Succinct Data Structures for Whole-Genome Shotgun Assembly

de Bruijn assembly graph using bit vector
VS
string graph using FM-index

Presentation by Adam Przedniczek

2012-04-12

This presentation was based on the following publications:
Efficinet construction of an assembly string graph using the FM-index by Jared T. Simpson and Richard Durbin.
Succinct data structures for assembling large genomes by Thomas C. Conway and Andrew J. Bromage.
1 Foreword
   - References

2 The de Bruijn assembly graph approach
   - The de Bruijn assembly graph
   - Effects of sequencing errors
   - Succinct data structure for de Bruijn assembly graph
   - Results

3 The Meyers’ string graph approach
   - The formulation of a string graph
   - Detecting overlaps
   - Detecting only irreducible overlaps
   - Results
Reference

This publication was based mainly on the following publications:

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

- *Succinct data structures for assembling large genomes*
  Thomas C. Conway, Andrew J. Bromage.

The list of supplementary materials is given below:

- *Assembly Algorithms for Next-Generation Sequencing Data*
  Jason R. Miller, Sergey Koren, Granger Sutton.

- *Indexing Compressed Text*
  Paolo Ferragina, Giovanni Manzini.

- *Compressed Full-Text Indexes*
  Gonzalo Navarro, Veli Mäkinen.

- *Compressed Text Indexes: From Theory to Practice*
  Paolo Ferragina, Rossano Venturini.

- *Linear Suffix Array Construction by Almost Pure Induced-Sorting*
  Ge Nong, Sen Zhang, Wai Hong Chan.

- *Fast Algorithms for Sorting and Searching Strings*
  Jon L. Bentley, Robert Sedgewick.
Succinct data structures for assembling large genomes

Thomas C. Conway and Andrew J. Bromage
The de Bruijn graph

Let $\Sigma$ be an alphabet (e.g. $\Sigma = \{A, T, C, G\}$) and let $k, \rho \in \mathbb{N}_+$ where $\rho = k + 1$.

The de Bruijn graph is defined as follows:

$G_* = \langle V_*, E_* \rangle$

$V_* = \{ s : s \in \Sigma^k \}$

$E_* = \{ \langle s_f, s_t \rangle : s_f[1, k) = s_t[0, k - 1) \}$

Note

Strings are indexed from 0. $s[i, j]$ denotes $s[i, j) = s_i, s_{i+1}, \ldots, s_{j-1}$. The vertices represent all $k$-mers over $\Sigma$ and edges correspond to $k - 1$ letter overlaps between them (edges are labeled with $\rho - mers$).

Figure: A de Bruijn graph of degree 3 over 2 letter alphabet.
The de Bruijn assembly graph

Let $S$ be a set of all reads. The de Bruijn assembly graph is the subgraph of a de Bruijn graph with edges annotated with a number of times a relevant $\rho - mer$ is observed in sequenced reads.

$G_S = < V_S, E_S >$  
$V_S = F_S \cup T_S$

$E_S = \{s[i, i + \rho) : \forall s \in S \ 0 \leq i < |s| - k\}$

$F_S = \{s[1, k + 1) : \forall e \in E_S\}$

$T_S = \{s[0, k) : \forall e \in E_S\}$

Reverse Complements

In assembly process we use DNA sequences that can be read in two directions: forward and reverse complement. We store this sequences separately which makes our graph symmetric.
The de Bruijn assembly graph

(a) Sequence: GCTTTCGACGTTTCA
    Reverse complement: TGAAACGTGGAAAGC

(b) Figure: A simple de Bruijn assembly graph.

(c) 

<table>
<thead>
<tr>
<th>ρ-mer</th>
<th>count</th>
<th>ρ-mer</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAC</td>
<td>1</td>
<td>GAAA</td>
<td>2</td>
</tr>
<tr>
<td>AAAG</td>
<td>1</td>
<td>GACG</td>
<td>1</td>
</tr>
<tr>
<td>AACC</td>
<td>1</td>
<td>GCTT</td>
<td>1</td>
</tr>
<tr>
<td>AACG</td>
<td>1</td>
<td>GTCG</td>
<td>1</td>
</tr>
<tr>
<td>ACAC</td>
<td>1</td>
<td>GTTC</td>
<td>1</td>
</tr>
<tr>
<td>CGAT</td>
<td>1</td>
<td>TCGA</td>
<td>2</td>
</tr>
<tr>
<td>CGAC</td>
<td>1</td>
<td>TGAA</td>
<td>1</td>
</tr>
<tr>
<td>CGTC</td>
<td>1</td>
<td>TTCA</td>
<td>1</td>
</tr>
<tr>
<td>CGTT</td>
<td>1</td>
<td>TTCC</td>
<td>1</td>
</tr>
<tr>
<td>CTTT</td>
<td>1</td>
<td>TTTC</td>
<td>2</td>
</tr>
</tbody>
</table>
How sequencing errors affects de Bruijn assembly graph

- Sequencing errors give rise to unique k-mers and hence spurious edges that occur only once.
- It can be assumed that the number of spurious edges is proportional to the volume of input data.
- The number of true edges is proportional to the size of the genome.
- Once enough the number of reads has sufficient coverage, the spurious edges will outnumber the true edges.

Figure: The number of true edges can be easily exceeded by spurious ones.
The most straightforward attempt

The simplest attempt is to store vertices as records (e.g. `struct` in C/C++) and edges as pointers:

- Assuming $k \leq 32$ we can represent k-mers as 64 bit integers (8 B per node).
- 32 bit edge counts to 4 possible successors (16 B per node).
- 64 bit pointers to 4 possible successors (32 B per node).

Finally, this representation require at least 56 B per node. Moreover, we must somehow locate nodes e.g. with simple hash table which will require next 16B for each entry.

For human genome and $k = 25$ de Bruijn assembly graph has about 4.8 billion nodes and will require over 250GB plus over 70GB for hash table.
Estimating the lower bound of space requirements

Simple observation

The graph nodes don’t have to be stored explicitly, but rather their labels can be inferred from the edge:

\[ \text{fromNode}(e) = e[0, \rho - 1] \quad \text{toNode}(e) = e[1, \rho] \]

Encoding de Bruijn assembly graph as subset of de Bruijn graph

The minimum number of bits required to encode edges of assembly graph, is

\[ \log \left( \frac{|\Sigma|^\rho}{|E_S|} \right) \]

For human genome and \( k = 25 \) that gives

\[ \log \left( \frac{4^{26}}{4, 796, 397, 453} \right) \approx 12 \text{ GB} \]
Estimating the lower bound of space requirements

Having some approximation of the lower bound for the space required to represent assembly graph, we examine the number of bits needed to represent single edge by dividing this constraint by the number of edges.

Curious observation

The number of bits per edge monotonically decreases as the number of edges increases.
How to represent edges using bit vector

Bare essential functionality supported by bit vectors

There’s given a bit vector $b$ indexed from 0 and two functions:

$\text{rank}_b(p) = \sum_{0 \leq i < p} b_i$ \quad $\text{select}_b(i) = \max\{p < |b| : \text{rank}_b(p) \leq i\}$

Representing edges as 1’s in bit vector $E_S$

The alphabet is enumerated e.g. $A = 0, C = 1, G = 2, T = 3$. Each $\rho$ – mer is treated as $2^{\rho}$ bit integer. There’s $4^\rho$ bit vector, in which 1’s denote presence of respective edge in assembly graph.

Computing outgoing edges

Let $n$ denotes value of given $k$ – mer node. We can compute numbers representing possible outgoing edges as follows

$succ_{E_S} = \{\text{select}_{E_S}(r) : r \in [\text{rank}_{E_S}(4n), \text{rank}_{E_S}(4n + 4))\}$
How to succinctly represent bit vectors

In the most common case of human genome and $k = 25$, the bit vector would require **512 TB**.

This problem can be solved by exploiting extreme sparsity of the bit vector. Only $4.8 \times 10^9$ of total $4^{26} \approx 4.5 \times 10^{15}$ bits are set to 1.

The bit vector $B$ with operations rank and select can be implemented using space of:

- $\nu + o(\nu)$ bits for dense version (rank & select in $O(1)$ time).
- $\nu H_0(B) + O(\frac{\nu \log \log \nu}{\log \nu})$ (rank & select in $O(1)$ time).
- $\mu \log \frac{\nu}{\mu} + 1.92\mu + o(\mu)$ (rank in $O(\log \frac{\nu}{\mu}) + O(\frac{\log^4 \mu}{\log \nu}$) time and select in $O(\frac{\log^4 \mu}{\log \nu}$) time.

where $\nu$ - vectors total capacity, $\mu$ - number of bits set to 1.
How to succinctly store the edge counts

The edge counts could be stored in vector of $|E_S|$ 32 bit entries, but that’s the waste of memory space. It’s much better to use approach described by Brisaboa et al. (2009):

- Each count is divided into 3 parts: the least significant 8 bits, the middle 8 bits and the most significant 16 bits. Only non-zero parts are stored respectively in dense arrays: L, M, H.
- Corresponding to L and M arrays we store succinct bit vectors $B_L$ and $B_M$.
- In $B_L$ we mark 1 this entries for which the middle part or the most significant part is non-zero.
- In $B_M$ we mark 1 this entries for which the most significant part is non-zero.
Proof-of-concept assembly procedure

To demonstrate the feasibility of assembly process, 4 separate phases were performed:

1. Extract $\rho - mers$ form sequenced reads. This step ends with succinct graph (and edge counts) representation.
2. Discard low frequency edges (that probably occured due to sequencing errors).
3. Simplify the graph by tip removal.
4. Find possible contigs (non-branching paths) by performing deep-first traversal.
The proof-of-concept assembly pipeline was benchmarked on sequence data NA18507 (Yoruban individual from *Bentley et al. 2008*) with $k = 27$.

It was run on single machine equipped with $8 \times 2.0\text{GHz}$ Operton cores and 32 GB memory.

<table>
<thead>
<tr>
<th>After</th>
<th>Edges</th>
<th>Graph</th>
<th>Counts</th>
<th>Total</th>
<th>Build [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>12,292,819,311</td>
<td>40.8</td>
<td>11.5</td>
<td>52.3</td>
<td>2080</td>
</tr>
<tr>
<td>Step 2</td>
<td>4,799,738,381</td>
<td>15.1</td>
<td>4.5</td>
<td>19.6</td>
<td>46</td>
</tr>
<tr>
<td>Step 3</td>
<td>3,840,690,715</td>
<td>12.2</td>
<td>3.6</td>
<td>15.8</td>
<td>845</td>
</tr>
</tbody>
</table>

*Table: Memory usage and robustness of proof-of-concept pipeline*
Efficient construction of an assembly string graph using the FM-index
Jared T. Simpson and Richard Durbin
Notation

Let $X, Y$ are strings over an alphabet $\Sigma = \{A, T, C, G\}$.

- $X'$ denotes reverse of $X$.
- $\overline{X}$ denotes the reverse complement of $X$.
- $\mathcal{R}$ denotes an indexed set of sequenced reads, which can originate from either DNA strand. We assume that there're no sequencing errors.
- Reads $X$ and $Y$ overlap only if a prefix of $X$ is equal to a suffix of $Y$ (or vice versa) or if the reverse complement of one of them overlaps the other.
- Read $X$ is contained within $Y$ ($X$ and $Y$ are said to be redundant) if $X$ is a substring of (perhaps identical to) $Y$.
- $\tau$ denotes the minimum acceptable overlap length.
The main concept of string graph

A string graph represents the proper overlap relationship between sequence reads. In such a graph there’re vertices for all non-contained reads. Two vertices are connected by an edge only if their corresponding reads overlap.

\[
\begin{align*}
R_1 & \quad ACATACGATAACA \\
R_2 & \quad TACGATAACAGTT \\
R_3 & \quad GATACAGTTGCA
\end{align*}
\]

**Figure:** A simple string graph (almost correctly).
Additional definitions

- Let $X$, $Y$ originate from the same strand and they share a proper overlap $X[s_{xy}, e_{xy}] = Y[s_{yx}, e_{yx}]$.
- There’s a bidirected edge $X \leftrightarrow Y$ connecting them, which is annotated with following properties: $type_{xy}$, $type_{yx}$, $label_{xy}$ and $label_{yx}$:
  \[ type_{xy} = \begin{cases} 
  B & \text{if } s_{xy} = 1 \\
  E & \text{if } e_{xy} = |X| 
  \end{cases} \]
  \[ label_{xy} = \begin{cases} 
  Y[e_{yx} + 1, |Y|] & \text{if } s_{yx} = 1 \\
  Y[1, s_{yx} - 1] & \text{if } e_{yx} = |Y| 
  \end{cases} \]

- The concatenation of $X \cdot label_{xy}$ is an assembly of reads $X$ and $Y$ ($label_{xy}$ is unmatched portion of $Y$).
- If $X$, $Y$ originate from opposite strands, their overlap is reverse-complemented $X[s_{xy}, e_{xy}] = Y[s_{yx}, e_{yx}]$. 

Irreducible and transitive edges

- Reads $R_1, R_2, R_3$ mutually overlap.
- $R_2$ and $R_3$ overlap $R_1$ on the same end and hence they must share a common substring. This substring is a prefix or suffix of one of them.
- There's a path visiting all vertices once: $R_1 \rightarrow R_2 \rightarrow R_3$. The string corresponding to this path is a valid assembly.
- The transitive edge $R_1 \leftrightarrow R_3$ is redundant, thus it can be removed leaving only irreducible edges.
- The label of irreducible edge is a prefix of label of the transitive edge.
A brief recall of FM-index

**Suffix array**

Let $X$ be a string terminated with $\$$(lowest) symbol.
Suffix array $SA_X$ contains starting positions of lexographically sorted suffixes of $X$. $SA_X[i] = p$ denotes a string $X[p : |X|]$ is i-th lexographically lowest suffix of $X$.

**Burrows–Wheeler transform**

Burrows–Wheeler transform $BWT_X$ contains symbols preceding lexographically sorted suffixes of $X$. $BWT_X[i]$ is a symbol preceding the suffix starting at position $SA_X[i]$.

$$BWT_X[i] = \begin{cases} X[SA_X[i] - 1] & \text{if } SA_X[i] > 1 \\ \$ & \text{if } SA_X[i] = 1 \end{cases}$$
A brief recall of FM-index

\( C_X \)

\( C_X(a) \) is the number of symbols in \( X \) which are lexographically lower than symbol \( a \).

\( \text{Occ}_X \)

\( \text{Occ}_X(a, i) \) is the number of occurrences of \( a \) in \( \text{BWT}_X[1 : i] \).

\( \text{OccLT}_X \)

\( \text{OccLT}_X(a, i) \) is the number of occurrences of symbol lexographically lower than symbol \( a \) in \( \text{BWT}_X[1 : i] \).
FM-index backward search

**Algorithm 1:** UPDATE-BACKWARD([l, u], a)

\[
l \leftarrow C_X(a) + \text{Occ}_X(a, l - 1) + 1;
\]

\[
u \leftarrow C_X(a) + \text{Occ}_X(a, u);
\]

return [l, u]

**Algorithm 2:** BACKWARD-SEARCH(Q)

Result: The interval in $SA_X$ for the pattern Q

\[
i \leftarrow |Q|;
\]

\[
l \leftarrow C_R(Q[i]) + 1;
\]

\[
u \leftarrow C_R(\text{succ}(Q[i]));
\]

\[
i \leftarrow i - 1;
\]

while \( l \leq u \) and \( i \geq 1 \) do

\[
[l, u] \leftarrow \text{UPDATE-BACKWARD}([l, u], Q[i]);
\]

\[
i \leftarrow i - 1;
\]

return [l, u]

**Time complexity of BACKWARD-SEARCH** is only $O(|Q|)$.
### Backward search example

<table>
<thead>
<tr>
<th>i</th>
<th>SA</th>
<th>BWT</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>i</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>p</td>
<td>i$</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>s</td>
<td>ippi$</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>s</td>
<td>issippi$</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>m</td>
<td>ississippi$</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>$</td>
<td>mississippi$</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>p</td>
<td>pi$</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>i</td>
<td>ppi$</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>s</td>
<td>sippi$</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>s</td>
<td>sissippi$</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>i</td>
<td>ssippi$</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>i</td>
<td>ssissippi$</td>
</tr>
</tbody>
</table>

\( X = \text{'mississippi$'} \) and \( Q = \text{'mis'} \)
\( C[\$] = 0, C[i] = 1, C[m] = 5, C[p] = 6, C[s] = 8 \)

1. Matching against \( Q[3] = s \)
   \( l = C_X[s] + 1 = 9 \)
   \( u = C_X[succ(s)] = 12 \)

2. Matching against \( Q[2] = i \)
   \( l = C_X(i) + Occ_X(i, 9 - 1) + 1 = 4 \)
   \( u = C_X(i) + Occ_X(i, 12) = 5 \)

3. Matching against \( Q[1] = m \)
   \( l = C_X(m) + Occ_X(m, 4 - 1) + 1 = 6 \)
   \( u = C_X(m) + Occ_X(m, 5) = 6 \)

The final range is \([6, 6]\) and hence 'mis' occurs at position \( SA_X[6] = 1 \).
Generalizing FM-index for multiple strings

**Generalized suffix array** $\mathbf{SA}_\mathcal{T}$

Let $\mathcal{T}$ be an indexed set of strings (each terminated by unique sentinel symbol $\$\$).

Generalized suffix array fullfil condition $\mathbf{SA}_\mathcal{T}[i] = (j, k)$ if and only iff $i$-th lexographically lowest suffix in $\mathcal{T}$ is $\mathcal{T}[k : |\mathcal{T}_j|]$.

**Generalized Burrows-Wheeler transform** $\mathbf{BWT}_\mathcal{T}$

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

Definitions of $C_\mathcal{T}(a)$, $\text{Occ}_\mathcal{T}(a, i)$ and $\text{OccLT}_\mathcal{T}(a, i)$ do not change.
Building an FM-index from a set of reads

At the very beginning, the FM-index of $R$ must be built and hence suffix array of $R$ must be computed. The modification of Nong-Zhang-Chan algorithm (with multikey quicksort) is used. For further information check:

- *Linear Suffix Array Construction by Almost Pure Induced-Sorting* Ge Nong, Sen Zhang, Wai Hong Chan.

We also compute FM-index for the set of reversed reads $R'$ and lexicographic index of $R$ (order of entire reads in lexicographic order).
Exhaustive overlap detection

Algorithm 3: FIND-OVERLAPS(X, τ)

Result: The reads from R that overlap X by at least τ symbols

\[
i \leftarrow |X|;
\]

\[
l \leftarrow C_R(X[i]) + 1;
\]

\[
u \leftarrow C_R(succ(X[i]));
\]

\[
i \leftarrow i - 1;
\]

while \( l \leq u \) and \( i \geq 1 \) do

\[
\text{if } |X| - i + 1 \geq \tau \text{ then}
\]

\[
[l, u] \leftarrow \text{UPDATE-BACKWARD}([l, u], $);
\]

\[
\text{if } l \leq u \text{ then}
\]

\[
\text{OUTPUT-OVERLAPS}(X, [l, u]);
\]

\[
[l, u] \leftarrow \text{UPDATE-BACKWARD}([l, u], X[i]);
\]

\[
i \leftarrow i - 1;
\]

if \( l \leq u \) then

\[
\text{OUTPUT-CONTAINED}(X, [l, u]);
\]
Exhaustive overlap detection

- The time complexity of FIND-OVERLAPS is only $O(N + C)$ where $N = \sum_{i=1}^{\left|\mathcal{R}\right|} |\mathcal{R}_i|$ and $C$ is total number of all overlaps found.

- Remember that FIND-OVERLAPS($X$, $\tau$) (with FM-index of $\mathcal{R}$) finds only edges of type $(E, B)$.

- Edges of type $(B, B)$ are found by FIND-OVERLAPS($\overline{X}$, $\tau$) (with FM-index of $\mathcal{R}$).

- Edges of type $(E, E)$ are found by FIND-OVERLAPS on the complement of $X$ (not $\overline{X}$) and the FM-index of $\mathcal{R}'$. 
Direct overlap detection

To output only irreducible edges make use of the property:

*The label of irreducible edge is a prefix of label of the transitive edge.*

Let $S$ be a substring that occurs in $R$ (its $SA$ interval is $[l, u]$).

- A *left extension* of $S$ is $\forall a \in BWT[l,u]a \cdot S$.
- $S$ is a *left terminal* if $S \in BWT[l,u]$ ($S$ is a prefix of some read).
- Similarly a *right extension* of $S$ is of form $S \cdot a$.
- $S$ is a *right terminal* if it is a suffix of some read.

To find all irreducible edges of a read $X$ we search for all $SA$ intervals containing the prefixes of reads that overlap a suffix of $X$, then iteratively extend them rightwards until a right terminal is found.
Searching backwards and extending forwards

Let $\mathcal{R}$ is a singleton and we’re trying to find $X = 'sis'$. We have already matched suffix $S = X[2, 3] = 'is'$ inside interval $[l_2 = 4, u_2 = 5]$. $S' = 'si'$ is located inside interval $[l'_2 = 9, u'_2 = 10]$. Now, we want to match $X[1] = 's'$. We can compute the interval $[l'_1, u'_1]$ for $S' \cdot s$ in the following way:

\[ l'_1 = l'_2 + (\text{OccLT}_{\mathcal{R}}(s, u_2) - \text{OccLT}_{\mathcal{R}}(s, l_2 - 1)) = 10 \]
\[ u'_1 = l'_1 + (\text{Occ}_{\mathcal{R}}(s, u_2) - \text{Occ}_{\mathcal{R}}(s, l_2 - 1) - 1) = 10 \]

<table>
<thead>
<tr>
<th>i</th>
<th>SA</th>
<th>BWT</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>i</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>p</td>
<td>i$</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>s</td>
<td>ippi$</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>s</td>
<td>issippi$</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>m</td>
<td>ississippi$</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>$</td>
<td>mississippi$</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>p</td>
<td>pi$</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>i</td>
<td>ppi$</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>s</td>
<td>sippi$</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>s</td>
<td>sippi$</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>i</td>
<td>ssippi$</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>i</td>
<td>ssissippi$</td>
</tr>
</tbody>
</table>

Table: $\mathcal{R}_1 = 'mississippi$$' 

<table>
<thead>
<tr>
<th>i</th>
<th>SA</th>
<th>BWT</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>m</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>s</td>
<td>im$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$</td>
<td>ippiississim$</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>s</td>
<td>issim$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>p</td>
<td>ississim$</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>i</td>
<td>m$</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>p</td>
<td>pississim$</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>i</td>
<td>ppississim$</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>s</td>
<td>sim$</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>s</td>
<td>ssim$</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>i</td>
<td>ssim$</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>i</td>
<td>ssissim$</td>
</tr>
</tbody>
</table>

Table: $\mathcal{R}'_1 = 'ippississim$$'
The procedure performing forward and backward update

**Algorithm 4:** UPDATE-FWD-BWD([l, u, l', u'], a, F)

\[
\begin{align*}
    l' & \leftarrow l' + \text{OccLT}_F(a, u) - \text{OccLT}_F(a, l - 1); \\
    u' & \leftarrow u' + \text{Occ}_F(a, u) - \text{Occ}_F(a, l - 1) - 1; \\
    [l, u] & \leftarrow \text{UPDATE-BACKWARD}([l, u], a, F); \\
    \text{return} & \ [l, u, l', u'];
\end{align*}
\]
Finding the set of intervals

**Algorithm 5:** FIND-INTERVALS($X, \tau$)

$I \leftarrow \emptyset$;

$i \leftarrow |X|$;

$l \leftarrow C(X[i]) + 1$;

$u \leftarrow C(succ(X[i]))$;

$[l', u'] \leftarrow [l, u]$;

$i \leftarrow i - 1$;

**while** $l \leq u$ and $i \geq 1$ **do**

**if** $|X| - i + 1 \geq \tau$ **then**

$[l$, $u$, $l'$, $u'] \leftarrow \text{UPDATE-FWD-BWD}([l, u, l', u'], X[i], \mathcal{R})$;

**if** $l \leq u$ **then**

$I \leftarrow I \cup [l, u, l', u']$;

$[l, u, l', u'] \leftarrow \text{UPDATE-FWD-BWD}([l, u, l', u'], X[i], \mathcal{R})$;

$i \leftarrow i - 1$;

**return** $I$;
Extracting the intervals of the irreducible edges

**Algorithm 6: **EXTRACT-IRREDUCIBLE($\mathcal{I}$)

1. if $\mathcal{I} = \emptyset$ then
   - return $\emptyset$;

2. $\mathcal{L} \leftarrow \emptyset$;

3. foreach $[l_s, u_s, l'_s, u'_s] \in \mathcal{I}$ do
   - $[l'_s, u'_s, l_s, u_s] \leftarrow$ UPDATE-FWD-BWD($[l', u', l, u], $, $\mathcal{R}'$);
   - if $l_s \leq u_s$ then
     - $\mathcal{L} \leftarrow \mathcal{L} \cup [l_s, u_s]$;

4. if $\mathcal{L} \neq \emptyset$ then
   - return $\mathcal{L}$;

5. foreach $a \in \Sigma$ do
   - $\mathcal{I}_a \leftarrow \emptyset$;
   - foreach $[l, u, l', u'] \in \mathcal{I}$ do
     - $[l'_a, u'_a, l_a, u_a] \leftarrow$ UPDATE-FWD-BWD($[l', u', l, u], a, \mathcal{R}'$);
     - if $l_a \leq u_a$ then
       - $\overline{\mathcal{I}}_a \leftarrow \mathcal{I}_a \cup [l_a, u_a, l'_a, u'_a]$;

6. $\mathcal{L} \leftarrow \mathcal{L} \cup$ EXTRACT-IRREDUCIBLE($\mathcal{I}_a$);

return $\mathcal{L}$;
Proof-of-concept assembly procedure

To demonstrate the performance of assembly process, 3 separate phases were performed:

1. **Index**: The suffix array and FM-index were constructed.
2. **Overlap**: The set of overlaps between the reads was computed.
3. **Assemble**: The string graph was computed (with transitive reduction if necessary). Then all unambiguous paths were compacted and finally a set of contigs was written out.
The tests were conducted on *Escherichia coli* reads with mean sequence depth varying from $5\times$ to $100\times$ and $\tau = 27$.

**Figure:** Exhaustive and direct overlap algorithms.
## Scaling the algorithm with the size of human genome

<table>
<thead>
<tr>
<th></th>
<th>chr 22</th>
<th>chr 15</th>
<th>chr 7</th>
<th>chr 2</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chr. size (Mb)</td>
<td>34.9</td>
<td>81.7</td>
<td>155.4</td>
<td>238.2</td>
<td>6.8</td>
</tr>
<tr>
<td>Number of reads (M)</td>
<td>7.0</td>
<td>16.3</td>
<td>31.1</td>
<td>47.6</td>
<td>6.8</td>
</tr>
<tr>
<td>Contained reads (k)</td>
<td>684</td>
<td>1668</td>
<td>3103</td>
<td>4709</td>
<td>6.9</td>
</tr>
<tr>
<td>Contained (%)</td>
<td>9.8</td>
<td>10.2</td>
<td>10.0</td>
<td>9.9</td>
<td>–</td>
</tr>
<tr>
<td>Transitive edges (M)</td>
<td>38.0</td>
<td>93.0</td>
<td>177.7</td>
<td>274.6</td>
<td>7.2</td>
</tr>
<tr>
<td>Irreducible edges (M)</td>
<td>6.3</td>
<td>14.9</td>
<td>28.7</td>
<td>44.4</td>
<td>7.0</td>
</tr>
<tr>
<td>Assembly N50 (kbp)</td>
<td>4.0</td>
<td>4.6</td>
<td>4.2</td>
<td>4.7</td>
<td>–</td>
</tr>
<tr>
<td>Longest contig (kbp)</td>
<td>31.9</td>
<td>47.7</td>
<td>53.1</td>
<td>48.6</td>
<td>–</td>
</tr>
<tr>
<td>Index time (s)</td>
<td>2606</td>
<td>9743</td>
<td>19779</td>
<td>30866</td>
<td>11.8</td>
</tr>
<tr>
<td>Overlap -e time (s)</td>
<td>2657</td>
<td>6572</td>
<td>12970</td>
<td>18060</td>
<td>6.8</td>
</tr>
<tr>
<td>Overlap -d time (s)</td>
<td>2885</td>
<td>6750</td>
<td>13271</td>
<td>19437</td>
<td>6.7</td>
</tr>
<tr>
<td>Assemble -e time (s)</td>
<td>1836</td>
<td>4043</td>
<td>8112</td>
<td>13095</td>
<td>7.1</td>
</tr>
<tr>
<td>Assemble -d time (s)</td>
<td>423</td>
<td>1161</td>
<td>2044</td>
<td>3226</td>
<td>7.6</td>
</tr>
<tr>
<td>Index memory (GB)</td>
<td>8.0</td>
<td>18.6</td>
<td>35.4</td>
<td>54.5</td>
<td>6.8</td>
</tr>
<tr>
<td>Overlap -e mem. (GB)</td>
<td>2.4</td>
<td>5.5</td>
<td>10.5</td>
<td>16.1</td>
<td>6.7</td>
</tr>
<tr>
<td>Overlap -d mem. (GB)</td>
<td>2.4</td>
<td>5.5</td>
<td>10.4</td>
<td>16.1</td>
<td>6.7</td>
</tr>
<tr>
<td>Assemble -e mem. (GB)</td>
<td>5.9</td>
<td>14.2</td>
<td>27.2</td>
<td>41.9</td>
<td>7.1</td>
</tr>
<tr>
<td>Assemble -d mem. (GB)</td>
<td>2.7</td>
<td>6.3</td>
<td>12.1</td>
<td>18.6</td>
<td>6.9</td>
</tr>
</tbody>
</table>

**Succinct Data Structures for Whole-Genome Shotgun Assembly**
Conclusions and time for questions

Thank you for your attention.
Any questions?