BIOINFORMATICS

Vol. 26 ISMB 2010, pages i367–i373
doi:10.1093/bioinformatics/btq217

Efficient construction of an assembly string graph using the FM-index

Jared T. Simpson* and Richard Durbin

Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus, Cambridge, CB10 1SA, UK

ABSTRACT

Motivation: Sequence assembly is a difficult problem whose importance has grown again recently as the cost of sequencing has dramatically dropped. Most new sequence assembly software has started by building a de Bruijn graph, avoiding the overlap-based methods used previously because of the computational cost and complexity of these with very large numbers of short reads. Here, we show how to use suffix array-based methods that have formed the basis of recent very fast sequence mapping algorithms to find overlaps and generate assembly string graphs asymptotically faster than previously described algorithms.

Results: Standard overlap assembly methods have time complexity $O(N^2)$, where N is the sum of the lengths of the reads. We use the Ferragina–Manzini index (FM-index) derived from the Burrows–Wheeler transform to find overlaps of length at least τ among a set of reads. As well as an approach that finds all overlaps then implements transitive reduction to produce a string graph, we show how to output directly only the irreducible overlaps, significantly shrinking memory requirements and reducing compute time to $O(N)$, independent of depth. Overlap-based assembly methods naturally handle mixed length read sets, including capillary reads or long reads promised by the third generation sequencing technologies. The algorithms we present here pave the way for overlap-based assembly approaches to be developed that scale to whole vertebrate genome *de novo* assembly.

Contact: js18@sanger.ac.uk

1 INTRODUCTION

The sequence assembly problem is one of the most important and difficult problems in bioinformatics. Most genomes, particularly eukaryotic genomes, are highly repetitive that complicates their assembly by obscuring true relationships between reads with many false options. To help disambiguate the true relationships between the reads from those induced by different copies of repeats, it is useful to construct a graph where all the copies of a repeat are collapsed into a single segment. Such a graph is commonly referred to as a *repeat graph*. This structure is a natural consequence of the de Bruijn graph method of sequence assembly as the deconstruction of the sequence reads into k-mers (short subsequences of the reads of length k) collapses repeats that share the same k-mer into a single vertex (Pevzner *et al.*, 2001). An alternative formulation was proposed by Gene Myers and is called the *string graph* (Myers, 2005). The string graph is built by first constructing a graph of the pairwise overlaps between sequence reads and transforming it into a string graph by removing transitive edges. The string graph shares with the de Bruijn graph the property that repeats are collapsed to a single unit without the need to first deconstruct the reads into k-mers. Because it is based on maximal overlaps, which are typically longer than de Bruijn k-mers, it also disambiguates shorter repeats that de Bruijn methods would only resolve in later processing steps (if at all). The string graph is much more expensive to construct, however, as the set of all pairwise, inexact overlaps between sequence reads must be found. For this reason, the majority of assemblers of short read sequence data have been based on the de Bruijn approach (Chaisson and Pevzner, 2008; Simpson *et al.*, 2009; Zerbino and Birney, 2008). A notable exception is the Edena assembler (Hernandez *et al.*, 2008) that uses a suffix array to compute exact overlaps between reads that are then used to construct the string graph. We address the construction of a string graph with a related approach by indexing the set of sequence reads using the Burrows–Wheeler transform(BWT)/Ferragina—Manzini(FM)-index, which has recently been used for the short read alignment problem (Langmead *et al.*, 2009; Li and Durbin, 2009; Li *et al.*, 2009). We show how to efficiently compute the set of overlaps needed to construct the string graph from the FM-index. Furthermore, we show that the string graph can be constructed directly using the FM-index without the need for explicitly finding all overlaps and a subsequent transitive removal step, yielding a space and time efficient construction algorithm.

2 BACKGROUND

2.1 Definitions and notation

Let X be a string of symbols $a_1,...,a_l$ from an alphabet Σ. The length of X is denoted $|X|$. We consider all strings to be terminated by a sentinel symbol \$ that is not in Σ and is lexographically lower than all the symbols in Σ. $X[i]=a_i$ is the i-th symbol of X and $X[i,j]$ is the substring $a_i,...,a_j$. A substring $X[k,|X|]$ is a *suffix* of X and a substring $X[1,k]$ is a *prefix* of X. Let $X'=a_l,a_{l-i},...,a_1$ denote the reverse of X.

2.2 Genomes and sequence reads

We define a *genome* to be a long string from the alphabet $\{A,C,G,T\}$ representing the complete DNA sequence of an individual, for simplicity ignoring potential subdivisions into chromosomes. A sequence *read* is a short substring from a genome. DNA is a double stranded molecule and sequence reads can originate from either strand. We use the notation $\overline{X}$ for the *reverse-complement* of a read X. In a shotgun sequencing experiment, a set of sequence reads, which we denote by the indexed set $\mathcal{R}$, is randomly sampled from a genome with an unknown sequence. The sequence assembly problem is to reconstruct the sequence of the genome given $\mathcal{R}$. We say that two reads X and Y *overlap* if a prefix of X is equal to a suffix of Y or vice versa. If X and Y originate from opposite strands, they overlap if the reverse complement of one of them

*To whom correspondence should be addressed.

overlaps the other. To help distinguish true overlaps from spurious overlaps, we set a threshold of τ on the minimum acceptable overlap length. We assume for the moment that sequence reads are perfect representations of the genome—there are no sequencing errors. We discuss how to relax this constraint in the discussion at the end of this article.

2.3 Overlap and string graphs

To help reconstruct the source genome from $\mathcal{R}$, we can build a graph of the relationships between sequence reads. One such graph is the *overlap graph*. In the overlap graph, each sequence read in $\mathcal{R}$ is a vertex and two vertices are joined by an edge if their corresponding reads overlap. Myers' string graph is a refinement of such a graph. In the string graph, reads that are *contained* within some other read, that is they are a substring of (or perhaps identical to) another read, are considered to be redundant and are not vertices in the graph. Each edge in a string graph is bidirectional to model the double-stranded nature of DNA and labelled with the unmatched substrings of the sequence reads. More formally, let X and Y be two reads where $X[s_{xy},e_{xy}]=Y[s_{yx},e_{yx}]$. We call $X[s_{xy},e_{xy}]$ the *matched* portion of X and the remainder *unmatched*. If $s_{xy}=1$ and $e_{xy}=|X|$ the entirety of X is matched by Y and X is said to be contained by Y. If Y is also contained by X ($s_{yx}=1$ and $e_{yx}=|Y|$), X and Y are identical. In this case, we break the tie by saying the read with the higher index in $\mathcal{R}$ is contained within the read with the lower index. If neither X nor Y are contained and $X[s_{xy},e_{xy}]$ is a prefix of X ($s_{xy}=1$) and $Y[s_{yx},e_{yx}]$ is a suffix of Y ($e_{yx}=|Y|$), or vice versa, we say the overlap between X and Y is *proper*. If X and Y are reads from opposite strands of the genome they can still form an overlap. In this case, $\overline{X[s_{xy},e_{xy}]}=Y[s_{yx},e_{yx}]$ and both $X[s_{xy},e_{xy}]$ and $Y[s_{yx},e_{yx}]$ must be prefixes or both must be suffixes.

All non-contained reads are vertices in the string graph. For each proper overlap between two reads, we add a bidirected edge to the graph $X \leftrightarrow Y$. The bidirected edge describes the nature of the overlap between the reads and has two labels, one for each of the unmatched substring of the reads. We denote the tuple of data for each edge as $(type_{xy}, type_{yx}, label_{xy}, label_{yx})$. We define the $type_{xy}$ property (respectively, $type_{yx}$) as:

$$type_{xy}=\begin{cases} B & \text{if } s_{xy}=1 \\ E & \text{if } e_{xy}=|X| \end{cases}$$

In other words, $type_{xy}$ is B if the matched portion of X is a prefix of X, otherwise the matched portion of X must be a suffix and $type_{xy}$ is E. Note that since the graph does not have contained reads these cases are mutually exclusive. The $label_{xy}$ property is

$$label_{xy}=\begin{cases} Y[e_{yx}+1,|Y|] & \text{if } s_{yx}=1 \\ Y[1,s_{yx}-1] & \text{if } e_{yx}=|Y| \end{cases}$$

Restated, $label_{xy}$ is the unmatched suffix of Y if the matched portion of Y is a prefix and vice versa. The concatenation of X and $label_{xy}$ is an assembly of reads X and Y — the resulting string contains both the sequence of X and Y. If the overlap between X and Y is reverse complemented, i.e. $\overline{X[s_{xy},e_{xy}]}=Y[s_{yx},e_{yx}]$ then $label_{xy}$ and $label_{yx}$ are also reverse complemented. Note that in the case of an edge built from a reverse-complement overlap, $type_{xy}$ is necessarily the same as $type_{yx}$. To perform a walk in the string graph, if one enters a vertex on an edge of type B then an edge of type E must be used to exit and vice versa. Figure 1 depicts a simple string graph built from three overlapping reads.

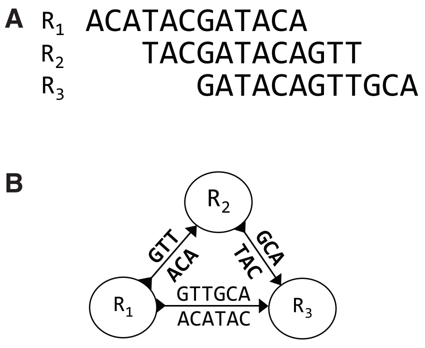

Fig. 1. Diagram of a simple string graph. Three overlapping reads (R_1, R_2, R_3) are shown in (**A**). (**B**) shows the string graph constructed from the overlaps between the reads. The arrowheads pointing into the nodes depict an edge of type B and arrowheads pointing away from the nodes depict edges of type E. The edge $R_1 \leftrightarrow R_3$ is transitive.

The initial graph built from the overlaps between reads is not a string graph yet. Consider a read X that overlaps reads Y and Z, which mutually overlap. The initial string graph will contain the edges $X \leftrightarrow Y$, $X \leftrightarrow Z$ and $Y \leftrightarrow Z$. If Y and Z overlap the same end of X, i.e $type_{xy}=type_{xz}$, then Y and Z must share a common substring of X which is a prefix or suffix of one of Y or Z. This implies that there is a valid path that visits each of the three reads in succession. Let $X \rightarrow Y \rightarrow Z$ be such a path. The string corresponding to this path is a valid assembly of the three reads which is identical to the string corresponding to the path $X \rightarrow Z$. In this case, we say that the edge $X \leftrightarrow Z$ is *transitive*. We will refer to non-transitive edges as *irreducible*. The transitive edges can be removed from the graph without losing any information—the transitive edges (and their corresponding overlaps) could be inferred from the irreducible edges. We can determine useful properties of transitive and irreducible edges. As the graph does not have contained reads, the length of the overlap between $X \leftrightarrow Y$ is necessarily larger than the overlap between $X \leftrightarrow Z$. Equivalently, the length of $label_{xy}$ is shorter than $label_{xz}$, and $label_{xz}$ can be seen as the concatenation of $label_{xy}$ and $label_{yz}$. In other words, $label_{xy}$ of the irreducible edge is a prefix of $label_{xz}$ of the transitive edge.

2.4 The suffix array, BWT and FM-index

The *suffix array* data structure was introduced by Manber and Myers (1990) as a succinct representation of the lexographic ordering of the suffixes of a string. The suffix array of a string X, denoted $\mathbf{SA}_X$, is a permutation of the integers $\{1,2,...,|X|\}$ such that $\mathbf{SA}_X[i]=j$ iff $X[j,|X|]$ is the i-th lexographically lowest suffix of X. For example, if $X=$AAGTA\$ then $\mathbf{SA}_X=[6,5,1,2,3,4]$. Since the suffix array is a sorted data structure, the start positions of all the instances of a pattern Q in X will occur in an interval in $\mathbf{SA}_X$. We refer to such an interval as a *suffix array interval* and associate with it a pair of integers $[l,u]$ denoting the first and last index in $\mathbf{SA}_X$ that correspond to a position in X of an instance of Q. Using $\mathbf{SA}_X$ and the original string X, l and u can be efficiently found with a binary search for Q. Ferragina and Manzini developed a related method of indexing text, called the FM-index, which requires considerably less memory than a suffix array and can compute l and u in $O(|Q|)$ time, independent of the size of the text being searched. Central to the FM-index is the BWT. Originally developed for text compression (Burrows and Wheeler, 1994) the BWT of X, denoted $\mathbf{B}_\mathrm{X}$, is a permutation of the symbols of X such that

$$\mathbf{B}_\mathrm{X}[i]=\begin{cases} X[\mathbf{SA}_\mathrm{X}[i]-1] & \text{if } \mathbf{SA}_\mathrm{X}[i]>1 \\ \$ & \text{if } \mathbf{SA}_\mathrm{X}[i]=1 \end{cases}$$

Restated, $\mathbf{B}_\mathrm{X}[i]$ is the symbol preceding the first symbol of the suffix starting at position $\mathbf{SA}_\mathrm{X}[i]$. Ferragina and Manzini (2000) extended the BWT representation of a string by adding two additional data structures to create a structure known as the FM-index. Let $\mathbf{C}_\mathrm{X}(a)$ be the number of symbols in X that are lexographically lower than the symbol a and $\mathbf{Occ}_\mathrm{X}(a,i)$ be the number of occurrences of the symbol a in $\mathbf{B}_\mathrm{X}[1,i]$. We note that $\mathbf{C}_\mathrm{X}$ and $\mathbf{Occ}_\mathrm{X}$ include counts for the sentinel symbol, \$. Using these two arrays, Ferragina and Manzini provided an algorithm to search for a string Q in X (Ferragina and Manzini, 2000). Let S be a string whose suffix array interval is known to be $[l,u]$. The interval for the string aS can be calculated from $[l,u]$ using $\mathbf{C}_\mathrm{X}$ and $\mathbf{Occ}_\mathrm{X}$ by the following:

$$l=\mathbf{C}_\mathrm{X}(a)+\mathbf{Occ}_\mathrm{X}(a,l-1) \tag{1}$$

$$u=\mathbf{C}_\mathrm{X}(a)+\mathbf{Occ}_\mathrm{X}(a,u)-1 \tag{2}$$

We encapsulate Equations (1) and (2) in the following algorithm, updateBackward.

Algorithm 1 updateBackward($[l,u]$, a)

```
$l \leftarrow \mathbf{C}_\mathrm{X}(a)+\mathbf{Occ}_\mathrm{X}(a,l-1)$
$u \leftarrow \mathbf{C}_\mathrm{X}(a)+\mathbf{Occ}_\mathrm{X}(a,u)-1$
**return** $[l,u]$
```

To search for a string Q, we need to first calculate the interval for the last symbol in Q then use Equations (1) and (2) to iteratively calculate the interval for the remainder of Q. The interval for a single symbol is simply calculated from $\mathbf{C}_\mathrm{X}$. The backwardsSearch algorithm presents the searching procedure in detail. If backwardsSearch returns an interval where $l>u$, Q is not contained in X otherwise $\mathbf{SA}_\mathrm{X}[i]$ is the position in X of each occurrence of Q for $l \le i \le u$.

Algorithm 2 backwardsSearch(Q) - find the interval in $\mathbf{SA}_\mathrm{X}$ for the pattern Q

```
$i \leftarrow |Q|$
$l \leftarrow \mathbf{C}_\mathrm{X}(Q[i])$
$u \leftarrow \mathbf{C}_\mathrm{X}(Q[i]+1)-1$
$i \leftarrow i-1$
**while** $l \le u$ & $i \ge 1$ **do**
    $[l,u] \leftarrow$ updateBackward($[l,u], Q[i]$)
    $i \leftarrow i-1$
**end while**
**return** $[l,u]$
```

The backwardsSearch algorithm requires updating the suffix array interval $|Q|$ times. As each update is a constant-time operation, the complexity of backwardsSearch is $O(|Q|)$. To save memory $\mathbf{Occ}_\mathrm{X}(a,i)$ is stored only for i divisible by d (typically d is around 128). The remaining values of $\mathbf{Occ}_\mathrm{X}$ can be calculated as needed using the sampled values and $\mathbf{B}_\mathrm{X}$.

2.5 The generalized suffix array

We can easily expand the definition of a suffix array to include multiple strings. Let $\mathcal{T}$ be an indexed set of strings and $\mathcal{T}_i$ be element $\mathcal{T}[i]$. We define $\mathbf{SA}_\mathcal{T}[i]=(j,k)$ iff $\mathcal{T}_j[k,|\mathcal{T}_j|]$ is the i-th lowest suffix in $\mathcal{T}$. In the generalized suffix array, unlike the suffix array of a single string, two suffixes can be lexographically equal. We break ties in this case by comparing the indices of the strings. In other words, we treat each string in $\mathcal{T}$ as if it was terminated by a unique sentinel character $\$_i$ where $\$_i<\$_j$ when $i<j$. We extend the definition of the BWT to collections of strings as follows. Let $\mathbf{SA}_\mathcal{T}[i]=(j,k)$ then

$$\mathbf{B}_\mathcal{T}[i]=\begin{cases} \mathcal{T}_j[k-1] & \text{if } k>1 \\ \$ & \text{if } k=1 \end{cases}$$

Like the BWT of a single string, $\mathbf{B}_\mathcal{T}$ is a permutation of the symbols in $\mathcal{T}$; therefore, the definitions of the auxiliary data structures for the FM-index, $\mathbf{C}_\mathcal{T}(a)$ and $\mathbf{Occ}_\mathcal{T}(a,i)$, do not change.

3 METHODS

The construction of the string graph occurs in two stages. First, the complete set of overlaps of length at least τ is computed for all elements of $\mathcal{R}$. The initial overlap graph is then built as described in Section 2.3 and transformed into the string graph using the linear expected time transitive reduction algorithm of Myers (2005). The first step in this process is the computational bottleneck. The all-pairs maximal overlap problem can be optimally solved in $O(N+k^2)$ time using a generalized suffix tree where $N=\sum_{i=1}^{|\mathcal{R}|}|\mathcal{R}_i|$ and $k=|\mathcal{R}|$ (Gusfield, 1997). It is straightforward to restrict this algorithm to only find overlaps of length at least τ at a lower computational cost; however, the amount of memory required for a suffix tree makes this algorithm impractical for large datasets. Myers' proposed the use of a q-gram filter to find the complete set of overlaps. This requires $O(N^2/D)$ time where D is a time-space tradeoff factor dependent on the amount of memory available. We will show that by using the FM-index of $\mathcal{R}$ the set of overlaps can be computed in $O(N+C)$ time for error-free reads where C is the total number of overlaps found. We then provide an algorithm that detects only the overlaps for irreducible edges—removing the need for the transitive reduction algorithm and allowing the direct construction of the string graph.

3.1 Building an FM-index from a set of sequence reads

To build the FM-index of $\mathcal{R}$, we must first compute the generalized suffix array of $\mathcal{R}$. We could do this by creating a string that is the concatenation of all members of $\mathcal{R}$, $S=\mathcal{R}_1\mathcal{R}_2...\mathcal{R}_m$ and then use one of the well-known efficient suffix array construction algorithms to compute $\mathbf{SA}_\mathrm{S}$ (Puglisi *et al.*, 2007). We have adopted a different strategy and have modified the induced-copying suffix array construction algorithm (Nong *et al.*, 2009) to handle an indexed set of strings $\mathcal{R}$ where each suffix array entry is a pair (j,k) as described in Section 2.5. This suffix array construction algorithm is similar to the Ko–Aluru algorithm (Ko and Aluru, 2005). A set of substrings of the text (termed LMS substrings for leftmost S-type, see Nong *et al.*, 2009) is sorted from which the ordering of all the suffixes in the text is induced. Our algorithm differs from the Nong–Zhang–Chan algorithm as we directly sort the LMS substrings using multikey quicksort (Bentley and Sedgewick, 1997) instead of sorting them recursively. This method of construction is very fast in practice as typically only $30-40\%$ of the substrings must be directly sorted. Once $\mathbf{SA}_\mathcal{R}$ has been constructed, the BWT of $\mathcal{R}$, and hence the FM-index is easily computed as described above. We also compute the FM-index for the set of *reversed* reads, denoted $\mathcal{R}'$, which is necessary to compute overlaps between reverse complemented reads. We also output the *lexographic index* of $\mathcal{R}$, which is a permutation of the indices $\{1,2,...,|\mathcal{R}|\}$ of $\mathcal{R}$ sorted by the lexographic order of the strings. This can be found directly from $\mathbf{SA}_\mathcal{R}$ and is used to determine the identities of the reads in $\mathcal{R}$ from the suffix array interval positions once an overlap has been found.

3.2 Overlap detection using the FM-index

We now consider the problem of constructing the set of overlaps between reads in $\mathcal{R}$. Consider two reads X and Y. If a suffix of X matches a prefix of Y an edge of type (E,B) will be created in the initial overlap graph. We will describe a procedure to detect overlaps of this type from the FM-index of $\mathcal{R}$. Let X be an arbitrary read in $\mathcal{R}$. If we perform the backwardsSearch procedure on the string X, after k steps we have calculated the interval $[l,u]$ for the suffix of length k of X. The reads indicated by the suffix array entries in $[l,u]$, therefore, have a substring that matches a suffix of X. Our task is to determine which of these substrings are prefixes of the reads. Recall that if a given element in the suffix array, $\mathbf{SA}_{\mathcal{R}}[i]$, is a prefix then $\mathbf{B}_{\mathcal{R}}[i]=\$$ by definition. Therefore, if we know the suffix array interval for a string P, the interval for the strings beginning with P can be determined by calculating the interval for the string $\$P$ using Equations (1) and (2). This interval, denoted $[l_{\$},u_{\$}]$, indicates that the reads with prefix P are the $l_{\$}$-th to $u_{\$}$-th lexographically lowest strings in $\mathcal{R}$. We can, therefore, recover the indices in $\mathcal{R}$ of the reads overlapping X using lexographic index of $\mathcal{R}$. The algorithm is presented below in findOverlaps.

Algorithm 3 findOverlaps(X, τ) - determine the reads in $\mathcal{R}$ that overlap X by at least τ symbols

```
$i \leftarrow |X|$
$l \leftarrow \mathbf{C}_{\mathcal{R}}(X[i])$
$u \leftarrow \mathbf{C}_{\mathcal{R}}(X[i]+1)-1$
$i \leftarrow i-1$
while $l \leq u$ & $i \geq 1$ do
    if $|X|-i+1 \geq \tau$ then
        $[l_{\$},u_{\$}] \leftarrow$ updateBackwards($[l,u]$, \$)
        if $l_{\$} \leq u_{\$}$ then
            outputOverlaps($X$, $[l_{\$},u_{\$}]$)
        end if
    end if
    $[l,u] \leftarrow$ updateBackward($[l,u]$, $X[i]$)
    $i \leftarrow i-1$
end while
if $l \leq u$ then
    outputContained($X$, $[l,u]$)
end if
```

The findOverlaps algorithm is similar to the backwards search procedure presented in Section 2.4. It begins by initializing $[l,u]$ to the interval containing all suffixes that begin with the last symbol of X. The interval $[l,u]$ is then iteratively updated for longer suffixes of X. When the length of the suffix is at least the minimum overlap size, τ, we determine the interval for the reads that have a prefix matching the suffix of X and output an overlap record for each entry (using the subroutine outputOverlaps). When the update loop terminates, $[l,u]$ holds the interval corresponding to the full length of X. The outputContained procedure writes a containment record for X if X is contained by any read in $[l,u]$ based on the rules described in Section 2.3. The overlaps detected by findOverlaps correspond to edges of type (E,B). We must also calculate the overlaps for edges of type (E,E) and (B,B), which arise from overlapping reads originating from opposite strands. To calculate edges of type (E,E), we use findOverlaps on the complement of X (not reversed) and the FM-index of $\mathcal{R}'$. Similarly, to calculate edges of type (B,B), we use findOverlaps on $\overline{X}$ (the reverse complement of X) and the FM-index of $\mathcal{R}$.

The overlap records created by outputOverlaps are constructed in constant time as they only require a lookup in the lexographic index of $\mathcal{R}$. Let c_i be the number of overlaps for read $\mathcal{R}_i$. The findOverlaps algorithm makes at most $|\mathcal{R}_i|$ calls to updateBackwards and a total of c_i iterations in outputOverlaps for a total complexity of $O(|\mathcal{R}_i|+c_i)$. For the entire set $\mathcal{R}$, the complexity is $O(N+C)$ where $C=\sum_{i=1}^{|\mathcal{R}|} c_i$. Note that the majority of these edges are transitive and subsequently removed. We can, therefore, improve this algorithm by only outputting the set of irreducible edges, allowing the direct construction of the string graph. We address this in Section 3.3.

In rare cases, multiple valid overlaps may occur between a pair of reads. In this case, the intervals detected during findOverlaps will contain intersecting or duplicated intervals. To handle this, we can modify findOverlaps to first collect the entire set of found intervals. This interval set could then be sorted and duplicated or intersecting intervals that represent sub-maximal overlaps can be removed. The outputOverlaps procedure can be called on the entire reduced interval set to output the set of maximal overlaps.

3.3 Detecting irreducible overlaps

To directly construct the string graph, we must only output irreducible edges. Recall from Section 2.3 that the labels of the irreducible edges for a given read are prefixes of the labels of transitive edges. We use this fact to differentiate between irreducible and transitive edges during the overlap computation. Consider a read X and the set of reads that overlap a suffix of X, $\mathcal{O}$. We could devise an algorithm to find the subset consisting only of irreducible edges by calculating the edge-labels of all members of $\mathcal{O}$ and filtering out the members whose label is the extension of the label of some other read. This would require iterating over all members of $\mathcal{O}$, which can be quite large for repetitive reads. We will now show that the labels of the irreducible edges can be constructed directly from the suffix array intervals using the FM-index.

Consider a substring S that occurs in $\mathcal{R}$ and its suffix array interval $[l,u]$. Let a *left extension* of S be a string of length $|S|+1$ of the form aS. We can use $\mathbf{B}_{\mathcal{R}}[l,u]$ to determine the set of left extensions of S. Let $\mathcal{B}$ be the set of symbols that appear in the substring $\mathbf{B}_{\mathcal{R}}[l,u]$. The left extensions of S are the strings aS such that $a \in \mathcal{B}$. Note that we do not have to iterate over the range $\mathbf{B}_{\mathcal{R}}[l,u]$ to determine $\mathcal{B}$. Since $\mathbf{Occ}_{\mathcal{R}}(a,i)$ is defined to be the number of times symbol a occurs in $\mathbf{B}_{\mathcal{R}}[1,i]$ we can count the number of occurances of a in $\mathbf{B}_{\mathcal{R}}[l,u]$ (and hence aS in $\mathcal{R}$) in constant time by taking the difference $\mathbf{Occ}_{\mathcal{R}}(a,u)-\mathbf{Occ}_{\mathcal{R}}(a,l-1)$. If the \$ symbol occurs in $\mathbf{B}_{\mathcal{R}}[l,u]$ we say that S is *left terminal*, in other words one of the elements of $\mathcal{R}$ has S as a prefix. We similarly define a *right extension* of S as a string of length $|S|+1$ of the form Sa. While we cannot build the right extensions of S directly from the FM-index, the right extensions of S are equivalent to left extensions of S' (the reverse of S) in $\mathcal{R}'$. Let S be *right terminal* if \$ exists in $\mathbf{B}_{\mathcal{R}'}[l',u']$, in other words S is a suffix of some string in $\mathcal{R}$.

The procedure to find all the irreducible edges of a read X and construct their labels is to find all the intervals containing the prefixes of reads that overlap a suffix of X, then iteratively extend them rightwards until a right-terminal extension is found. The terminated read forms an irreducible edge with X and the label of the edge is the sequence of bases that were used during the right-extension. All non-terminated strings with the same sequence of extensions are transitive and, therefore, not considered further.

The algorithm requires searching the FM-index in two directions, first backwards to determine the intervals of overlapping prefixes and then forwards to extend those prefixes and build the irreducible labels. Naively this would require first determining the intervals $[l,u]$ for each matching prefix, P, and then reversing the prefix and performing a backwards search on the FM-index of $\mathcal{R}'$ to find the interval $[l',u']$ for P'. The intervals $[l',u']$ would then be used in the extension stage to determine the labels of the irreducible edges. We can do better, however, by noting that the interval $[l',u']$ can be calculated directly during the backwards search without using the FM-index of $\mathcal{R}'$. We define $\mathbf{OccLT}_{\mathcal{R}}(a,i)$ to be the number of symbols that are lexographically lower than a in $\mathbf{B}_{\mathcal{R}}[1,i]$. Let $S=X[i,|X|]$ be a suffix of X and $[l_i,u_i]$ its suffix array interval. Suppose we know the interval $[l'_i,u'_i]$ for S' in $\mathcal{R}'$. Let $a=X[i-1]$. The interval for $S'a=[l'_{i-1},u'_{i-1}]$ is therefore

$$l'_{i-1}=l'_i+(\mathbf{OccLT}_{\mathcal{R}}(a,u_i)-\mathbf{OccLT}_{\mathcal{R}}(a,l_i-1)) \tag{3}$$

$$u'_{i-1}=l'_{i-1}+(\mathbf{Occ}_{\mathcal{R}}(a,u_i)-\mathbf{Occ}_{\mathcal{R}}(a,l_i-1)-1) \tag{4}$$

The interval for $X'[1]$ is identical to that of $X[|X|]$, since $\mathbf{B}_{\mathcal{R}}$ and $\mathbf{B}_{\mathcal{R}'}$ are both permutations of symbols in $\mathcal{R}$, therefore, $\mathbf{C}_{\mathcal{R}}=\mathbf{C}_{\mathcal{R}'}$. We can, therefore, initialize the interval $[l',u']$ to the same initial value of $[l,u]$ and perform a forward search of X' simulatenously while performing a backward search of X using only the FM-index of $\mathcal{R}$. This does not require any additional storage as the $\mathbf{OccLT}_{\mathcal{R}}$ array can easily be computed from $\mathbf{Occ}_{\mathcal{R}}$ by summing the values for symbols less than a. This procedure is similar to the 2way-BWT search recently proposed by Lam *et al.* (2009). The updateFwdBwd algorithm implements Equations (3) and (4) along with updateBackward to calculate the pair of intervals. The $\mathcal{F}$ parameter to updateFwdBwd indicates the FM-index used — that of $\mathcal{R}$ or $\mathcal{R}'$.

Algorithm 4 updateFwdBwd($[l,u,l',u']$, a, $\mathcal{F}$)

```
$l' \leftarrow l' + (\mathbf{OccLT}_{\mathcal{F}}(a,u) - \mathbf{OccLT}_{\mathcal{F}}(a,l-1))$
$u' \leftarrow l' + (\mathbf{Occ}_{\mathcal{F}}(a,u) - \mathbf{Occ}_{\mathcal{F}}(a,l-1) - 1)$
$[l,u] \leftarrow$ updateBackwards$(l,u,a,\mathcal{F})$
return $[l,u,l',u']$
```

We now give the full algorithm for detecting the irreducible overlaps for a read X. The algorithm is performed in two stages, first a backwards search on X is performed to collect the set of interval pairs, denoted $\mathcal{I}$, for prefixes that match a suffix of X. This algorithm is presented in findIntervals below and is conceptually similar to findOverlaps.

Algorithm 5 findIntervals(X, τ)

```
$\mathcal{I} \leftarrow \emptyset$
$i \leftarrow |X|$
$l \leftarrow C(X[i])$
$u \leftarrow C(X[i]+1)-1$
$[l',u'] \leftarrow [l,u]$
$i \leftarrow i-1$
while $l \leq u$ & $i \geq 1$ do
  if $|X|-i+1 \geq \tau$ then
    $[l_{\$},u_{\$},l'_{\$},u'_{\$}] \leftarrow$ updateFwdBwd$([l,u,l',u'],\$,\mathcal{R})$
    if $l_{\$} \leq u_{\$}$ then
      $\mathcal{I} \leftarrow \mathcal{I} \cup [l_{\$},u_{\$},l'_{\$},u'_{\$}]$
    end if
  end if
  $[l,u,l',u'] \leftarrow$ updateFwdBwd$([l,u,l',u'],X[i],\mathcal{R})$
  $i \leftarrow i-1$
end while
return $\mathcal{I}$
```

The interval set found by findIntervals is processed by extractIrreducible to find the intervals corresponding to the irreducible edges of X. This algorithm has two parts. First, the set of intervals is tested to see if some read in the interval set is right terminal. If so, the intervals corresponding to the right terminal reads form irreducible edges with X and are returned. If no interval has terminated, we create a subset of intervals for each right extension of $\mathcal{I}$ and recursively call extractIrreducible on each subset.

The algorithm above assumes that there are no reads that are strict substrings of other reads (in other words, all the containments are between identical reads). If this is not the case, a slight modification must be made. If the set of reads overlapping X includes a read that is a proper substring of some other read it is possible that the first right terminal extension found is not that of an irreducible edge but of the contained read. It is straightforward to handle this case by observing that such a read will have an overlap which is strictly shorter than that of the irreducible edge. In other words, the only acceptable right terminal extension is to the reads in $\mathcal{I}$ that have the longest overlap with X.

We can similarly modify extractIrreducible to handle overlaps for reads from opposite strands. To do this, we use findIntervals to determine the intervals for overlaps for the same strand as X and overlaps from the opposite strand of X (using the complement of X as in the previous section). When extending an interval that was found by the complement of X, we extend it by the complement of a. In other words, if we are extending same-strand intervals by A, we extend opposite strand intervals by T and so on.

Algorithm 6 extractIrreducible($\mathcal{I}$)

```
if $\mathcal{I} = \emptyset$ then
  return $\emptyset$
end if
$\mathcal{L} \leftarrow \emptyset$
for all $[l,u,l',u'] \in \mathcal{I}$ do
  $[l'_{\$},u'_{\$},l_{\$},u_{\$}] \leftarrow$ updateFwdBwd$([l',u',l,u],\$,\mathcal{R}')$
  if $l_{\$} \leq u_{\$}$ then
    $\mathcal{L} \leftarrow \mathcal{L} \cup [l_{\$},u_{\$}]$
  end if
end for
if $\mathcal{L} \neq \emptyset$ then
  return $\mathcal{L}$
end if
for all $a \in \Sigma$ do
  $\mathcal{I}_a \leftarrow \emptyset$
  for all $[l,u,l',u'] \in \mathcal{I}$ do
    $[l'_a,u'_a,l_a,u_a] \leftarrow$ updateFwdBwd$([l',u',l,u],a,\mathcal{R}')$
    if $l_a \leq u_a$ then
      $\mathcal{I}_a \leftarrow \mathcal{I}_a \cup [l_a,u_a,l'_a,u'_a]$
    end if
  end for
  $\mathcal{L} \leftarrow \mathcal{L} \cup$ extractIrreducible$(\mathcal{I}_a)$
end for
return $\mathcal{L}$
```

We now offer a sketch of the complexity of the irreducible overlap algorithm. Let L_i be the label of irreducible edge i. During the construction of L_i at most k_i intervals must be updated, corresponding to the number of reads that have an edge-label containing L_i. The sum over all irreducible edges, $E = \sum_i (|L_i| k_i)$, is the total number of interval updates performed by extractIrreducible. Note that each read in $\mathcal{R}$ is represented by a path through the string graph. The total number of times edge i is used in the set of paths spelling all the reads in $\mathcal{R}$ is k_i and the amount of sequence in $\mathcal{R}$ contributed by edge i is $|L_i| k_i$. This implies E can be no larger than N, the total amount of sequence in $\mathcal{R}$, and extractIrreducible is $O(N)$. As findIntervals is also $O(N)$, the entire irreducible overlap detection algorithm is $O(N)$.

4 RESULTS

As a proof of concept, we implemented the above algorithms. The program is broken into three stages: index, overlap and assemble. The index stage constructs the suffix array and FM-index for a set of sequence reads, the overlap stage computes the set of overlaps between the reads and the assemble stage builds the string graph, performs transitive reduction if necessary, then compacts unambiguous paths in the graph and writes out a set of contigs. We tested the performance of the algorithms with two sets of simulations. In both sets of simulations, we compared the exhaustive overlap algorithm (which constructs the set of all overlaps) and the direct construction algorithm (which only outputs overlaps for irreducible edges). First, we simulated *Escherichia coli* read data with mean sequence depth from 5× to 100× to investigate the computational complexity of the overlap algorithms as a function of depth. After constructing the index for each dataset, we ran the overlap step in exhaustive and direct mode with $\tau = 27$. The running times of these simulations are shown in Figure 2. As expected,

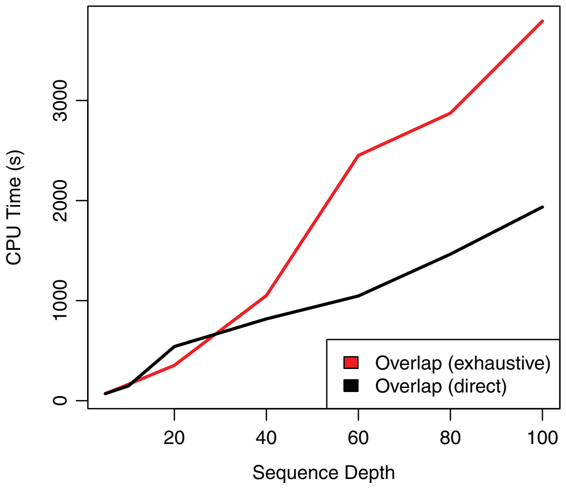

Fig. 2. The running time of the direct and exhaustive overlap algorithms for simulated *E. coli* data with sequence depth from 5× to 100×. The direct overlap algorithm scales linearly with sequence depth. As the number of overlaps grows quadratically with sequence depth, the exhaustive overlap algorithm exhibits above-linear scaling.

Table 1. Simulation results for human chromosomes 22, 15, 7 and 2

	chr 22	chr 15	chr 7	chr 2	ratio
Chr. size (Mb)	34.9	81.7	155.4	238.2	6.8
Number of reads (M)	7.0	16.3	31.1	47.6	6.8
Contained reads (k)	684	1668	3103	4709	6.9
Contained (%)	9.8	10.2	10.0	9.9	–
Transitive edges (M)	38.0	93.0	177.7	274.6	7.2
Irreducible edges (M)	6.3	14.9	28.7	44.4	7.0
Assembly N50 (kbp)	4.0	4.6	4.2	4.7	–
Longest contig (kbp)	31.9	47.7	53.1	48.6	–
Index time (s)	2606	9743	19 779	30 866	11.8
Overlap -e time (s)	2657	6572	12 970	18 060	6.8
Overlap -d time (s)	2885	6750	13 271	19 437	6.7
Assemble -e time (s)	1836	4043	8112	13 095	7.1
Assemble -d time (s)	423	1161	2044	3226	7.6
Index memory (GB)	8.0	18.6	35.4	54.5	6.8
Overlap -e mem. (GB)	2.4	5.5	10.5	16.1	6.7
Overlap -d mem. (GB)	2.4	5.5	10.4	16.1	6.7
Assemble -e mem. (GB)	5.9	14.2	27.2	41.9	7.1
Assemble -d mem. (GB)	2.7	6.3	12.1	18.6	6.9

For the overlap and assemble rows, -e and -d indicate the exhaustive and direct algorithms, respectively. The last column is the ratio between chromosome 2 and 22.

the direct overlap algorithm scales linearly with sequence depth. The exhaustive overlap algorithm exhibits the expected above-linear scaling as the number of overlaps for a given read grows quadratically with sequence depth.

To assess the quality of the resulting assembly, we assembled the data using the direct overlap algorithm and compared the contigs to the reference. For each level of coverage, we selected τ to maximize the assembly N50 value. The N50 values ranged from 1.7 kbp (5× data, $\tau = 17$) to 80 kbp (100× data, $\tau = 85$). We aligned the contigs to the reference genome with bwa-sw (Li and Durbin, 2010) and found that no contigs were misassembled.

We also simulated data from human chromosomes 22, 15, 7 and 2 to assess how the algorithms scale with the size of the genome. We pre-processed the chromosome sequences to remove sequence gaps then generated 100 bp error-free reads randomly at an average coverage of 20× for each chromosome. The results of the simulations are summarized in Table 1. The running time of the exhaustive and direct overlap algorithms are comparable. As the sequence depth is fixed at 20×, both overlap algorithms scale linearly with the size of the input data. The final stage of the algorithm, building the string graph and constructing contigs, is much shorter for the direct algorithm as the transitive reduction step does not need to be performed. In addition, this step requires considerably less memory as the initial graph constructed by the direct algorithm only contains irreducible edges.

The bottleneck in terms of both computation time and memory usage is the indexing step, which builds the suffix array and FM-index for the entire read set. This required 8.5 h and ∼55 GB of memory for chromosome 2. Extrapolating to the size of the human genome indicates it would require ∼4.5 days and 700 GB of memory to index 20× sequence data. While the computational time is tractable, the amount of memory required is not practical for the routine assembly of human genomes. We address ways to reduce the computational requirements in Section 5.

5 DISCUSSION

We have described an efficient method of constructing a string graph from a set of sequence reads using the FM-index. This work is the first step in the construction of a new, general purpose sequence assembler that will be effective for both short reads of the current generation of sequence technology and the longer reads of the sequencing instruments on the horizon. Unlike the de Bruijn graph formulation, the string graph is particularly well-suited for the assembly of mixed length read data. While the primary algorithms are in place, a considerable amount of work remains. Most pressing is the issue of adapting the assembler to handle real sequence data that contains base-calling errors. This amounts to adapting the algorithms to handle inexact overlaps. The BWA, Bowtie and SOAP2 aligners implement a number of strategies and heuristics for dealing with base mismatches and small insertion/deletions (Langmead *et al.*, 2009; Li and Durbin, 2009; Li *et al.*, 2009). These strategies directly translate to finding overlaps. Let ϵ be the maximum allowed difference between two overlapping reads. When performing the backwards search to find overlaps, we can allow the search to proceed to mismatched bases or gaps while ensuring that the ϵ bound is respected. We can similarly modify the irreducible overlap detection algorithm by allowing the right-extension phase to extend to mismatch bases or gaps. Here, we would only consider an interval to be transitive with respect to one of the irreducible intervals if the inferred difference between the intervals is less than ϵ.

Our intention is to build an assembler that can handle genomes up to several gigabases in size, such as for human or other vertebrate genomes and our initial results indicate that our algorithms scale well. The introduction of sequencing errors will increase the complexity of the irreducible overlap identification step but this step is straightforward to parallelize if necessary because it is carried out for each non-contained read. The construction of the suffix array is currently the computational bottleneck; however, there are a number of established ways to improve this. We can lower the amount of memory required by exploiting the redundancy present in a set of sequencing reads by using a compressed index. The compressed suffix array is one such index and a method was recently developed to merge two compressed suffix arrays that possibly allows a distributed construction algorithm (Sirén, 2009). Additionally, efficient external memory (disk-based) BWT construction algorithms have been developed that allow the construction of the FM-index for very large datasets while using a limited amount of main memory (Dementiev *et al.*, 2008; Ferragina *et al.*, 2010).

It is worth investigating the equivalency of the de Bruijn graph and string graph formulations (Pop, 2009). This has been studied in terms of the computational complexity of reconstructing the complete sequence of the genome and both formulations have been shown to be NP-hard (Medvedev *et al.*, 2007). We would like to know the equivalence in terms of the information contained in the graph. Consider the case where all sequence reads are of length l and every l-mer in the genome has been sampled once. In this case, the de Bruijn graph and string graph constructions (using parameters $k=l-1$ and $\tau=l-1$ respectively) are equivalent. In the realistic case where the genome is unevenly sampled, the relationship is not clear. In the original paper on the EULER assembler Pevzner presents an algorithm to recover the information lost during the deconstruction of reads into k-mers by finding consistent read-paths through the k-mer graph (Pevzner *et al.*, 2001). It is conceivable that if this procedure is able to perfectly reconstruct the information lost the resulting graph would be equivalent to Myers' string graph. This is not clear, however, and the equivalency of these formulations is a question we would like to address.

ACKNOWLEDGEMENTS

We thank Veli Mäkinen and members of the Durbin group for discussions related to string matching and sequence assembly.

Funding: This work was supported by the Wellcome Trust (grant number WT077192). J.T.S. is supported by a Wellcome Trust Sanger Institute Research Studentship.

Conflict of interest: none declared.

REFERENCES

Bentley,J.L. and Sedgewick,R. (1997) Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 360–369.

Burrows,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. In *Technical report 124*, Digital Equipment Corporation, Palo Alto, CA.

Chaisson,M.J. and Pevzner,P.A. (2008) Short read fragment assembly of bacterial genomes. *Genome Res.*, **18**, 324–330.

Dementiev,R. *et al.* (2008) Better external memory suffix array construction. *J. Exp. Algorithmics*, **12**, 1–24.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 390–398.

Ferragina,P. *et al.* (2010) Lightweight data indexing and compression in external memory. In *Proceedings of the Latin American Theoretical Informatics Symposium. Lecture Notes of Computer Science*, Springer, Heidelberg, Germany.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK.

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.

Ko,P. and Aluru,S. (2005) Space efficient linear time construction of suffix arrays. *J. Discrete Algorithm.*, **3**, 143–156.

Lam,T. W. *et al.* (2009) High throughput short read alignment via bi-directional bwt. In *2009 IEEE International Conference on Bioinformatics and Biomedicine*, Vol. 0. IEEE, Los Alamitos, CA, pp. 31–36.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25+.

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

Li,H. and Durbin,R. (2010) Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, **26**, 589–595.

Li,R. *et al.* (2009) Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**, 1966–1967.

Manber,U. and Myers,G. (1990) Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 319–327.

Medvedev,P. *et al.* (2007) Computability of models for sequence assembly. In *Algorithms in Bioinformatics*, Lecture Notes in Computer Science, chapter 27, Heidelberg, Germany, pp. 289–301.

Myers,E.W. (2005) The fragment assembly string graph. *Bioinformatics*, **21** (suppl_2), ii79–85.

Nong,G. *et al.* (2009) Linear suffix array construction by almost pure induced-sorting. In *DCC '09 Proceedings of the IEEE Conference on Data Compression*, Los Alamitos, CA, USA, pp. 193–202.

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

Pop,M. (2009) Genome assembly reborn: recent computational challenges. *Brief Bioinform*, **10**, 354–366.

Puglisi,S.J. *et al.* (2007) A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, **39**, 4+.

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

Sirén,J. (2009) Compressed suffix arrays for massive data. In *String Processing and Information Retrieval*, pp. 63–74.

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