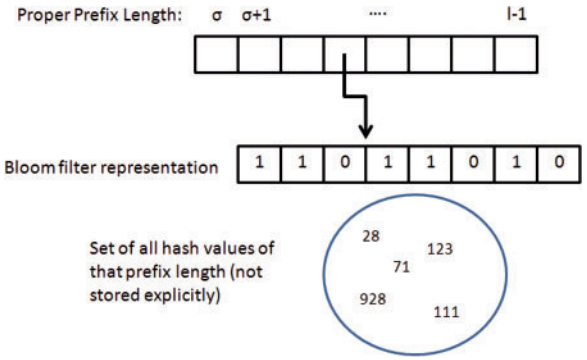

Fig. 6. `Prefixes` data structure, using a Bloom filter. The main array contains a cell for every proper prefix length. Each cell points to a secondary data structure, which is a Bloom filter representation of the set of all hash values computed for that proper prefix length

read (its signature). Using the KR hash function, described in Section 2, we can index every read in time linear in its length. The hash values of the prefixes and the signatures are kept in two distinct data structures: Prefixes and Signatures.

The Prefixes data structure stores hash values of all proper read prefixes. It supports the following operations:

(1) Inserting hash values according to prefix length.

(2) Answering queries of the form: given a hash value h and a prefix length k, does there exist a read index i such that $F_p(r_i, k) = h$.

A natural implementation of the Prefixes data structures is an array of $\ell - \sigma$ *sets* (where ℓ is the length of the reads, and σ is the minimum overlap), with each index corresponding to a different proper prefix length (from σ to $\ell - 1$). Every cell contains a set of hash values computed on prefixes with a certain length. Handling different prefix lengths separately eliminates collisions between hash values computed on prefixes of different lengths. This has an effect on decreasing the number of false-positive and false-negative results.

To reduce memory consumption, a set of hash values is implemented as a Bloom filter, which is designed to store n elements (see Fig. 6).

The second data structure, Signatures, contains a mapping between hash values and all read indexes that have these hash values as their signature. In other words, a mapping between a hash value, h, and the list of all indexes, i, such that $F(r_i) = F_p(r_i, \ell) = h$.

When the indexing phase ends, the data structures contain the entire data needed: both the hash values of all proper prefixes and the mapping of signatures to read indexes.

To account for the double-stranded nature of the DNA sequences, the two data structures are also computed for the set of reverse complements of every read $r \in R$.

4.1.2 Overlaps phase In this phase, we detect, for every read s, the set of right overlaps. However, the goal is not to output a list of all read indexes r_i whose prefixes overlap the read s but to find all hash values of these overlaps instead. So, for every read s, the output of this phase is a set of hash values $h_1, h_2, ..., h_p$, such that for every $1 \leq j \leq p$ there exists an integer $k, \sigma \leq k$ and a read r_i, such that the hash value of the suffix of s of length k equals the hash value of the prefix of length k of r_i (Fig. 5). Note that p is bounded by $\ell - \sigma$, even though the number of reads overlapping s can be up to $O(n)$ (where n is the number of reads).

For every read, s, the overlaps phase is performed in the following manner:

(1) Compute the list of KR hash values of all proper suffixes of s.

(2) For every suffix value and its corresponding length, query the Prefixes data structure for the existence of a prefix of some read, with the same hash value and length.

On overlap detection, a tuple that contains the length of the overlap and its hash value is created. All these suffixes are mutually compatible. We place them in the root node of the ol-prefixes tree of s.

The overlaps phase is summarized by algorithm 1 pseudocode, in the Supplementary Materials. It assumes the existence of a method *GetSuffixValues*. This method incrementally computes hash values of all proper suffixes of a given read, which are longer than a predefined threshold σ. We note that false overlaps might be reported by the algorithm: Hash collisions produce identical hash values for non-identical strings. The same effect is also caused because of false-positive Bloom filter responses. These collisions are rectified in the verification procedure.

Overlaps extension phase:

The ol-prefixes tree construction is done while processing ol-prefixes sets, as if we were scanning all reads that overlap suffixes of s.

Every node contains an ol-prefixes set. For technical reasons, it also contains its string of extending bases. In addition, for future computations, every hash value is also associated with its relevant overlap size.

We construct the ol-prefixes tree in a top-down manner: Initially, all ol-prefixes agree on the same extension ('the empty extension'), and their hash values form the ol-prefixes set at the root. As we extend the overlaps to the right, we add new successor(s) to existing nodes. The construction is performed in a breadth-first manner, storing only the lowest level reached along every branch at each time.

When extending the ol-prefixes that belong to an ol-prefixes set by one base, we may encounter three possible scenarios:

(1) All ol-prefixes agree on the same extending base. In this case, the current node has a single successor in the tree. Its ol-prefixes set contains hash values of this extension (efficiently produced by the IH function).

(2) Some ol-prefixes can be extended by one base while others by another one. In this case, the ol-prefixes set splits to several (two to four) ol-prefixes sets, and the current node has several successors.

(3) One of the ol-prefixes cannot be extended any more. In this case, this ol-prefix is a complete read, and we have reached an irreducible edge (as the extending part of this read is a prefix of the rest of the extending parts in that ol-prefixes set). In this case, the current node is a leaf of the ol-prefixes tree.

The process of adding successor(s) to an existing node is done by working with prefix hash values and querying the Prefixes and Signatures data structures, as follows:

Given a node in the ol-prefixes tree, the algorithm iterates over all hash values in the node. For every hash value, equation 1 (Section 2.1) is evaluated four times, once for every possible base b, to compute the hash value of the prefix, if it was to be extended by base b. This value is then checked against the Prefixes data structure (according to the extended prefix length), to determine whether a prefix with such a hash value and length does exist. Going over all hash values in the node, we maintain a mapping between all four bases and the extending hash values that were found in Prefixes. At the end of the iteration, we can deduce the child nodes of the analyzed tree node: every base b with corresponding hash values belongs to a new child. The edge connecting this child to its parent corresponds to the base b. However, a termination check is conducted before creating the node's children, using the second above mentioned data structure, Signatures. For every new hash value calculated, if the sum of the overlap size and the length of the extending part equals ℓ,

the algorithms check for the existence of that value in Signatures. In case this value exists, the node is not expanded and the path is terminated, as an irreducible edge has just been detected for that ol-prefixes set. We do not further process hash values of prefixes whose length exceeds ℓ (they are removed from the node). A node with no remaining hash values also brings its path to an end. Figure 6 demonstrates the overlaps extension phase. A pseudocode of this phase is given in algorithm 2 of the Supplementary Materials.

There are cases where two reads have more than a single overlap, and only the largest overlap should be considered for an irreducible edge. The algorithm, as described above, might produce false edges in some cases. This is solved by observing that such cases imply a directed edge from a leaf in the ol-prefixes tree back to its root. We identify these cases by retaining the ol-prefixes set of the root, and comparing the hash value of each extension with it.

Verifications:

Because of the probabilistic nature of our method, redundant overlaps, including irreducible string graph edges, can be reported. In addition, true irreducible edges may be missed. We define the following possible types of errors:

(1) false-positive type I: a case where the algorithm considers two non-overlapping reads as overlapping with an irreducible overlap. The error stems from the possibility of two distinct strings having the same hash value or from false-positive queries to the Bloom filter data structures. This error can occur in each step during the construction of the ol-prefixes tree: on creation of the root, on expanding a node or in the termination step, where hash values are compared against the Signatures data structure. The error can also be viewed as a false root node, or a false branching, or an incorrect pruning of a path in the tree.

Figure 1 of the Supplementary Materials illustrates a case of false branching that can cause a false positive of type I. Even though the real overlap should be extended to the right only with the base A at every stage, the algorithm might create a false branching. When trying to expand the hash value $f^{A\ldots A}$ of the prefix of length $k-1$, there could exist a read t, which satisfies $H(f^{A\ldots A}, T) = F_p(t, k)$. This leads to the creation of another path in the tree, which will be extended according to the suffix of length $\ell - k$ of read t (until an irreducible edge is reported).

(2) false-positive type II: a case where the algorithm considers a transitive edge to be an irreducible edge. This error can occur if there is a collision of a proper prefix hash value with a signature of an overlapping read.

(3) False negative: a case where the algorithm misses an irreducible edge. At first glance, this case seems impossible, as the hash values collisions (and the Bloom filter queries) can only create false-positive overlaps. Still, an early termination of a path can cause such a case. Consider a node on a path that corresponds to an irreducible edge. If the algorithm considers one of the node's values to be an evidence of an irreducible edge, then this path is pruned. However, if the reported overlap is a false positive (either type I or II), the algorithm discards the node and no longer processes the other hash values. This means a true irreducible edge is skipped. Another interesting case (described in Fig. 2, Supplementary Materials) can yield only a false negative overlap (without a related false positive).

All of the above error types motivate the need for a verification procedure on the claimed edges. Such procedure should verify the following conditions:

(1) Each detected overlap is genuine. This will discard false-positive results of type I.

(2) The length of the extended prefix equals ℓ. The extended prefix length is the sum of the overlap size and the number of extension steps performed. This ensures that the end of the overlapping read has indeed been reached (as it is possible that a hash value of a proper prefix collides with a read signature). This discards false-positive results of type II.

(3) The suffix of the overlapping read equals the extending bases along the path in the ol-prefixes tree. This prevents false-negative cases as depicted above.

The verification process is not a separate phase but is rather interleaved with the extension phase.

The algorithm described above can produce duplicated irreducible edges, in the case of two reads that overlap in more than a single way. This can be detected by a suitable test.

To address all types of overlaps between reads and their reverse complement, the algorithm should be applied twice for every read: once for overlaps between the read itself and other reads (whether original reads or reverse complement reads) and once for the reverse complement of the read. The first application reveals irreducible edges involving the 3′ side of the read (type S edges in terms of that read), and the second application is responsible for overlaps of the 5′ side (type P edges). The same pair of overlapping reads may create two identical edges, corresponding to different orientations. The algorithm detects this situation and inserts just one of them into the graph.

Correctness proof:

We first show that the algorithm reports no false-negative results: consider an irreducible edge O between reads r and s. Without loss of generality, the hash value of the overlapping part appears in the root of the ol-prefixes tree of read r. The extending part of read s defines a path from the root till the node where this irreducible edge should be discovered. We mark this path as $p = v_1, v_2, \ldots, v_k$, where k equals the length of the read minus the overlap size. Now let us assume, by contradiction, that this path is pruned in some node v_j for some $j \leq k$, such that the true irreducible edge is not reported. If $j < k$ and a different overlap, O', was reported as an irreducible edge, then owing to the verification process, there exists a read t such that:

(1) Read t genuinely overlaps read r.

(2) The extending part of read t (with regard to read r) is a prefix of the extending part of read s (as this overlap is found on the path induced from the overlap between r and s, and this was asserted by the verification process).

(3) The overlap between reads r and t is longer than the overlap between r and s (as the sum of the overlap size and the extending part truly equals the read length, and $j < k$).

These claims imply that overlap O' is an irreducible edge and not O. If $j = k$, a similar proof shows the existence of two identical reads, with contradiction to the assumption of distinct reads.

We now show that the algorithm does not produce false-positive results. It is straightforward to see that no false-positive results of type I occur (due to overlap check). As to type II false-positive results, recall that the algorithm verifies an irreducible edge by checking all three constraints described above. Therefore, if an irreducible edge was found on some path, and bearing in mind that false-negative results do not occur, then this overlap is, by definition, an irreducible edge.

Analysis:

The time complexity of the algorithm, when ignoring the possible false-positive and negative results, is linear in the sum of the lengths of the reads and the total number of irreducible edges. The proof is similar to the one in Simpson and Durbin (2010), bearing in mind that we use IH functions. When constructing the ol-prefixes trees, an irreducible edge e_i is discovered in depth d_i that equals the length of its label (more precisely,

the label that corresponds to the sequence of bases derived from the process of right extension). At every stage where this path is created, the number of hash values updated is at most k_i, the number of reads that belong to the same ol-prefixes set. By the constant number of operations needed for every hash value update, we get that the total number of operations performed for every irreducible edge e_i is $\sum_i d_i k_i$. On the other hand, every read r_i is represented by a path in the string graph, and the edge e_i appears in k_i such paths. As the relevant label is of length d_i, we get that the overall number of bases contributed by edge e_i in all these paths equals $d_i k_i$. Summing over all irreducible edges, we deduce that they contribute $\sum_i d_i k_i$ bases. But as this can not exceed the sum of the lengths of the reads, we get that the time complexity is linear in that size (and in the total number of irreducible edges, as they are stored into memory).

False paths in the ol-prefixes tree might increase the running time of the algorithm. These can occur primarily because of collisions in the KR hash values of different prefixes with the same length. The number of equi length prefixes equals the number of reads. Assuming that reads are unique and of equal length, this number is not larger than the genome length. Therefore, if our prime modulus satisfies $m \approx 2^{61}$, the typical number of reads for current technology is substantially smaller than $\sqrt{m} \approx 2^{30}$. Under the assumption that KR maps equi length strings to $\{0, \ldots, m-1\}$ uniformly at random [this assumption may not hold in a worst case setting (D. Lemire and O. Kaser, submitted for publication)], the probability of having even a single KR collisions is negligible. We remark that small values of m do lead to numerous false collisions, creating false branches in the ol-prefixes trees.

The false-positive rate for our Bloom filter, `fp`, is not negligible. Yet, branches resulting from such false reports are quickly eliminated by the queries to Prefixes or Signatures. Moreover, when verifying a false-positive overlap of two reads, the expected number of comparisons is less than two (as in each position, the two overlapping reads are different with a probability 0.75). This argument supports our experimental results, which indicate that false edges are not created.

The memory requirements of the algorithm are dominated by the index size, which is also linear in the sum of the lengths of all reads. An argument similar to the one made with regard to the time complexity, shows that most ol-prefixes trees do not grow substantially beyond their expected sizes.

5 RESULTS

We have implemented the algorithm in C++. The program receives a file containing reads in a FASTA format, and produces the set of corresponding irreducible edges. We performed two sets of tests, all conducted on a machine with a 2.40 GHz Intel Xeon E7-4870 80 core processor with 132 GB RAM, running a 64 bit Linux operating system, using only a single core.

The first set of tests aimed to check the correctness of the implemented algorithm. In these tests, we have created simulated genomes with different lengths and different frequencies of repetitive zones. These genomes were then randomly sampled to create read files. The output of these tests was compared with the expected set of irreducible edges, obtained by an inefficient naïve algorithm (that was applicable owing to the relatively small size of the simulated datasets).

The second set of tests was conducted using simulated data from human chromosomes 22, 15, 7 and 2. After removing sequence gaps from those chromosomes, we generated 100-base-long reads randomly, at an average coverage of 20×. Reads that appear more than once (including reverse complements) were removed. We then applied our algorithm to construct the string graph of these four different sets of reads (with a minimum overlap of $\sigma = 63$ bases). The time required for these executions, and the memory consumed, are presented in Table 1. In addition, this table includes the time and memory performance of three other algorithms: SGA 2010 (SGA10), corresponding to present SGA code, with the basic index and overlap options of Simpson and Durbin (2010), SGA 2012 (SGA12), corresponding to latest SGA, using more advanced features (Simpson and Durbin, 2012) and RJ, corresponding to ReadJoiner (Gonnella and Kurtz, 2012). See the Supplementary Material for a specification of the commands executed in our comparative study. SGA10 reflects a current implementation, and the results we report here seem better than the original ones, reported in Simpson and Durbin (2010). Our executions were all obtained using a single thread. SGA12 uses a number of optimizations, both in the indexing and overlapping phase, which dramatically improved the memory consumption. We remark that in our current implementation, all data are stored in RAM throughout the entire execution. This differs from the three other tools, where the index is stored on disk.

The performance of our algorithm compares favorably with that of SGA10: its memory consumption is ∼55% of the memory required for SGA10, whereas its execution time is only 75–80%. Regarding SGA12 and RJ, these two are far more memory efficient than our tool (∼10% of our memory consumption); however, IH is slightly faster than SGA12. RJ is

Table 1. Results of running four algorithms on simulated reads for human chromosomes 22, 15, 7 and 2 (coverage = 20×, read length $\ell = 100$, minimum overlap $\sigma = 63$): Incremental Hash (IH), SGA 2010 (S10), SGA 2012 (S12), ReadJoiner (RJ). Memory corresponds to peak usage of RAM

	Chr 22				Chr 15				Chr 7				Chr 2			
	IH	S10	S12	RJ	IH	S10	S12	RJ	IH	S10	S12	RJ	IH	S10	S12	RJ
Memory (GB)	3.6	6.6	0.37	0.33	8.6	15	0.93	0.8	16.6	30	1.8	1.6	25.8	46	2.8	2.3
Total time (s)	3815	4543	3612	170	9310	11892	9886	477	18367	24365	19954	1160	29428	39783	31467	1599
Chr size (Mbp)	34.9				81.7				155.4				238.2			
Number of reads (M)	6.74				16.18				30.84				47.36			
Number edges (M)[a]	6.79				16.42				31.69				48.97			

Memory corresponds to peak usage of RAM.
[a]SGA12 merges non-branching chains of reads, and thus the produced graph has fewer edges.

Table 2. Effect of fp rate on performance (chr 22)

Fp	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Overall memory (GB)	4.8	4.5	4.2	3.9	3.6	3.3
Overall time (s)	6399	5588	4893	4027	**3561**	5675

substantially faster than the other three tools (~15-fold more efficient).

We observed no false-positive edges were created during the executions on the above-mentioned data. This indicates that in practice, the verifications could be safely ignored (see Section 6).

As our algorithm is a proof of concept, we expect that future optimizations will lead to an improved performance, much like the way SGA12 improves on SGA10.

Table 2 depicts the effect of different expected false-positive rates of the Bloom filter on the overall performance. The false-positive rate determines the number of hash functions used and the number of bits allocated for the Bloom filter. If the false-positive rate is low, *e.g.* 10^{-6}, the number of bits in the data structure and the number of hash functions must be large, implying high space and time consumption.

When we increase the false-positive rate, we observe a continuous improvement in both memory and time, with peak performance for $\texttt{fp} = 10^{-2}$. From this point on, the performance of the algorithm deteriorates. This occurs because too many hash values are incorrectly reported to be in the Bloom filter, so the program consumes more memory and runs longer. We note that 10^{-2} can be considered high in many applications. However, false-positive in the Bloom filter part typically does not cause long false branching (as opposed to collisions in the IH part, which tend to propagate longer).

6 DISCUSSION

Our main contribution in this work is the introduction of a novel and efficient method for the construction of the string graph from a set of sequence reads. probabilistic methods are still not widely used in the domain of assembly [apart from the usage of Bloom filter by Cikhi and Rizk (2012)]. The combination of the probabilistic nature and the usage of easily computed hash functions yield a rather efficient and fast index. An encouraging feature of our algorithms is that no false-positive edges have reached the verification stage in all four chromosomes tested. We argued that this desired property is expected to occur for larger instances as well. At this stage, our tool is a proof of concept, and *not* a full fledged assembler. To assemble contigs, the string graph can be traversed and processed.

A strength of our approach, as well as of SGA, is the possibility to access a large number of reads by referring only to a single entity—the hash value of a common prefix (in SGA, this will be a range of consecutive indexes). This implies that the number values in each ol-prefixes set is bounded by the length of the read rather than the number of overlapping reads.

The hash functions used must be incremental to enable us not to store the set of ol-prefixes in memory. The overlap extension phase deals mainly with hash values, and would otherwise be much less memory and time efficient. In addition, the incremental nature of the hash function enable indexing all reads in time linear in the total length of the reads.

We made two assumptions in the design of this algorithm: error-free data and equal-length reads. The first assumption can be dealt with by using existing algorithms that preprocess the reads and use statistical methods to correct sequencing errors. As for the second assumption, modifications can be done in the algorithm to not rely on reads having the same length. For example, we consider an irreducible edge to belong to a read whose extension process is terminated first. If some reads are shorter, this condition is violated, and we need to revert to checking the overlap lengths as well.

The verification process we currently perform requires accessing the reads during the construction of the string graph. We observed that false-positive edges hardly ever appear under the current choice of modulus, m, in the KR fingerprint and the parameters of the Bloom filter. This should enable us to remove all verifications, thus making it possible not to store and access the original reads after the indexing phase, resulting in a somewhat lower memory consumption.

Our present implementation uses a separate Bloom filter for every suffix length, to reduce the number of possible collisions. It may be possible to use a single Bloom filter for all lengths, by properly adjusting the values of the relevant parameters. This could potentially lead to a substantial saving in memory.

There are additional aspects that could be improved. For example, we have used a version of the KR fingerprint, but other IH functions (D. Lemire and O. Kaser, submitted for publication) may perform even better. In addition, many parts of our algorithm can be parallelized. The impact of the hash function moduli and the Bloom filter parameters on the performance can be further investigated. We believe that we have not yet reached the optimal choice of parameter values, which can save more memory and running time, even at the expanse of an increased number of false-positive results (which are subsequently detected). Memory can be slightly saved by representing bases in reads using 2 bits per base, instead of a single character (however, this will not effect the peak memory used). The preliminary results shown here make us believe that the hash-based approach can yield even better results in the future.

The performance of our algorithm is favorably comparable with the first implementation of the FM-index–based assembler, by Simpson and Durbin (2010). A number of newer assemblers are based on string graphs, such as Edena (Hernandez *et al.*, 2008), SGA (Simpson and Durbin, 2012), LEAP (Dinh and Rajasekaran, 2011) and Readjoiner (Gonnella and Kurtz, 2012). These assemblers differ in the data structures and algorithms used to construct the string graph, and in particular how they compute suffix–prefix matches. Their reported performances are substantially better than the performance of our initial implementation. However, we believe that our algorithm is of interest due to its simplicity and the probabilistic techniques that are incorporated in it, and that improvements as outlined above can make it competitive with state-of-the-art string graph algorithms. More generally, we expect that probabilistic approaches can play a key factor in improving other string graph-based approaches.

ACKNOWLEDGEMENTS

The authors thank Richard Durbin, Jared Simpson and the anonymous referees for helpful suggestions.

Funding: This research was partially funded by an Agilent Technologies University Relations grant, by the Deutsch Institute and by the Safra Center for Bioinformatics at Tel-Aviv University.

Conflict of interest: none declared.

REFERENCES

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

Cikhi,R. and Rizk,G. (2013) Space-efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter. *Algorithms for Molecular Biology*, **8**, 22.

Conway,T.C. and Bromage,A.J. (2011) Succinct data structures for assembling large genomes. *Bioinformatics*, **27**, 479–86.

Dinh,H. and Rajasekaran,S. (2011) A memory-efficient data structure representing exact-match overlap graphs with application for next-generation DNA assembly. *Bioinformatics*, **27**, 1901–1907.

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

Hernandez,D. *et al.* (2008) De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Res.*, **18**, 802–809.

Karp,R.M. and Rabin,M.O. (1987) Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, **31**, 249–260.

Li,R. *et al.* (2010) De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, **20**, 265.

Myers,E.W. (1995) Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.*, **2**, 275–290.

Myers,E.W. *et al.* (2000) A whole-genome assembly of *Drosophila*. *Science*, **287**, 2196–2204.

Myers,E.W. (2005) The fragment assembly string graph. *Bioinformatics*, **21**(Suppl. 2), 79-85.

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

Salikhov,K. *et al.* (2013) Using cascading Bloom filters to improve the memory usage for de Bruijn graphs. In: *Algorithms in Bioinformatics Lecture Notes in Computer Science*. Vol. 8126, pp. 364–376.

Simpson,J.T. and Durbin,R. (2010) Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, **26**, 367–373.

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.

Ye,C. *et al.* (2012) Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, **13**(Suppl. 6), 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.