FSG: Fast String Graph Construction for De Novo Assembly of Reads Data

Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali(✉), and Raffaella Rizzi

DISCo, University of Milano-Bicocca, Milan, Italy
{bonizzoni,yuri.pirola,marco.previtali,rizzi}@disco.unimib.it,
gianluca.dellavedova@unimib.it

Abstract. The string graph for a collection of next-generation reads is a lossless data representation that is fundamental for de novo assemblers based on the overlap-layout-consensus paradigm. In this paper, we explore a novel approach to compute the string graph, based on the FM-index and Burrows-Wheeler Transform (BWT). We describe a simple algorithm that uses only the FM-index representation of the collection of reads to construct the string graph, without accessing the input reads. Our algorithm has been integrated into the SGA assembler as a standalone module to construct the string graph.

The new integrated assembler has been assessed on a standard benchmark, showing that FSG is significantly faster than SGA while maintaining a moderate use of main memory, and showing practical advantages in running FSG on multiple threads.

1 Introduction

De novo sequence assembly continues to be one of the most fundamental problems in Bioinformatics. Most of the available assemblers [1,12,13,19,20,25] are based on the notions of de Bruijn graphs and of k-mers (short k-long substrings of input data). Currently, biological data are produced by different Next-Generation Sequencing (NGS) technologies which routinely and cheaply produce a large number of reads whose length varies according to the specific technology. For example, reads obtained by Illumina technology (which is the most used) have length between 50 and 150 bases [21].

To analyze datasets coming from different technologies, hence with a large variation of read lengths, an approach based on same-length strings is likely to be limiting, as witnessed by the recent introduction of variable-length de Bruijn graphs [9]. The *string graph* [18] representation is an alternative approach that does not need to break the reads into k-mers (as in the de Bruijn graphs), and has the advantage of immediately distinguishing the repeats that result in different arcs. The string graph is the main data representation used by assemblers based on the overlap-layout-consensus paradigm. Indeed, in a string graph, the vertices are the input reads and the arcs corresponds to overlapping reads, with the property that contigs are paths of the string graph. An immediate advantage of string graphs is that they can disambiguate some repeats that methods based on de Bruijn graphs might resolve only at later stages—for example, the repeats that are longer than $k/2$ but contained in a read. Even without repetitions, analyzing only k-mers instead of the longer reads can result in some information loss, since bases of a read that are more than k positions apart are not part of the same k-mer, but might be part of the same read. Indeed, differently from de Brujin graphs, any path of a string graph is a valid assembly of reads. On the other hand, string graphs are more computationally intensive to compute [24], justifying our search for faster algorithms. From an algorithmic point of view, the most used string graph assembler is SGA [23], which first constructs the BWT [11] and the FM-index of a set of strings, and then uses those data structures to efficiently compute the arcs of the string graph (connecting overlapping reads). Another string graph assembler is Fermi [17] which implements a variant of the original SGA algorithm [23] that is tailored for SNP and variant calling. A number of recent works face the problem of designing efficient algorithmic strategies or data structures for building string graphs. Among those works we can find a string graph assembler [4], based on a careful use of hashing and Bloom filters, with performance comparable with the first SGA implementation [23]. Another important alternative approach to SGA is Readjoiner [15] which is based on an efficient computation of a subset of exact suffix-prefix matches, and by subsequent rounds of suffix sorting, scanning, and filtering outputs the non-redundant arcs of the graph.

All assemblers based on string graphs (such as SGA) need to both (1) query an indexing data structures (such as an FM-index), and (2) access the original reads set to detect prefix-suffix overlaps between the elements. Since the self-indexing data structures, such as FM-index, represent the whole information of the original dataset, an interesting problem is to design efficient algorithms for the construction of string graphs that only require to keep the index and do not need to access the read set together with the index. Improvements in this direction have both theoretical and practical motivations. Indeed, detecting prefix-suffix overlaps only by analyzing the (compressed) index is an almost unexplored problem, and managing such data structure is usually more efficient.

Following this research direction, we propose a new algorithm, called FSG, to compute the string graph of a set R of reads, whose $O(nm)$ time complexity matches that of SGA—n is the number of reads in R and m is the maximum read length. To the best of our knowledge, it is the first algorithm that computes a string graph using only the FM-index of the input reads. The vast literature on BWT and FM-index hints that this approach is amenable to further research. An important observation is that SGA computes the string graph basically performing, for each read r, a query to the FM-index for each character of r, to compute the arcs outgoing from r. While this approach works in $O(nm)$ time, it can perform several redundant queries, most notably when the reads share common suffixes (a very common case). Our algorithm queries the FM-index in a specific order, so that each string is processed only once, while SGA might process more than once each repeated string. It is important to notice that our novel algorithm uses a characterization of a string graph that is different, but equivalent, to the one in [18] stated in [7] and which is quite useful when processing reads with their FM-index. Moreover, since we have integrated our algorithm into SGA, the read correction and the assembly phases of SGA can be applied without any modification. These facts guarantees that the assemblies produced by our approach and SGA are the same. In a previous paper, we have tackled the problem of constructing the string graph in external memory [8] by taking advantages of some recent results on the external memory implementation of the FM-index [2]. Experimental results [8] have revealed that computing the FM-index and LCP (Longest Common Prefix) array are the two main limiting factors towards an efficient (in terms of running time and main memory requirements) external memory algorithm to construct the string graph. In fact, even the best known algorithms for these steps do not have an optimal I/O complexity [2,3].

The FSG algorithm provides an approach to build a string graph that could be used for different read assembly purposes. We have implemented FSG and integrated it with the SGA assembler, by replacing in SGA the step related to the string graph construction. Our implementation follows the SGA guidelines, *i.e.*, we use the correction step of SGA before computing the overlaps without allowing mismatches (which is also SGA's default). Notice that SGA is a finely tuned implementation that has performed very nicely in the latest Assemblathon competition [10]. We have compared FSG with SGA, where we have used the latter's default parameter (that is, we compute overlaps without errors). Our experimental evaluation on a standard benchmark dataset shows that our approach is 2.3–4.8 times faster than SGA in terms of wall clock time.

2 Preliminaries

We briefly recall some standard definitions that will be used in the following. Let Σ be a constant-sized alphabet and let S be a string over Σ. We denote by $S[i]$ the i-th symbol of S, by $\ell = |S|$ the length of S, and by $S[i : j]$ the substring $S[i]S[i+1]\cdots S[j]$ of S. The *suffix* and *prefix* of S of length k are the substrings $S[\ell - k + 1 : \ell]$ (denoted by $S[\ell - k + 1 :]$) and $S[1 : k]$ (denoted by $S[: k]$) respectively. Given two strings (S_i, S_j), we say that S_i *overlaps* S_j iff a nonempty suffix β of S_i is also a prefix of S_j, that is $S_i = \alpha\beta$ and $S_j = \beta\gamma$. In this paper we consider a set R of n strings over Σ that are terminated by the sentinel \$, which is the smallest character. To simplify the exposition, we will assume that all input strings have exactly m characters, excluding the \$. The *overlap graph* of a set R of strings is the directed graph $G_O = (R, A)$ whose vertices are the strings in R, and each two overlapping strings $r_i = \alpha\beta$ and $r_j = \beta\gamma$ form the arc $(r_i, r_j) \in A$ labeled by α. In this case β is called the *overlap* of the arc and α is called the *extension* of the arc. Observe that the notion of overlap graph originally given by [18] is defined by labeling with γ the arc $(r_i, r_j) \in A$.

The notion of a string graph derives from the observation that in a overlap graph the label of an arc (r, s) may be obtained by concatenating the labels of a pair of arcs (r, t) and (t, s), thus arc (r, s) can be removed from the overlap graph without loss of information, since removing all such arcs, called *redundant* arcs, does not changet the set of valid paths. In [18] redundant arcs are those arcs (r, s) labeled by $\alpha\beta$, for α the prefix of an arc (r, t). An equivalent definition of string graphs is below. An arc $e_1 = (r_i, r_j)$ of G_O labeled by α is *transitive* (or *reducible*) if there exists another arc $e_2 = (r_k, r_j)$ labeled by δ where δ is a suffix of α [7]. Therefore, we say that e_1 is *non-transitive* (or *irreducible*) if no such arc e_2 exists. The string graph of R is obtained from G_O by removing all reducible arcs. This definition allows to use the FM-index to compute the labels of the string graph via backward extensions on the index.

The *Generalized Suffix Array (GSA)* [22] of R is the array SA where each element $SA[i]$ is equal to (k, j) iff the k-long suffix $r_j[|r_j| - k + 1 :]$ of the string r_j is the i-th smallest element in the lexicographic ordered set of all suffixes of the strings in R. The *Burrows-Wheeler Transform (BWT)* of R is the sequence B such that $B[i] = r_j[|r_j| - k]$, if $SA[i] = (k, j)$ and $k > 1$, or $B[i] = \$$, otherwise. Informally, $B[i]$ is the symbol that precedes the k-long suffix of a string r_j where such suffix is the i-th smallest suffix in the ordering given by SA. For any string ω, all suffixes of (the lexicographically sorted) SA whose prefix is ω appear consecutively in SA. Consequently, we define the ω-interval [2], denoted by $q(\omega)$, as the maximal interval $[b, e]$ such that $b \leq e$, $SA[b]$ and $SA[e]$ both have prefix ω. Notice that the width $e - b + 1$ of the ω-interval is equal to the number of occurrences of ω in some read of R. Since the BWT B and SA are closely related, we also say that $[b, e]$ is a ω-interval on B. Given a ω-interval and a character c, the *backward c-extension* of the ω-interval is the $c\omega$-interval.

3 The Algorithm

Our algorithm is based on two steps: the first is to compute the overlap graph, the second is to remove all transitive arcs. Given a string ω and R a set of strings (reads), let $R^S(\omega)$ and $R^P(\omega)$ be respectively the subset of R with suffix (resp. prefix) ω. As usual in string graph construction algorithms, we will assume that the set R is *substring free*, *i.e.*, no string is a substring of another. A fundamental observation is that the list of all nonempty overlaps β is a compact representation of the overlap graph, since all pairs in $R^S(\beta) \times R^P(\beta)$ are arcs of the overlap graph. Our approach to compute all overlaps between pairs of strings is based on the notion of *potential overlap*, which is a nonempty string $\beta^* \in \Sigma^+$, s.t. there exists at least one input string $r_i = \alpha\beta^*$ $(\alpha \neq \epsilon)$ with suffix β^*, and there exists at least one input string $r_j = \gamma\beta^*\delta$ $(\delta \neq \epsilon)$ with β^* as a substring (possibly a prefix). The first part of Algorithm 1 (lines 3–11) computes all potential overlaps, starting from those of length 1 and extending the potential overlaps by adding a new leading character. For each potential overlap, we check if it is an actual overlap. Lemma 1 is a direct consequence of the definition of potential overlap.

Lemma 1. *Let β be an overlap. Then all suffixes of β are potential overlaps.*

The second part of our algorithm, that is to detect all transitive arcs, can be sped up if we cluster together and examine some sets of arcs. We start considering the set of all arcs sharing the same overlap and a suffix of their extensions, as stated in the following definition.

Definition 2. *Assume that $\alpha, \beta \in \Sigma^*, \beta \neq \epsilon$ and $X \subseteq R^P(\beta)$. The* arc-set *$ARC(\alpha, \alpha\beta, X)$ is the set $\{(r_1, r_2) : \alpha\beta$ is a suffix of r_1, β is a prefix of r_2, and $r_1 \in R, r_2 \in X\}$. The strings α and β are called the* extension *and the* overlap *of the arc-set. The set X is called the* destination *set of the arc-set.*

In other words, an arc-set contains the arcs with overlap β and extension α. An arc-set is *terminal* if there exists $r \in R$ s.t. $r = \alpha\beta$, while an arc-set is *basic* if $\alpha = \epsilon$ (the empty string). Since the arc-set $ARC(\alpha, \alpha\beta, X)$ is uniquely determined by strings α, $\alpha\beta$, and X, the triple $(\alpha, \alpha\beta, X)$ encodes the arc-set $ARC(\alpha, \alpha\beta, X)$. Moreover, the arc-set $ARC(\alpha, \alpha\beta, X)$ is *correct* if X includes all irreducible arcs that have overlap β and extension with suffix α, that is $X \supseteq \{r_2 \in R^P(\beta) : r_1 \in R^S(\alpha\beta)$ and (r_1, r_2) is irreducible$\}$. Observe that our algorithm computes only correct arc-sets. Moreover, terminal arc-sets only contain irreducible arcs (Lemma 5). Lemma 3 shows the use of arc-sets to detect transitive arcs. Due to space constraints, all proofs are omitted.

Lemma 3. *Let (r_1, r_2) be an arc with overlap β. Then (r_1, r_2) is transitive iff (i) there exist $\alpha, \gamma, \delta, \eta \in \Sigma^*$, $\gamma, \eta \neq \epsilon$ such that $r_1 = \gamma\alpha\beta$, $r_2 = \beta\delta\eta$, (ii) there exists an input read $r_3 = \alpha\beta\delta$ such that (r_3, r_2) is an irreducible arc of a nonempty arc-set $ARC(\alpha, \alpha\beta\delta, X)$.*

A direct consequence of Lemma 3 is that a nonempty correct terminal arc-set $ARC(\alpha, \alpha\beta\delta, X)$ implies that all arcs of the form $(\gamma\alpha\beta, \beta\delta\eta)$, with $\gamma, \eta \neq \epsilon$ are transitive. Another consequence of Lemma 3 is that an irreducible arc $(\alpha\beta\delta, \beta\delta\eta)$ with extension α and overlap $\beta\delta$ reduces all arcs with overlap β and extension $\gamma\alpha$, with $\gamma \neq \epsilon$. Lemma 3 is the main ingredient used in our algorithm. More precisely, it computes terminal correct arc-sets of the form $ARC(\alpha, \alpha\beta\delta, X)$ for extensions α of increasing length. By Lemma 3, $ARC(\alpha, \alpha\beta\delta, X)$ contains arcs that reduce all the arcs contained in $ARC(\alpha, \alpha\beta, X')$ which have a destination in X. Since the transitivity of an arc is related to the extension α of the arc that is used to reduce it, and our algorithm considers extensions of increasing length, a main consequence of Lemma 3 is that it computes terminal arc-sets that are correct, that is they contain only irreducible arcs. We will further speed up the computation by clustering together the arc-sets sharing the same extension.

Definition 4. *Let T be a set of arc-sets, and let α be a string. The* cluster *of α, denoted by $C(\alpha)$, is the union of all arc-sets of T whose extension is α.*

We sketch Algorithm 1 which consists of two phases: the first phase to compute the overlap graph, and the second phase to remove all transitive arcs. In our description, we assume that, given a string ω, we can compute in constant time (1) the number $\mathsf{suff}(\omega)$ of input strings whose suffix is ω, (2) the number $\mathsf{pref}(\omega)$ of input strings whose prefix is ω, (3) the number $\mathsf{substr}(\omega)$ of occurrences of ω in the input strings. Moreover, we assume to be able to list the set $\mathsf{listpref}(\omega)$ of input strings with prefix ω in $O(|\mathsf{listpref}(\omega)|)$ time. In Sect. 4 we will describe such a data structure. The first phase (lines 3–11) exploits Lemma 1 to compute all overlaps. Potential overlaps are defined inductively. The empty string ϵ is a potential overlap of length 0; given an i-long potential overlap β^*, the $(i+1)$-long string $c\beta^*$, for $c \in \Sigma$, is a potential overlap iff $\mathsf{suff}(c\beta^*) > 0$ and $\mathsf{substr}(c\beta^*) > \mathsf{suff}(c\beta^*)$. Our algorithm uses this definition to build potential overlaps of increasing length, starting from those with length 1, *i.e.*, symbols of Σ (line 2). The lists *Last* and *New* store the potential overlaps computed at the previous and current iteration respectively. Observe that a potential overlap β^* is an overlap iff $\mathsf{pref}(\beta^*) > 0$. Since a potential overlap is a suffix of some input string, there are at most nm distinct suffixes, where m and n are the length and the number of input strings, respectively. Each query $\mathsf{suff}(\cdot), \mathsf{pref}(\cdot), \mathsf{substr}(\cdot)$ requires $O(1)$ time, thus the time complexity related to the total number of such queries is $O(nm)$. Given two strings β_1 and β_2, when $|\beta_1| = |\beta_2|$ no input string can be in both $\mathsf{listpref}(\beta_1)$ and $\mathsf{listpref}(\beta_2)$. Since each overlap is at most m long, the overall time spent in the $\mathsf{listpref}(\cdot)$ queries is $O(nm)$. The first phase produces (line 7) the set of disjoint *basic* arc-sets $ARC(\epsilon, \beta, R^p(\beta))$ for each overlap β, whose union is the set of arcs of the overlap graph. Recall that $\mathsf{listpref}(\beta)$ gives the set of reads with prefix β, which has been denoted by $R^p(\beta)$.

The second phase (lines 13–25) classifies the arcs of the overlap graph into reducible or irreducible by computing arc-sets of increasing extension length, starting from the basic arc-sets $ARC(\epsilon, \epsilon\beta, R^p(\beta))$ obtained in the previous phase. By Lemma 3, we compute all correct terminal arc-sets $ARC(\alpha, \alpha\beta, X)$ and remove all arcs that are reduced by $ARC(\alpha, \alpha\beta, X)$. The set *Rdc* is used to store the destination set X of the computed terminal arc-sets. Notice that if $ARC(\alpha, \alpha\beta, X)$ is terminal, then all of its arcs have the same origin $r = \alpha\beta$, *i.e.*, $ARC(\alpha, \alpha\beta, X) = \{(r, x) : x \in X\}$. By Lemma 3 all arcs in the cluster $C(\alpha)$ with a destination in X and with an origin different from r are transitive and can be removed, simply by removing X from all destination sets in the arc-sets of $C(\alpha)$. Another application of Lemma 3 is that when we find a terminal arc-set all of its arcs are irreducible, *i.e.*, it is also correct. In fact, Lemma 3 classifies an arc as transitive according to the existence of a read $r = \alpha\beta$ with extension α. Since the algorithm considers extensions α of increasing length, all arcs whose extensions is shorter than α have been reduced in a previous step, thus all terminal arc-set of previous iterations are irreducible. More precisely, the test at line 18 is true iff the current arc-set is terminal. In that case, at line 19 all arcs of the arc-set are output as arcs of the string graph, and at line 20 the destination set X is added to the set *Rdc* that contains the destinations of $C(\alpha)$ that must be removed. For each cluster $C(\alpha)$, we read twice all arc-sets that are included in $C(\alpha)$. The first time to determine which arc-sets are terminal and, in that case, to determine the set *Rdc* of reads that must be removed from all destinations of the arc-sets included in $C(\alpha)$. The second time to compute the clusters $C(c\alpha)$ that contain the nonempty arc-sets with extension $c\alpha$ consisting of the arcs that we still have to check if they are transitive or not (that is the arcs with destination set $X \setminus Rdc$). In Algorithm 1, the cluster $C(\alpha)$ that is currently analyzed is stored in *CurrentCluster*, that is a list of the arc-sets included in the cluster. Moreover, the clusters that still have to be analyzed are stored in the stack *Clusters*. We use a stack to guarantee that the clusters are analyzed in the correct order, that is the cluster $C(\alpha)$ is analyzed after all clusters $C(\alpha[i:])$—$\alpha[i:]$ is a generic suffix

Algorithm 1. Compute the string graph

```
    Input  : The set $R$ of input strings
    Output : The string graph of $R$, given as a list of arcs
 1  Cluster ← empty list;
 2  Last ← $\{c \in \Sigma \mid \mathsf{suff}(c) > 0$ and $\mathsf{substr}(c) > \mathsf{suff}(c)\}$;
 3  while Last is not empty do
 4      New ← $\varnothing$;
 5      foreach $\beta^* \in$ Last do
 6          if $\mathsf{pref}(\beta^*) > 0$ then
 7              Append $(\epsilon, \beta^*, \mathsf{listpref}(\beta^*))$ to Cluster;
 8          for $c \in \Sigma$ do
 9              if $\mathsf{suff}(c\beta^*) > 0$ and $\mathsf{substr}(c\beta^*) > \mathsf{suff}(c\beta^*)$ then
10                  Add $c\beta^*$ to New;
11      Last ← New;
12  Clusters ← the stack with Cluster as its only element;
13  while Clusters is not empty do
14      CurrentCluster ← Pop(Clusters);
15      Rdc ← $\varnothing$;
16      Let ExtendedClusters be an array of $|\Sigma|$ empty clusters;
17      foreach $(\alpha, \alpha\beta, X) \in$ CurrentCluster do
18          if $\mathsf{substr}(\alpha\beta) = \mathsf{pref}(\alpha\beta) = \mathsf{suff}(\alpha\beta) > 0$ then
19              Output the arcs $(\alpha\beta, x)$ with label $\alpha$ for each $x \in X$;
20              Rdc ← Rdc $\cup X$;
21      foreach $(\alpha, \alpha\beta, X) \in$ CurrentCluster do
22          if $X \nsubseteq Rdc$ then
23              for $c \in \Sigma$ do
24                  if $\mathsf{suff}(c\alpha\beta) > 0$ then
25                      Append $(c\alpha, c\alpha\beta, X \setminus \text{Rdc})$ to ExtendedClusters[$c$];
26      Push each non-empty cluster of ExtendedClusters to Clusters;
```

of α. We can prove that a generic irreducible arc (r_1, r_2) with extension α and overlap β belongs exactly to the clusters $C(\epsilon), \ldots, C(\alpha[2:]), C(\alpha)$. Moreover, r_2 does not belong to the set *Rdc* when considering $C(\epsilon), \ldots, C(\alpha[2:])$, hence the arc (r_1, r_2) is correctly output when considering the cluster $C(\alpha)$. The lemmas leading to the correctness of the algorithm follow.

Lemma 5. *Let $ARC(\alpha, \alpha\beta, X)$ be an arc-set inserted into a cluster by Algorithm 1. Then such arc-set is correct.*

Lemma 6. *Let e_1 be a transitive arc (r_1, r_2) with overlap β. Then the algorithm does not output e_1.*

Theorem 7. *Given as input a set of strings R, Algorithm 1 computes exactly the arcs of the string graph.*

We can now sketch the time complexity of the second phase. Previously, we have shown that the first phase produces at most $O(nm)$ arc-sets, one for each distinct overlap β. Since each string $\alpha\beta$ considered in the second phase is a suffix of an input string, and there are at most nm such suffixes, at most nm arc-sets are considered in the second phase. In the second phase, for each cluster a set Rdc is computed. If Rdc is empty, then each arc-set of the cluster can be examined in constant time, since all unions at line 20 are trivially empty and at line 25 the set $X \setminus Rdc$ is equal to X, therefore no operation must be computed. The interesting case is when $X \neq \varnothing$ for some arc-set. In that case the union at line 20 and the difference $X \setminus Rdc$ at line 25 are computed. Let $d(n)$ be the time complexity of those two operations on n-element sets (the actual time complexity depends on the data structure used). Notice that X is not empty only if we have found an irreducible arc, that is an arc of the string graph. Overall, there can be at most $|E|$ nonempty such sets X, where E is the set of arcs of the string graph. Hence, the time complexity of the entire algorithm is $O(nm + |E|d(n))$.

4 Data Representation

Our algorithm entirely operates on the (potentially compressed) FM-index of the collection of input reads. Indeed, each processed string ω (both in the first and in the second phase) can be represented in constant space by the ω-interval $[b_\omega, e_\omega]$ on the BWT (*i.e.*, $\mathsf{q}(\omega)$), instead of using the naïve representation with $O(|\omega|)$ space. Notice that in the first phase, the i-long potential overlaps, for a given iteration, are obtained by prepending a symbol $c \in \Sigma$ to the $(i-1)$-long potential overlaps of the previous iteration (lines 8–10). In the same way the arc-sets of increasing extension length are computed in the second phase. In other words, our algorithm needs in general to obtain string $c\omega$ from string ω, and, since we represent strings as intervals on the BWT, this operation can be performed in $O(1)$ time via backward c-extension of the interval $\mathsf{q}(\omega)$ [14].

Moreover, both queries $\mathsf{pref}(\omega)$ and $\mathsf{substr}(\omega)$ can be answered in $O(1)$ time. In fact, given $\mathsf{q}(\omega) = [b_\omega, e_\omega]$, then $\mathsf{substr}(\omega) = e_\omega - b_\omega + 1$ and $\mathsf{pref}(\omega) = e_{\$\omega} - b_{\$\omega} + 1$ where $\mathsf{q}(\$\omega) = [b_{\$\omega}, e_{\$\omega}]$ is the result of the backward \$-extension of $\mathsf{q}(\omega)$. Similarly, it is easy to compute $\mathsf{listpref}(\omega)$ as it corresponds to the set of reads that have a suffix in the interval $\mathsf{q}(\$\omega)$ of the GSA. The interval $\mathsf{q}(\omega\$) = [b_{\omega\$}, e_{\omega\$}]$ allows to answer to the query $\mathsf{suff}(\omega)$ which is computed as $e_{\omega\$} - b_{\omega\$} + 1$. The interval $\mathsf{q}(\omega\$)$ is maintained along with $\mathsf{q}(\omega)$. Moreover, since $\mathsf{q}(\omega\$)$ and $\mathsf{q}(\omega)$ share the lower extreme $b_\omega = b_{\omega\$}$ (recall that \$ is the smallest symbol), each string ω can be compactly represented by the three integers $b_\omega, e_{\omega\$}, e_\omega$. While in our algorithm a substring ω of some input read can be represented by those three integers, we exploited the following representation for greater efficiency. In the first phase of the algorithm we mainly have to represent the set of potential overlaps. At each iteration, the potential overlaps in *Last* (*New*, resp.) have the same length, hence their corresponding intervals on the BWT are disjoint. Hence we can store those intervals using a pair of $n(m+1)$-long bitvectors. For each potential overlap $\beta \in \mathit{Last}$ (*New*, resp.) represented by the β-interval $[b_\beta, e_\beta]$, the first bitvector has 1 in position b_β and the second bitvector has 1 in positions $e_{\beta\$}$ and e_β. Recall that we want also to maintain the interval $q(\beta\$) = [b_\beta, e_{\beta\$}]$. Since

$\mathsf{substr}(\beta) > \mathsf{suff}(\beta)$, then $e_{\beta\$} \neq e_\beta$ and can be stored in the same bitvector. In the second phase of the algorithm, we mainly represent clusters. A cluster groups together arc-sets whose overlaps are pairwise different or one is the prefix of the other. Thus, the corresponding intervals on the BWT are disjoint or nested. Moreover, also the destination set of the *basic* arc-sets can be represented by a set of pairwise disjoint or nested intervals on the BWT (since $\mathsf{listpref}(\beta)$ of line 7 correspond to the interval $\mathsf{q}(\$\beta)$). Moreover, the loop at lines 13–25 preserves the following invariant: let $ARC(\alpha, \alpha\beta_1, X_1)$ and $ARC(\alpha, \alpha\beta_2, X_2)$ be two arc-sets of the same cluster $C(\alpha)$ with β_1 prefix of β_2, then $X_2 \subseteq X_1$. Hence, each subset of arc-sets whose extensions plus overlaps share a common nonempty prefix γ is represented by means of the following three vectors: two integers vectors V_b, V_e of length $e_\gamma - b_\gamma + 1$ and a bitvector B_x of length $e_{\$\gamma} - b_{\$\gamma} + 1$, where $[b_\gamma, e_\gamma]$ is the γ-interval and $[b_{\$\gamma}, e_{\$\gamma}]$ is the $\$\gamma$-interval. More specifically, $V_b[i]$ ($V_e[i]$, resp.) is the number of arc-sets whose representation (BWT interval) of the overlap starts (ends, resp.) at $b_\gamma + i$, while $B_x[i]$ is 1 iff the read at position $b_{\$\gamma} + i$, in the lexicographic order of the GSA, belongs to the destination set of all the arc-sets. As a consequence, the number of backward extensions performed by Algorithm 1 is at most $O(nm)$, while SGA performs $\Theta(nm)$ extensions.

5 Experimental Analysis

A C++ implementation of our approach, called FSG (short for Fast String Graph), has been integrated in the SGA suite and is available at http://fsg.algolab.eu under the GPLv3 license. We have evaluated the performance of FSG on a standard benchmark of 875 million 101 bp-long reads sequenced from the NA12878 individual of the International HapMap and 1000 genomes project and comparing the running time of FSG with SGA. We have run SGA with its default parameters, that is SGA has compute exact overlaps after having corrected the input reads. Since the string graphs computed by FSG and SGA are the same, we have not compared the entire pipeline, but only the string graph construction phase. We could not compare FSG with Fermi, since Fermi does not split its steps in a way that allows to isolate the running time of the string graph construction—most notably, it includes reads correction and scaffolding.

Especially on the DNA alphabet, short overlaps between reads may happen by chance. Hence, for genome assembly purposes, only overlaps whose length is larger than a user-defined threshold are considered. The value of the minimum overlap length threshold that empirically showed the best results in terms of genome assembly quality is around the 75 % of the read length [24]. To assess how graph size affects performance, different values of minimum overlap length (called τ) between reads have been used (clearly, the lower this value, the larger the graph). The minimum overlap lengths used in this experimental assessment are 55, 65, 75, and 85, hence the chosen values test the approaches also on larger-than-normal ($\tau = 55$) and smaller-than-normal ($\tau = 85$) string graphs. Another aspect that we have wanted to measure is the scalability of FSG. We have run the programs with 1, 4, 8, 16, and 32 threads. In all cases, we have measured

Table 1. Comparison of FSG and SGA, for different minimum overlap lengths and numbers of threads. The wall-clock time is the time used to compute the string graph. The CPU time is the overall execution time over all CPUs actually used.

Min. overlap	No. of threads	Wall time [min]			Work time [min]		
		FSG	SGA	$\frac{\text{FSG}}{\text{SGA}}$	FSG	SGA	$\frac{\text{FSG}}{\text{SGA}}$
55	1	1,485	4,486	0.331	1,483	4,480	0.331
	4	474	1,961	0.242	1,828	4,673	0.391
	8	318	1,527	0.209	2,203	4,936	0.446
	16	278	1,295	0.215	3,430	5,915	0.580
	32	328	1,007	0.326	7,094	5,881	1.206
65	1	1,174	3,238	0.363	1,171	3,234	0.363
	4	416	1,165	0.358	1,606	3,392	0.473
	8	271	863	0.315	1,842	3,596	0.512
	16	255	729	0.351	3,091	4,469	0.692
	32	316	579	0.546	6,690	4,444	1.505
75	1	1,065	2,877	0.37	1,063	2,868	0.371
	4	379	915	0.415	1,473	2,903	0.507
	8	251	748	0.336	1,708	3,232	0.528
	16	246	561	0.439	2,890	3,975	0.727
	32	306	455	0.674	6,368	4,062	1.568
85	1	1,000	2,592	0.386	999	2,588	0.386
	4	360	833	0.432	1,392	2,715	0.513
	8	238	623	0.383	1,595	3,053	0.523
	16	229	502	0.457	2,686	3,653	0.735
	32	298	407	0.733	6,117	3,735	1.638

the elapsed (wall-clock) time and the total CPU time (the time a CPU has been working). All experiments have been performed on an Ubuntu 14.04 server with four 8-core Intel® Xeon E5-4610v2 2.30 GHz CPUs. The server has a NUMA architecture with 64 GiB of RAM for each node (256 GiB in total).

Table 1 summarizes the running times of both approaches on the different configurations of the parameters. Notice that LSG approach is from 2.3 to 4.8 times faster than SGA in terms of wall-clock time and from 1.9 to 3 times in terms of CPU time. On the other hand, FSG uses approximately 2.2 times the memory used by SGA—on the executions with at most 8 threads. On a larger number of threads, and in particular the fact that the elapsed time of FSG on 32 threads is larger than that on 16 threads suggests that, in its current form, FSG might not be suitable for a large number of threads. However, since the current implementation of FSG is almost a proof of concept, future improvements to its codebase and a better analysis of the race conditions of our tool will likely lead to better performances with a large number of threads. Furthermore, notice that also the SGA algorithm, which is (almost) embarrassingly parallel and has a stable implementation, does not achieve a speed-up better than 6.4 with 32 threads. As such, a factor that likely contributes to a poor scaling behaviour of both FSG and SGA could be also the NUMA architecture of the server used for the experimental analysis, which makes different-unit memory accesses more expensive (in our case, the processors in each unit can manage at most 16 logical threads, and only 8 on physical cores). FSG uses more memory than SGA since genome assemblers must correctly manage reads extracted from both strands of the genome. In our case, this fact has been addressed by adding each reverse-and-complement read to the set of strings on which the FM-index has been built, hence immediately doubling the size of the FM-index. Moreover, FSG needs some additional data structures to correctly maintain potential overlaps and arc-sets: two pairs of $n(m+1)$-long bitvectors and the combination of two (usually) small integer vectors and a bitvector of the same size. Our experimental evaluation shows that the memory required by the latter is usually negligible, hence a better implementation of the four bitvectors could decrease the memory use. The main goal of FSG is to improve the running time, not the memory use.

The combined analysis of the CPU time and the wall-clock time on at most 8 threads (which is the number of physical cores of each CPU on our server) suggests that FSG is more CPU efficient than SGA and is able to better distribute the workload across the threads. In our opinion, our greater efficiency is achieved by operating only on the FM-index of the input reads and by the order on which extension operations (*i.e.*, considering a new string $c\alpha$ after α has been processed) are performed. These two characteristics of our algorithm allow to eliminate the redundant queries to the index which, instead, are performed by SGA. In fact, FSG considers each string that is longer than the threshold at most once, while SGA potentially reconsiders the same string once for each read in which the string occurs. Indeed, FSG uses 2.3–3 times less user time than SGA when $\tau = 55$ (hence, when such sufficiently-long substrings occur more frequently) and "only" 2–2.6 times less user time when $\tau = 85$ (hence, when such sufficiently-long substrings are more rare).

6 Conclusions and Future Work

We present FSG: a tool implementing a new algorithm for constructing a string graph that works directly querying a FM-index representing a collection of reads, instead of processing the input reads. Our main goal is to provide a simpler and fast algorithm to construct string graphs, so that its implementation can be easily integrated into an assembly pipeline that analyzes the paths of the string graph to produce the final assembly. Indeed, FSG could be used for related purposes, such as transcriptome assembly [5,16], and haplotype assembly [6]. These topics are some of the research directions that we plan to investigate.

References

1. Bankevich, A., Nurk, S., et al.: SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J. Comput. Biol. **19**(5), 455–477 (2012)
2. Bauer, M., Cox, A., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theoret. Comput. Sci. **483**, 134–148 (2013)
3. Bauer, M.J., Cox, A.J., Rosone, G., Sciortino, M.: Lightweight LCP construction for next-generation sequencing datasets. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 326–337. Springer, Heidelberg (2012)
4. Ben-Bassat, I., Chor, B.: String graph construction using incremental hashing. Bioinformatics **30**(24), 3515–3523 (2014)
5. Beretta, S., Bonizzoni, P., Della Vedova, G., Pirola, Y., Rizzi, R.: Modeling alternative splicing variants from RNA-Seq data with isoform graphs. J. Comput. Biol. **16**(1), 16–40 (2014)
6. Bonizzoni, P., Della Vedova, G., Dondi, R., Li, J.: The haplotyping problem: an overview of computational models and solutions. J. Comput. Sci. Technol. **18**(6), 675–688 (2003)
7. Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M., Rizzi, R.: Constructing string graphs in external memory. In: Brown, D., Morgenstern, B. (eds.) WABI 2014. LNCS, vol. 8701, pp. 311–325. Springer, Heidelberg (2014)
8. Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M., Rizzi, R.: LSG: an external-memory tool to compute string graphs for NGS data assembly. J. Comp. Biol. **23**(3), 137–149 (2016)
9. Boucher, C., Bowe, A., Gagie, T., et al.: Variable-order de bruijn graphs. In: 2015 Data Compression Conference (DCC), pp. 383–392. IEEE (2015)
10. Bradnam, K.R., Fass, J.N., Alexandrov, A., et al.: Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. GigaScience **2**(1), 1–31 (2013)
11. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center (1994)
12. Chikhi, R., Limasset, A., Jackman, S., Simpson, J.T., Medvedev, P.: On the representation of de bruijn graphs. J. Comp. Biol. **22**(5), 336–352 (2015)
13. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Alg. Mol. Biol. **8**(22), 1–9 (2013)
14. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
15. Gonnella, G., Kurtz, S.: Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC Bioinform. **13**(1), 82 (2012)
16. Lacroix, V., Sammeth, M., Guigo, R., Bergeron, A.: Exact transcriptome reconstruction from short sequence reads. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 50–63. Springer, Heidelberg (2008)
17. Li, H.: Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. Bioinformatics **28**(14), 1838–1844 (2012)
18. Myers, E.: The fragment assembly string graph. Bioinformatics **21**(s2), 79–85 (2005)
19. Peng, Y., Leung, H.C., Yiu, S.-M., Chin, F.: IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. Bioinformatics **28**(11), 1420–1428 (2012)
20. Salikhov, K., Sacomoto, G., Kucherov, G.: Using cascading bloom filters to improve the memory usage for de brujin graphs. Alg. Mol. Biol. **9**(1), 2 (2014)

21. Salzberg, S.L., et al.: GAGE: a critical evaluation of genome assemblies and assembly algorithms. Genome Res. **22**(3), 557–567 (2012)
22. Shi, F.: Suffix arrays for multiple strings: a method for on-line multiple string searches. In: Jaffar, J., Yap, R.H.C. (eds.) ASIAN 1996. LNCS, vol. 1179, pp. 11–22. Springer, Heidelberg (1996)
23. Simpson, J., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics **26**(12), i367–i373 (2010)
24. Simpson, J., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Res. **22**, 549–556 (2012)
25. Simpson, J., Wong, K., Jackman, S., et al.: ABySS: a parallel assembler for short read sequence data. Genome Res. **19**(6), 1117–1123 (2009)