# Efficient Construction of Assembly String Graphs using FM-index

**Sudhanva Shyam Kamath**

06-18-01-10-12-21-1-19267
DS 202
Department of Computational and Data Sciences

# 1 Introduction

The sequence assembly problem is usually approached by building de Bruijn graphs. However, eukaryotic genomes typically have highly repetitive segments that can potentially offer spurious relationships between reads. Overlap based methods typically have larger computational costs. One such method is the concept of a *string graph* introduced by Myers in 2005 [1].

Pairwise overlaps between all pairs of reads must first be calculated and 'transitive' edges must be removed to generate a string graph. Relative to the de Bruijn graph, however, constructing a string graph is much more expensive, because the set of all pairwise, inexact overlaps between reads must be found. This is a bottleneck in the assembly problem - the computation of the all-pairs overlap, which has time complexity $O(N^2)$, where $N$ is the sum of the lengths of the reads. The Ferragina-Manzini index (FM-index) data structure can be used to find overlaps of length at least $\tau$ among a set of reads, with time compleity $O(N)$.

# 2 Problem

## 2.1 Definitions

**Definition 1.** *A* string $X$ *is a sequence of symbols from the alphabet* $\Sigma$*. All strings are terminated by a special* \$ *character, which is lexicographically smaller than all characters in* $\Sigma$*. The* length *of a string is denoted* $|X|$*.* $X[i] = a_i$*, for* $a_i \in \Sigma$*.* $X[|X|] = \$$*.* $X[i, j]$ *denotes the substring* $X[i], X[i + 1], \ldots, X[j]$*, when* $i < j$*. The* suffix *of a read is any substring* $X[k, |X|]$*. The* prefix *of a read is any substring* $X[1, k]$*. The reverse of a read* $X$ *is the string* $X' = X[|X| - 1], X[|X| - 2], \ldots, X[1]$*.*

**Definition 2.** *A* genome *is defined to be a long string from the alphabet* $\{A, C, G, T\}$*. A* read *is a substring of a genome.*

The notation $\overline{X}$ is used to denote the *reverse complement* of a read $X$. For instance, if a sample read is $AACCGGTT$, then its reverse complement is $TTGGCCAA$.

As a result of a shotgun sequencing experiment, we obtain an indexed set of reads $\mathcal{R}$, which are assumed to be randomly sampled from the genome with an unknown sequence. Loosely, the sequence assembly problem aims to reconstruct the genome using $\mathcal{R}$. *Overlaps* occur between a pair of reads $r_i, r_j \in \mathcal{R}$, when a suffix of $r_i$ is equal to the prefix of $r_j$. To reject spurious overlaps, a minimum acceptable overlap threshold $\tau$ can be set based on parameters of the experiment.

## 2.2 String Graphs

A *string graph* can be built from $\mathcal{R}$ to reconstruct the genome. First, any reads which are contained in other reads, i.e. are substrings of other reads, are removed from $\mathcal{R}$. The string graph is constructed using the remaining reads in $\mathcal{R}$.

**Definition 3.** *All reads in* $\mathcal{R}$ *left after the deletion of contained reads form the vertex set of the string graph. The edges in a string graph are bidirectional. Suppose* $X$ *and* $Y$ *are two reads such that* $X[s_{xy}, e_{xy}] = Y[s_{yx}, e_{yx}]$*.* $X[s_{xy}, e_{xy}]$ *is called the 'matched' portion of* $X$ *and the rest is called the 'unmatched' portion of* $X$*. A bidirectional edge* $X \leftrightarrow Y$*, exists between the vertices* $X$ *and* $Y$ *in the string graph.*

Some more details are associated with edges in string graphs and they are as follow. If $s_{xy} = 1$ and $e_{xy} = |X|$, the read $X$ fully overlaps with $Y$ and is said to be *contained* in $Y$. If $X$ and $Y$ are both contained in the other, then $X$ and $Y$ are identical reads. In this case, we make a choice and say that the read with the higher index in $\mathcal{R}$ is contained in the read with the lower index.

$X$ and $Y$ are said to have a *proper overlap*, if neither $X$ nor $Y$ are contained in the other, and either $X[s_{xy}, e_{xy}]$ is a prefix of $X$, with $s_{xy} = 1$, and $Y[s_{yx}, e_{yx}]$ is a suffix of $Y$, with $e_{yx} = |Y|$, or vice versa. Reads from opposite strands of DNA they can still form an overlap. Here, $\overline{X[s_{xy}, e_{xy}]} = Y[s_{yx}, e_{yx}]$, and both of $X[s_{xy}, e_{xy}]$ and $Y[s_{yx}, e_{yx}]$ are prefixes or suffixes.

Reads which aren't contained in other reads form the vertex set of the string graph. A bidirected edge is added between two vertices which have a proper overlap. A $4-$tuple is assigned as a label to this edge: $(type_{xy}, type_{yx}, label_{xy}, label_{yx})$. The *type* of an edge is used to denote the type of overlap between reads $X$ and $Y$.

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

The *label* of an edge is the unmatched portion of the read. The concatenation of $X$ and $label_{xy}$ is the assembly of the reads $X$ and $Y$. It is defined thus:

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

In the event that the reverse complement of a read overlaps with another read, $type_{xy} = type_{yx}$, because the overlap is always a prefix-prefix overlap or a suffix-suffix overlap. Suppose $\overline{X[s_{xy}, e_{xy}]} = Y[s_{yx}, e_{yx}]$, then, $label_{xy}$ is the reverse complement of the unmatched portion of $Y$ and $label_{yx}$ is the reverse complement of the unmatched portion of $X$. While traversing a walk in a string graph, suppose a vertex is entered using an edge of type $B$, then it must be exited using an edge of type $E$ and vice versa.

To complete the construction of the string graph, one more step is required. Suppose there are three reads, $X, Y, Z$ such that the edges $X \leftrightarrow Y, Y \leftrightarrow Z$, and $X \leftrightarrow Z$ represent proper overlaps between the reads. Suppose $type_{xy} = type_{xz}$. Then, both $Y$ and $Z$ overlap at the same end of $X$. Thus, there is a substring of $X$ which is a prefix of $Y$ or $Z$ but a substring of both $Y$ and $Z$. Thus, there is a path which starts at one of the reads and visits each of the other reads. If $X \rightarrow Y \rightarrow Z$ is such a path, then the string corresponding to this path is an assembly of the three reads $X, Y$, and $Z$. This string is the same as the string corresponding to the path $X \rightarrow Z$. The edge $X \leftrightarrow Z$ is called a *transitive edge*. Transitive edges can be safely deleted without losing any information about the assembly of the reads in the string graph. Edges which aren't transitive are called *irreducible edges*. Since there are no contained reads in the string graph, the length of the overlap between $X$ and $Y$ is larger than the length of the overlap between $X$ and $Z$. Hence, $label_{xy}$ is shorter than $label_{xz}$. The label of the irreducible edge has to be shorter than the label of the transitive edge.

# 3 Data Structures

## 3.1 Suffix Array

**Definition 4.** *The* suffix array *of a string $X$, denoted $SA_X$, is a permutation of $\{1, 2, \ldots, |X|\}$ such that $SA_X[i] = j$ if, and only if, $X[j, |X|]$ is the $i$th lexicographically smallest suffix of $X$.*

For a pattern $Q$, all the indices of suffixes starting with $Q$ will occur in an interval in $SA_X$. Suppose $[l, u]$ denote the range of indices in which $Q$ appears at the beginning of the suffixes of $X$ in $SA_X$. Then, $[l, u]$ is called the *suffix array interval* corresponding to $Q$ in $SA_X$. Using the FM-index, this interval can be found in $O(|Q|)$ time.

## 3.2 Burrows-Wheeler Transform

**Definition 5.** *The Burrows-Wheeler Transformation of a string $X$, denoted $B_X$, is a permutation of the symbols of $X$ such that*

$$B_X[i] = \begin{cases} X[SA_X[i] - 1] & \text{if } SA_X[i] > 1 \\ \$ & \text{if } SA_X[i] = 1 \end{cases}$$

## 3.3 FM-Index

The FM-Index is an augmentation of the BWT of a string to enable fast location of the suffix array interval of a pattern $Q$ in the suffix array of the string. Let $C_X(a)$ denote the number of symbols in $X$ that are lexicographically smaller than $a$. Let $Occ_X(a, i)$ denote the number of occurrences of $a$ in $B_X[1, i]$. If $[l, u]$ is known to be the suffix array interval of a pattern $S$, then the suffix array interval of $aS$ can be determined using the following equations:

$$l = C_X(a) + Occ_X(a, l - 1) \tag{1}$$
$$u = C_X(a) + Occ_X(a, u) - 1 \tag{2}$$

### 3.3.1 UpdateBackward($[l, u], a$)

1. $l \leftarrow C_X(a) + Occ_X(a, l - 1)$
2. $u \leftarrow C_X(a) + Occ_X(a, u) - 1$
3. **return** $[l, u]$

Each call to the UpdateBackward procedure takes $O(1)$ time. This is because both $C_X$ and $Occ_X$ can be computed in $O(1)$ time. To reduce space requirements for $Occ_X$, $Occ_X(a, i)$ can be stored only for $i$ divisible by $d$. The remaining values can be calculated using the Burrows Wheeler Transform.

Searching for a pattern $Q$ in the string $X$ involves searching for the interval of the last symbol in $Q$ in $SA_X$ and then iteratively using the UpdateBackward procedure to find the suffix array interval for $Q$. The algorithm is as follows.

#### 3.3.2 BackwardSearch($Q$)

1. $i \leftarrow |Q|$
2. $l \leftarrow C_X(Q[i])$
3. $u \leftarrow C_X(Q[i]+1) - 1$
4. $i \leftarrow i - 1$
5. **while** $l \leq u$ and $i \geq 1$ **do**
   (a) $[l, u] \leftarrow UpdateBackward([l, u], Q[i])$
   (b) $i \leftarrow i - 1$
6. **return** $[l, u]$

Note that if the BackwardSearch procedure returns an interval with $l > u$, then $Q$ doesn't exist in $X$. The BackwardSearch algorithm calls UpdateBackward $|Q|$ times. Thus, the time complexity is $O(|Q|)$.

### 3.4 Generalized Suffix Array

For an indexed set of strings $\mathcal{T}$, a generalized suffix array can be constructed. To do this, we introduce $|\mathcal{T}|$ sentinel string terminator characters, each of which are lexicographically smaller than the symbols in $\Sigma$. The sentinels are also labelled using natural numbers. $\$_i$ is used as a string terminator sentinel character for string $\mathcal{T}_i$. With this in mind, $SA_{\mathcal{T}}[i] = (j, k)$ if, and only if $\mathcal{T}_j[k, |\mathcal{T}_j|]$ is the $i$th lexicographically smallest suffix among all strings in $\mathcal{T}$. If two suffixes are equal, then they are distinguished by their sentinel string terminator characters.

The Burrows Wheeler Transform of an indexed collection of strings $\mathcal{T}$ can be defined as follows. If $SA_{\mathcal{T}}[i] = (j, k)$,

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

Since $B_{\mathcal{T}}$ is a permutation of the symbols in $\mathcal{T}$, the definitions of $C_{\mathcal{T}}$ and $Occ_{\mathcal{T}}$ are unchanged and the procedures UpdateBackward and BackwardSearch are unchanged.

## 4 Construction of String Graph

In the original paper by Myers (2005) [1], the algorithm to delete transitive edges in the string graph had an average linear time in the total length of reads. If $N = \sum_{i=1}^{|\mathcal{R}|} |\mathcal{R}_i|$, the problem of finding all-pairs maximal overlap can be solved using generalized suffix trees in time $O(N + |\mathcal{R}|^2)$. Myers had suggested a different algorithm based on a q-gram filter which took time $O(N^2/D)$, where $D$ depended on the amount of memory available. Using the FM-index of $\mathcal{R}$, the set of overlaps can be computed in $O(N + C)$ time, where $C$ is the total numbers of overlaps.

## 4.1 Building FM-Index of $\mathcal{R}$

The generalized suffix array of $\mathcal{R}$ is computed by concatenating all strings in $\mathcal{R}$ into a single string $S = R_1 R_2 \cdots R_m$. A modified version of the Ko-Aluru algorithm is used to build the generalized suffix array of $S$. After construction of the suffix array of $S$, the Burrows-Wheeler Transform of $S$ is computed and the FM-Index of $S$ is also computed. To compute overlaps between reverse complemented reads, the FM-Index of the set of reversed reads is also computed. The indexed set of reversed reads is denoted $\mathcal{R}'$. The *lexicographic index* of $\mathcal{R}$ is a permutation of the indices of the strings in $\mathcal{R}$ which sorts the reads in $\mathcal{R}$ in lexicographic order. This is used to determine the identity of the reads in $\mathcal{R}$, based on the suffix array interval positions, once an overlap has been established.

## 4.2 Detecting All Overlaps

First, the authors created a method to find all overlaps of length at least $\tau$ between a read $X$ and all other reads of type $(E, B)$. Other overlaps can be computed by modifying the algorithm.

Let $X$ and $Y$ be two reads such that a suffix of $X$ matches a prefix of $Y$. This corresponds to an overlap of type $(E, B)$. Starting with a read $X$, all the reads $Y$ which overlap with a $k$ length suffix of $X$ are determined. This is accomplished by performing a BackwardSearch $k$ times on $X$ using the FM-Index of $\mathcal{R}$. The interval $[l, u]$ output by the BackwardSearch contain reads which have a substring which matches the $k$ length suffix of $X$. The substrings which are prefixes of the reads must be sieved out. To do this, the indicative fact is which of the elements in the suffix array $SA_{\mathcal{R}}[i]$ also correspond to $B_{\mathcal{R}}[i] = \$$. This can be done by computing the suffix array interval $[l_\$, u_\$]$ for the string which has prepended a \$ to the $k$ length suffix of $X$ and checking whether the range contains indices stored in the lexicographic index of $\mathcal{R}$.

### 4.2.1 FindOverlaps$(X, \tau)$

1. $i \leftarrow |X|$
2. $l \leftarrow C_{\mathcal{R}}(X[i])$
3. $u \leftarrow C_{\mathcal{R}}(X[i] + 1) - 1$
4. $i \leftarrow i - 1$
5. **while** $l \leq u$ and $i \geq 1$ **do**
   (a) **if** $|X| - i + 1 \geq \tau$
      i. $[l_\$, u_\$] \leftarrow$ UpdateBackward$([l, u], \$)$
      ii. **if** $l_\$ \leq u_\$$ OutputOverlaps$(X, [l_\$, u_\$])$
   (b) $[l, u] \leftarrow$ UpdateBackward$([l, u], X[i])$
   (c) $i \leftarrow i - 1$
6. **if** $l \leq u$ OutputContained$(X, [l, u])$

The FindOverlaps procedure was modified to be able to find overlaps of type $(E, E)$ and of type $(B, B)$ as well. When calculating overlaps of type $(E, E)$, note that a suffix of $X$ is matching a reverse complemented suffix of $Y$. So the FindOverlaps is used on the complement of $X$ and the FM-Index of $\mathcal{R}'$. When calculating overlaps of type $(B, B)$, note that a prefix of $X$ is matching a reverse complemented prefix of $Y$. So, the FindOverlaps is used on the reverse complment of $X$ and the FM-Index of $\mathcal{R}$.

If $c_i$ is the number of overlaps for read $R_i$, then FindOverlaps makes at most $|R_i|$ calls to UpdateBackward and a total of $c_i$ iterations in OutputOverlaps. Thus, the complexity FindOverlaps for one read $X$ is $O(|X| + c_X)$. Totalling for each read $R_i$, the time complexity of finding all overlaps is foud to be $O(N + C)$.

Once the entire set of overlaps is collected, the transitive reduction algorithm can be used to compute the string graph and the procedure outlined in [1] can be used to assemble the contigs.

## 4.3 Detecting Irreducible Overlaps Directly

Computing all overlaps in $O(N + C)$ time is efficient, but computing only the irreducible edges in $O(N)$ time would reduce the time required for assembling the string graph. To sieve out the transitive edges, the key idea is that the labels of irreducible edges are prefixes of labels of transitive edges. It is possible to construct the lables of the irreducible edges directly from the suffix array intervals using the FM-index.

Suppose $P$ is a substring in $\mathcal{R}$ and has suffix array interval $[l, u]$. Let $\mathcal{B}$ be the set of symbols that appear in $B_{\mathcal{R}}[l, u]$. A *left extension* of $P$ is a string of the form $aP$, where $a \in \mathcal{B}$. Determining $\mathcal{B}$ takes time proportional to the size of the alphabet $\Sigma$. To determine if a symbol $a$ is in $\mathcal{B}$, $Occ_{\mathcal{R}}(a, u) - Occ_{\mathcal{R}}(a, l-1) > 0$ is a characterizing condition. Repeating this for all symbols lets us compute the set $\mathcal{B}$. If $\$ \in \mathcal{B}$, $P$ is said to be *left terminal*. Thus, one of the reads in $\mathcal{R}$ has $P$ as a prefix.

*Right extensions* of a substring $P$ are defined in a similar fashion. To compute the set of right extensions of $P$, the FM-index of $\mathcal{R}'$ can be used. This is because right extensions of $P$ are also left extensions of $P'$. Call $P$ *right terminal* if $\$ \in B_{\mathcal{R}'}[l', u']$, where $[l', u']$ is the suffix array interval of $P'$. In this event, some read in $\mathcal{R}$ has $P$ as a suffix.

The suffix array intervals of overlapping overlapping prefixes must first be computed using the FM-index and then extended to determine the labels of irreducible edges. This can be achieved using just the FM-index of $\mathcal{R}$. Let $[l, u]$ be the suffix array interval for a matching prefix $P$. Ordinarily, to compute the interval for the right extension of $P$, $[l', u']$, BackwardSearch on FM-index of $\mathcal{R}'$.

Using another data structure, which can be computed in constant time, right extensions of $P$ can be computed using the FM-index of $\mathcal{R}$. Let $OccLT_{\mathcal{R}}(a, i)$ be the number of sumbols that are lexicographically smaller than $a$ in $B_{\mathcal{R}}[1, i]$. Let $S = X[i, |X|]$ be a suffix of $X$ and $[l_i, u_i]$ be its corresponding suffix array interval. If $[l'_i, u'_i]$ is the suffix array interval of $S'$ in $\mathcal{R}'$, then the suffix array interval for $S'X[i-1] = S'a$ can be calculated as

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

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

$$\tag{5}$$

Note that $C_{\mathcal{R}} = C_{\mathcal{R}'}$ because $B_{\mathcal{R}}$ and $B_{\mathcal{R}'}$ are permutations of symbols in $\mathcal{R}$. The suffix array interval for $X'[1]$ in the FM-index of $\mathcal{R}'$ is the same as the suffix array interval for $X[|X|]$ in the FM-index of $\mathcal{R}$. This lets us perform simultaneous forward and backward updates to compute the interval corresponding to the right extensions and prefixes.

#### 4.3.1 UpdateForwardBackward$([l, u, l', u'], a, \mathcal{F})$

1. $l' \leftarrow l' + (OccLT_{\mathcal{F}}(a, u) - OccLT_{\mathcal{F}}(a, l-1))$
2. $u' \leftarrow l' + (Occ_{\mathcal{F}}(a, u) - Occ_{\mathcal{F}}(a, l-1) - 1)$
3. $[l, u] \leftarrow$ UpdateBackward$(l, u, a, \mathcal{F})$
4. **return** $[l, u, l', u']$

To compute the set of irreducible overlaps for a read $X$, first the set $\mathcal{I}$ of interval pairs for prefixes that match a suffix of $X$.

#### 4.3.2 FindIntervals$(X, \tau)$

1. $\mathcal{I} \leftarrow \phi$
2. $i \leftarrow |X|$
3. $l \leftarrow C(X[i])$
4. $u \leftarrow C(X[i]+1) - 1$
5. $[l', u'] \leftarrow [l, u]$
6. $i \leftarrow i - 1$
7. **while** $l \leq u$ and $i \geq 1$ **do**
   (a) **if** $|X| - i + 1 \geq \tau$ **then**
      i. $[l_\$, u_\$, l'_\$, u'_\$] \leftarrow$ UpdateForwardBackward$([l, u, l', u'], \$, \mathcal{R})$
      ii. **if** $l_\$ \leq u_\$$ **then**
      $\mathcal{I} \leftarrow \mathcal{I} \cup [l_\$, u_\$, l'_\$, u'_\$]$
   (b) $[l, u, l', u'] \leftarrow$ UpdateForwardBackward$([l, u, l', u'], X[i], \mathcal{R})$
   (c) $i \leftarrow i - 1$
8. **return** $\mathcal{I}$

Every interval pair in $\mathcal{I}$ is tested to check whether a read corresponding to an interval set is right terminal. The intervals corresponding to right terminal reads describe prefixes of reads which form irreducible edges with $X$. These are returned. If no read in the interval set is right terminal, then a subset of intervals corresponding to each right extension of $\mathcal{I}$ is computed and among these intervals, irreducible edges are searched for.

To handle overlaps among reads from opposite strands, FindIntervals is used to find intervals for overlaps from the same strand as $X$ and from the opposite strand as $X$, using the complement of $X$. When extending an interval found by the complement of $X$, extend it using the complement of the corresponding character.

#### 4.3.3 ExtractIrreducible($\mathcal{I}$)

1. **if** $\mathcal{I} = \phi$ **then return** $\phi$
2. $\mathcal{L} \leftarrow \phi$
3. **for all** $[l, u, l', u'] \in \mathcal{I}$ **do**
   (a) $[l'_\$, u'_\$, l_\$, u_\$] \leftarrow$ UpdateForwardBackward($[l', u', l, u], \$, \mathcal{R}'$)
   (b) **if** $l_\$ \leq u_\$$ **then** $\mathcal{L} \leftarrow \mathcal{L} \cup [l_\$, u_\$]$
4. **if** $\mathcal{L} \neq \phi$ **then return** $\mathcal{L}$
5. **for all** $a \in \Sigma$ **do**
   (a) $\mathcal{I}_a \leftarrow \phi$
   (b) **for all** $[l, u, l', u'] \in \mathcal{I}$ **do**
      i. $[l'_a, u'_a, l_a, u_a] \leftarrow$ UpdateForwardBackward($[l', u', l, u], a, \mathcal{R}'$)
      ii. **if** $l_a \leq u_a$ **then** $\mathcal{I}_a \leftarrow \mathcal{I}_a \cup [l_a, u_a, l'_a, u'_a$
   (c) $\mathcal{L} \leftarrow \mathcal{L} \cup$ ExtractIrreducible($\mathcal{I}_a$)
6. **return** $\mathcal{L}$

The time complexity of procedures to compute the irreducible overlaps, involves the computation of the labels of irreducible edges, which are stored in $\mathcal{L}$. Let $L_i$ be the label of an irreducible edge $i$. Say at most $k_i$ intervals are updated when computing $L_i$. These are the reads in $\mathcal{R}$ which have an edge label containing $L_i$. The total length of all the irreducible edges is $E_{tot} = \sum_i |L_i| k_i$. This is the total number of interval updates performed by ExtractIrreducible. Each interval update takes $O(1)$ time. The number of times edge $i$ is used in the set of paths spelling the reads in $\mathcal{R}$ is $k_i$ and accounts for $|L_i|k_i$ symbols in $\mathcal{R}$. Thus, $E_{tot} \leq N$. Thus, ExtractIrreducible is $O(N)$. Since FindIntervals is $O(N)$, the whole procedure to compute irreducible overlaps is $O(N)$.

# 5 Experiments by Authors

The authors of [2] implemented the algorithms described in this report. The entire problem was divided into three stages, indexing - involving the construction of suffix array and FM-index for the set of reads, overlap - which computes the set of overlaps between reads, and assembly - which builds the string graph and compacts the unambiguous paths and outputs the contigs.

Both the exhaustive algorithm, which computes the set of all overlaps, builds the string graph using Myers' transitive reduction algorithm, and then computes the contigs, and the direct algorithm, which computes the set of irreducible overlaps, creates the string graph, and then computes the set of contigs were implemented.

To test the effects of sequence depth on computational complexity, E. coli read data was simulated with mean sequence depth between 5× and 100×. After building indices for the reads, the time to compute overlaps in direct mode and exhaustive mode was found with $\tau = 27$. The results, shown in Figure 1, show that the direct overlap algorithm scales linearly with sequence depth and the exhaustive overlap algorithm scales above linearly.

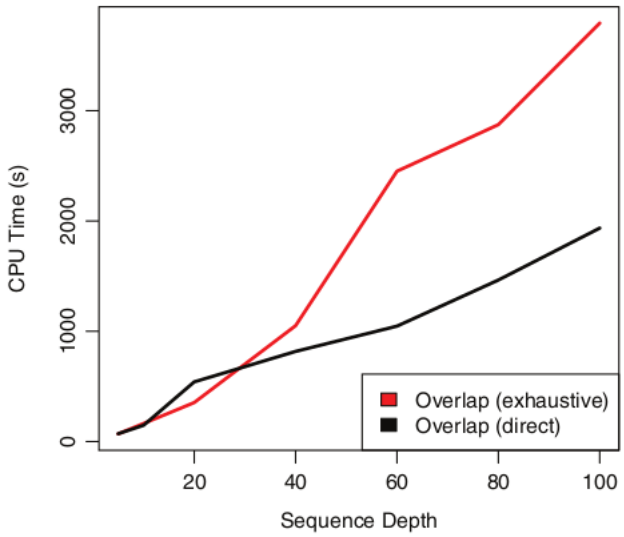


Figure 1: CPU Time vs Sequence Depth for computing Overlaps

The quality of the assembled contigs was ascertained by aligning the results of the direct overlap algorithm with the reference genome using bwa-sw. No contig was found to be misassembled.

To test the effectiveness of the algorithm at larger scales, 100bp error free reads from human chromosomes 22, 15, 7, and 2 were used simulated at an average of 20× coverage for each chromosome. Figure 2 shows a summary of the results as desribed in [2].

The running time of the direct and exhaustive modes were found to be comparable. The assembly was faster for the direct mode and required less memory. The major bottleneck was the indexing. Based on the data chromosome 2, indexing the entire human chromosome would require about 4.5 days and 700GB of memory to index 20× sequence data.

# References

[1] Eugene W. Myers, The fragment assembly string graph, Bioinformatics, Volume 21, Issue suppl_2, Pages ii79-ii85, https://doi.org/10.1093/bioinformatics/bti1114

[2] Simpson JT, Durbin R. Efficient construction of an assembly string graph using the FM-index. Bioinformatics. 2010 Jun 15;26(12):i367-73. doi: 10.1093/bioinformatics/btq217. PMID: 20529929; PMCID: PMC2881401.

| | 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.

Figure 2: Simulation Results for Human Chromosomes