Next Generation Sequencing

E.M. Bakker

Several slides are based on/taken from [7].

The Mapping Problem and the Assembly Problem

The Mapping Problem

INPUT: m reads $S_1,\ldots, S_m$ of length l and an approximate reference genome R.

QUESTION: What are the positions $x_1,\ldots, x_m$ along R where each read $S_1,\ldots, S_m$ matches, respectively?

The Assembly Problem

INPUT: m reads $S_1,\ldots, S_m$ of length l.

QUESTION: What is the sequence of the full genome?

The crucial difference between the problems of mapping and assembly is that in case of assembly **we do not have a reference genome**, and we must assemble the full sequence directly from the reads.

The Mapping Problem

The Short Read Mapping Problem on reference genome R

INPUT: m reads $S_1,\ldots, S_m$ of length l and an approximate reference genome R.

QUESTION: What are the positions $x_1,\ldots, x_m$ along R where each read $S_1,\ldots, S_m$ matches, respectively?

For example: after sequencing the genome of a person we want to map it to an **existing sequence of the human genome.**

- The new sample will not be 100% identical to the reference genome:
  - natural variation in the population
  - mismatches or gaps
  - sequencing errors
  - repetitive regions

  Humans are diploid organisms
  - different alleles on the maternal and paternal chromosomes
  - two slightly different reads mapping to the same location (some with mismatches)
- Viruses: high mutation rates, many variations haplotypes => hybrid mapping problem

The Bowtie Algorithm

Another way to **map reads to a reference genome** is given by the Bowtie algorithm, presented in 2009 by Langmead et al.[1].

It solves the mapping problem using a space-efficient indexing scheme.

The indexing scheme used is called the Burrows-Wheeler Transform [2] and was originally developed for data compression purposes.

The Burrows-Wheeler Transform

Applying the Burrows-Wheeler transform BW(T) to the text T = "the next text that i index.":

1. First, we generate all cyclic shifts of T.
2. Next, we sort these shifts lexicographically.
   - define the character '.' as the minimum and we assume that it appears exactly once, as the last symbol in the text.
   - followed lexicographically by ' ' (space)
   - followed by the English letters according to their natural ordering.
   - Call the resulting matrix M.

The transform BW(T) is defined as the sequence of the last characters in the rows of M.

Note that, the last column is a permutation of all characters in the text since each character appears in the last position in exactly one cyclic shift.

Burrows-Wheeler Transform

.the_next_text_that_i_index
_i_index.the_next_text_that
_index.the_next_text_that_i
_next_text_that_i_index.the
_text_that_i_index.the_next
_that_i_index.the_next_text
at_i_index.the_next_text_th
dex.the_next_text_that_i_in
e_next_text_that_i_index.th
ex.the_next_text_that_i_ind
ext_text_that_i_index.the_n
ext_that_i_index.the_next_t
hat_i_index.the_next_text_t
he_next_text_that_i_index.t

Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

Burrows-Wheeler Transform

- Storing BW(T) requires the same space as the size of the text T since it is a permutation of T.

BW(the human genome)

Each base {A,C,T,G} represented by 2 bits => storing the permutation requires 2 times $3x10^9$ bits (*instead of ~30 times $3x10^9$ for storing all indices of T*).

Burrows-Wheeler Transform

The following holds for BW(T):

1. # occurrences of char c in T = # occurrences of char c in BW(T) (BW(T) permutation of the T).

2. The first column of the matrix M can be obtained by sorting BW(T) lexicographically.

3. Determine the number of occurrences of the substring 'xt' in T:
   - BW(T) is the last column of the lexicographical sorting of the shifts.
   - **The character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift).**
   - => consider the interval of 't' in the first column, and check how many of these rows have an 'x' at the last position.

Burrows-Wheeler Transform

.the_next_text_that_i_index
_i_index.the_next_text_that
_index.the_next_text_that_i
_next_text_that_i_index.the
_text_that_i_index.the_next
_that_i_index.the_next_text
at_i_index.the_next_text_th
dex.the_next_text_that_i_in
e_next_text_that_i_index.th
ex.the_next_text_that_i_ind
ext_text_that_i_index.the_n
ext_that_i_index.the_next_t
hat_i_index.the_next_text_t
he_next_text_that_i_index.t

1. Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

Burrows-Wheeler Transform

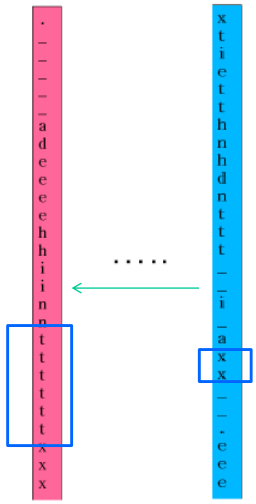
The following holds for BW(T):

1. # occurrences of char c in T = # occurrences of char c in BW(T) (BW(T) permutation of the T).

2. The first column of the matrix M can be obtained by sorting BW(T) lexicographically.

3. Determine the number of occurrences of the substring 'xt' in T:
   - BW(T) is the last column of the lexicographical sorting of the shifts.
   - **The character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift).**
   - => consider the interval of 't' in the first column, and check how many of these rows have an 'x' at the last position.

Sorted BW(T)



BW(T)

2. Recovering the first column (left) by sorting the last column.
3. Determine the number of occurrences of the substring 'xt' in T

Burrows-Wheeler Transform

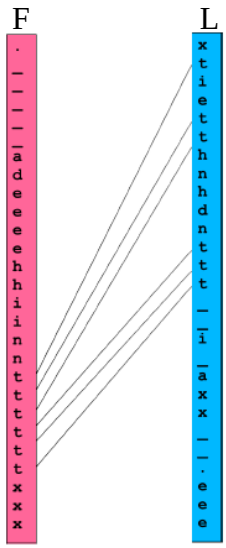
Given BW(T) also the second column can be derived:

- 'xt' appears *twice* in the text, and *three rows* start with an 'x'.
- *Two of the three* must be followed by a 't', where the lexicographical sorting determines which 'x'.
- The third 'x' is followed by a '.' (see first row) => '.' must follow the first 'x' in the first column since '.' is smaller lexicographically than 't'.
- The second and third occurrences of 'x' in the first column are therefore followed by 't'.

Note: We can use the same process to **recover the characters at the second column** for each interval, and then the third, etc.

. the next text that I index	1st x
…	
t text that i index. the next	2nd x
t that i index. the next tex	3rd x
text that i index.the next	
that i index. the next text	
the next text that i index.	

We have actually:



Last-first mapping: Each 't' character in L is linked to its position in F and no crossed links are possible.

The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F.

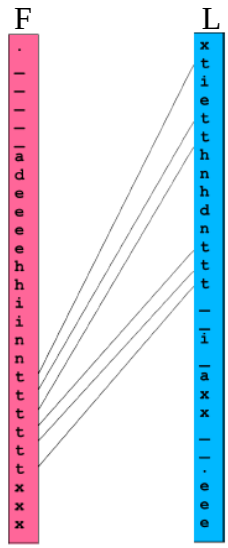
Burrows-Wheeler Transform

The previous two central properties of the BW-transform are captured in the Lemma by Ferragina and Manzini[4]:

Lemma 12.1 (Last-First Mapping):

Let M be the matrix whose rows are all cyclical shifts of T sorted lexicographically, and let L(i) be the character at the last column of row i and F(i) be the first character in that row. Then:

1. In row i of M, L(i) precedes F(i) in the original text: T =…L(i) F(i)…
2. The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F.



Last-first mapping: j-th occurrence of character 't' character in L is linked to its j-th position in F and no crossed links are possible.

Burrows-Wheeler Transform

Proof:

1. Follows directly from the fact that each row in M is a cyclical shift.

2. Let X_j denote the j-th occurrence of character X in L, and let α be the character following X_j in the text and β the character following X_{j+1}.

   Then, since X_j appears above X_{j+1} in L, α appears at the beginning of a row above the row that starts with β.

   The rows are lexicographically ordered, hence α must be equal or lexicographically smaller than β.

   Now clearly $X\ \alpha \leq_{\text{lexicographically}} X\ \beta$ holds.

   Hence, as the rows are lexicographically ordered, if character X_j appears in F it is followed by α, and thus will be above X_{j+1} which is followed by β.

   Thus proofing the Lemma.

Reconstructing the Text

Algorithm UNPERMUTE for reconstructing a text T from its Burrows-Wheeler transform BW(T) utilizing Lemma 12.1 [4]:

- assume the actual text T is of length u
- append a unique $ character (='.') at the end, which is the smallest lexicographically

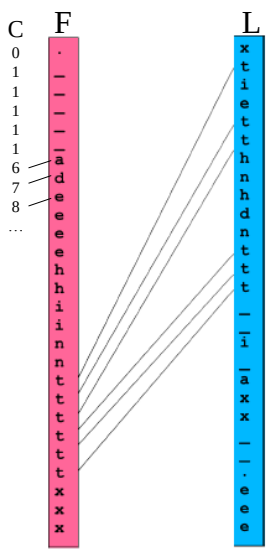
UNPERMUTE[BW(T)]

1. Compute the array C[1,…,|Σ|] : where
   C(**c**) is the number of characters {$, 1,…,**c**-1} in T,
   i.e., the number of characters that are lexicographically smaller than **c**
2. Construct the last-first mapping LF, tracing every character in L to its corresponding position in F:
   LF[i] = C(L[i]) + r(L[i], i) + 1, where
   r(c, i) is the number of occurrences of character **c** in the prefix L[1, i -1]
3. Reconstruct T backwards:

```
s = 1, T(u) = L[1];
for i = u – 1,…, 1
do
  s = LF[s];
  T[i] = L[s];
od;
```

Notice, that C(e) + 1 = 9 is the position of the first occurrence of 'e' in F.

C(e) = 8, as there are 8 characters in T that are smaller than 'e'.



Last-first mapping: Each 't' character in L is linked to its position in F and no crossed links are possible.

Example

T = acaacg$ (u = 6) is transformed to BW(T) = gc$aaac, and we now wish to reconstruct T from BW(T) using UNPERMUTE:

1. Compute array C. For example: C(c) = 4 since there are 4 occurrences of characters smaller than 'c' in T ('$' and 3 occurrences of 'a'). Now C(c) + 1 = 5 is the position of the first occurrence of 'c' in F.
2. Perform the LF mapping. For example, $LF[c_2] = C(c) + r(c,7) + 1 = 6$, and indeed the second occurrence of 'c' in F is at F[6].
3. Determine the last character in T: T(6) = L(1) = 'g'.
4. Iterate backwards over all positions using the LF mapping. For example, to recover the character T(5), we use the LF mapping to trace L(1) to F(7), and then T(5) = L(7) = 'c'.

Remarks

We do not actually hold F in memory, we only keep the array C defined above, of size | Σ |, which we can easily obtained from L.

r(c,7) = # of occurences of 'c' in length 7-1 prefix, used as an offset to obtain the right 'c'.

Determine its location in F and find its corresponding predecessor in L using C(.) and r(.,.)
i.e., find corresponding 'g' using the last-first mapping

Note: '$' = '.'

Example of running UNPERMUTE to recover the original text. Source: [1].

Exact Matching

EXACTMATCH exact matching of a query string P to T, given BW(T)[4] is similar to UNPERMUTE, we use

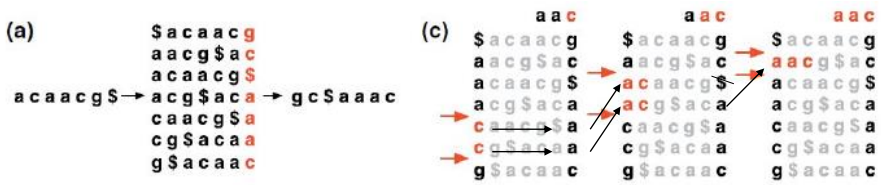
- the same C and r(c, i)
- denote by sp the position of the first row in the interval of rows in M we are currently considering
- denote by ep the position of the first row beyond this interval of rows

=> the interval of rows is defined by the rows sp,…, ep - 1.

EXACTMATCH[P[1,…,p], BW(T)]

1. **c** = P[p]; sp = C[**c**] + 1; ep = C[**c**+1] + 1; i = p - 1;
2. **while** sp < ep **and** i >= 1
   c = P[i];
   sp = C[**c**] + r(**c**, sp) + 1;
   ep = C[**c**] + r(**c**, ep) + 1;
   i = i - 1;
3. **if** (sp == ep) **return** "no match"; else return sp, ep;

P = ‘aac’ ←



Example of running EXACTMATCH to find a query string in the text [1].

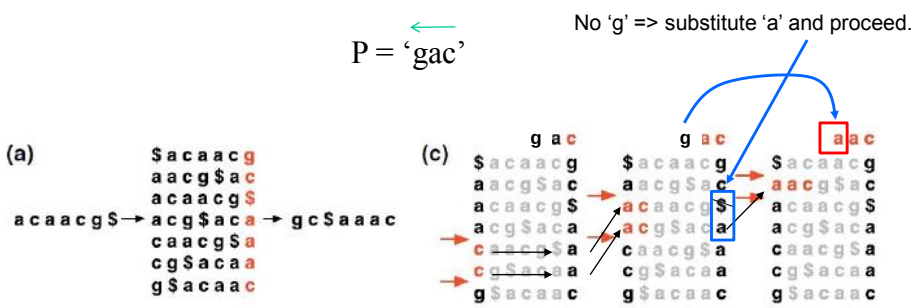
Inexact Matching

Sketch

- Each character in a read has a **numeric quality value**, with lower values indicating a higher likelihood of a sequencing error.
- Similar to EXACTMATCH, calculating matrix intervals for successively longer query suffixes.

- If the range becomes empty (a suffix does not occur in the text), then the algorithm selects an already-matched query position and substitute a different base there, introducing a mismatch into the alignment.
- The EXACTMATCH search resumes from just after the substituted position.

- The algorithm selects only those substitutions that
  - are consistent with the alignment policy and
  - yield a modified suffix that occurs at least once in the text.
  - If there are multiple candidate substitution positions, then the **algorithm greedily selects a position with a maximal quality value.**



Example of running INEXACTMATCH to find a query string in the text [1].

The Assembly Problem

How to assemble an unknown genome based on many highly overlapping short reads from it?

Problem Sequence assembly

INPUT: m l-long reads $S_1,\ldots, S_m$.

QUESTION: What is the sequence of the full genome?

The crucial difference between the problems of mapping and assembly is that now **we do not have a reference genome**, and we must assemble the full sequence directly from the reads.

De Bruijn Graphs

Definition

A k-dimensional de Bruijn graph of n symbols is a directed graph representing overlaps between sequences of symbols.

It has n^k vertices, consisting of all possible k-tuples of the given symbols. Note: the same symbol may appear multiple times in a tuple.

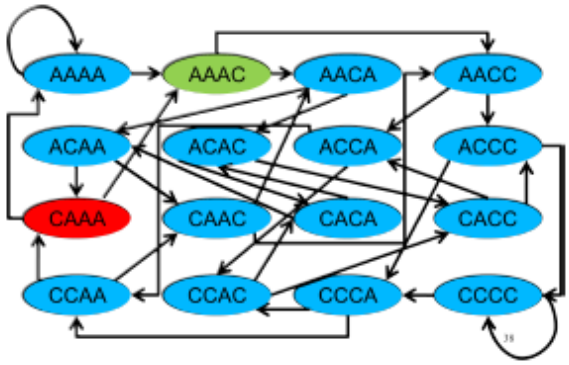
If we have the set of symbols $A = \{a_1,\ldots,a_n\}$ then the set of vertices is:

$V = \{ (a_1,\ldots, a_1, a_1), (a_1,\ldots, a_1, a_2),\ldots, (a_1,\ldots, a_1, a_n),$
$(a_1,\ldots, a_2, a_1), \ldots \quad \ldots, (a_n,\ldots, a_n, a_n)\}$

If a vertice w can be expressed by shifting all symbols of another vertex v by one place to the left and adding a new symbol at the end, then v has a directed edge to w.

Thus the set of directed edges E is:

$E = \{((v_1, v_2,\ldots, v_k), (w_1,w_2,\ldots, w_k))|\ v_2 = w_1, v_3 = w_2,\ldots, v_k = w_{k-1},$ and w_k new$\}$



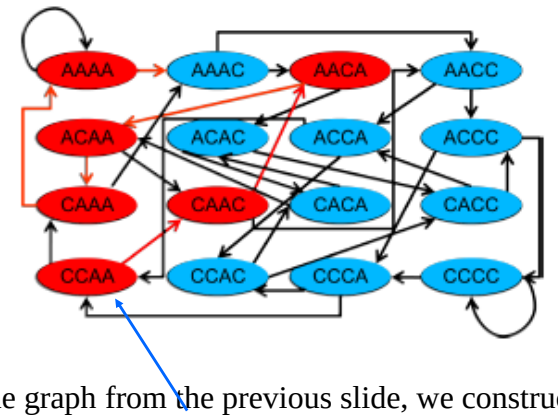
A portion of a 4-dimensional de Bruijn graph.

The vertex CAAA has a directed edge to vertex AAAC, because if we shift the label CAAA to the left and add the symbol C we get the label AAAC.

In this way the word CAAAC defines a directed edge

De Bruijn Graphs

- Given a read, every contiguous (k+1)-long word in it corresponds to an edge in the k-dimensional de Bruijn graph of the symbols {A,C,T,G}.
- Form a subgraph G of the full de Bruijn graph by introducing only the edges that correspond to (k+1)-long words in some read.
- A path in this graph defines a potential subsequence in the genome.
- Hence, we can convert a read to its corresponding path in the constructed subgraph G



Using the graph from the previous slide, we construct the path corresponding to CCAACAAAAC:

- shift the sequence one position to the left each time, and marking the vertex with the label of the 4 first nucleotides.

De Bruijn Graphs

- Form paths for all the reads
- Identify common vertices on different paths
- Merge different read-paths through these common vertices of the paths into one long sequence

Merging of Paths using Common Vertices

- The two reads in (a) are converted to paths in the graph
- The common vertex TGAG is identified
- Combine these two paths into (b).

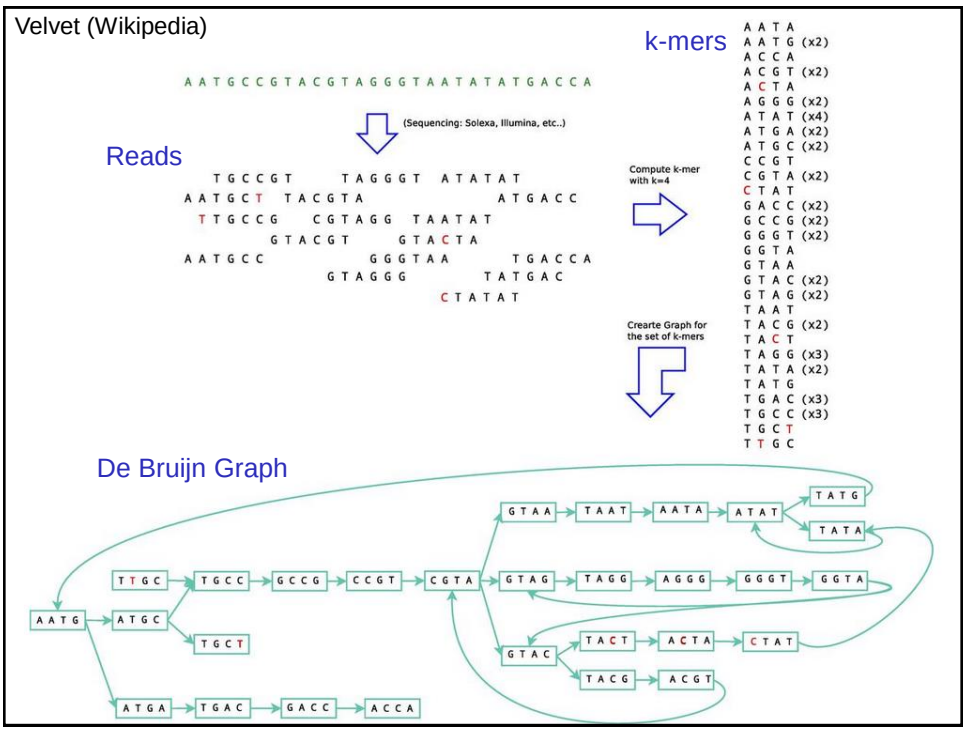
De Bruijn Graphs

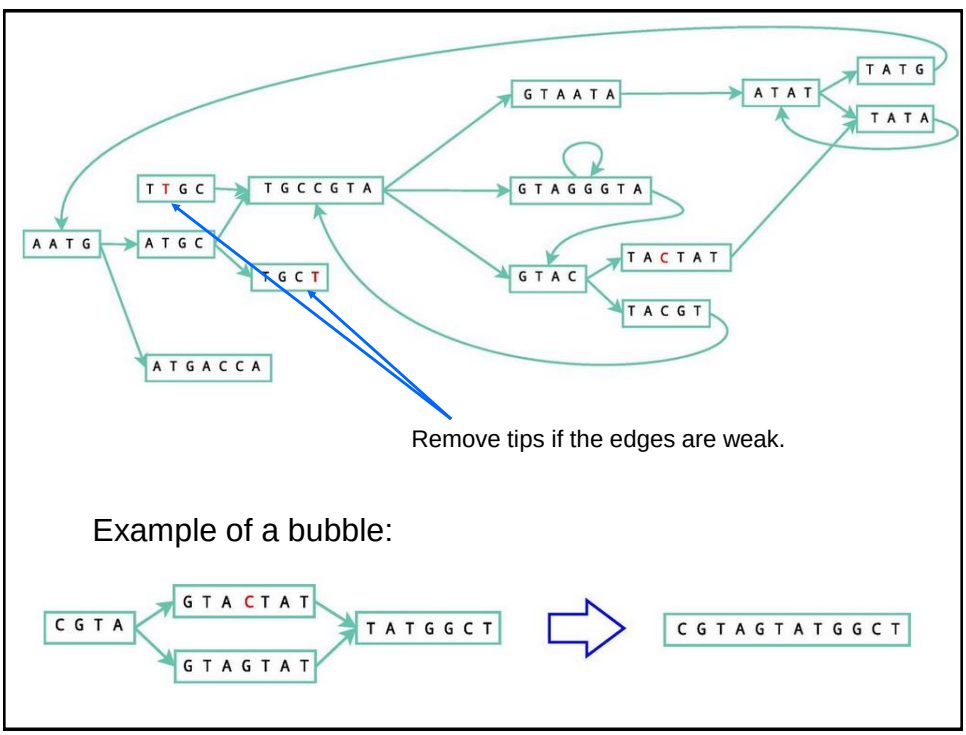
Velvet (2008), by Zerbino and Birney[6], Velvet GUI (2012) , is an algorithm which uses de Bruijn graphs to assemble reads; with the following difficulties:

- repeats will show up as cycles in the merged path that we form. We do not know how many times each cycle must be traversed in order to form the full sequence of the genome.
- If we have two cycles starting at the same vertex, we cannot tell which one to traverse first.
- Velvet attempts to address this issue by utilizing the extra information we have in the case of paired-ends sequencing (both ends of a DNA fragment are sequenced in Read 1 and Read 2, respectively. The distance between each paired read is known).

Read 1 Read 2

TGACCA …… distance d ….. CCTACC


Velvet (Wikipedia)
k-mers
A A T A
A A T G (x2)
A C C A
A C G T (x2)
A C T A
A G G G (x2)
A T A T (x4)
A T G A (x2)
A T G C (x2)
C C G T
C G T A (x2)
C T A T
G A C C (x2)
G C C G (x2)
G G G T (x2)
G G T A
G T A A
G T A C (x2)
G T A G (x2)
T A A T
T A C G (x2)
T A C T
T A G G (x3)
T A T A (x2)
T A T G
T G A C (x3)
T G C C (x3)
T G C T
T T G C
A A T G C C G T A C G T A G G G T A A T A T A T G A C C A
(Sequencing: Solexa, Illumina, etc..)
Reads
T G C C G T T A G G G T A T A T A T
A A T G C T T A C G T A A T G A C C
T T G C C G C G T A G G T A A T A T
G T A C G T G T A C T A
A A T G C C G G G T A A T G A C C A
G T A G G G T A T G A C
C T A T A T
Compute k-mer with k=4
Crearte Graph for the set of k-mers
De Bruijn Graph
A A T G
A T G C
T G C T
A T G A
T G A C
G A C C
A C C A
T T G C
T G C C
G C C G
C C G T
C G T A
G T A A
T A A T
A A T A
A T A T
T A T G
T A T A
G T A G
T A G G
A G G G
G G G T
G G T A
G T A C
T A C T
A C T A
C T A T
T A C G
A C G T


A A T G
A T G C
T T G C
T G C T
T G C C G T A
A T G A C C A
G T A A T A
G T A G G G T A
G T A C
T A C T A T
T A C G T
A T A T
T A T G
T A T A
Remove tips if the edges are weak.
Example of a bubble:
C G T A
G T A C T A T
G T A G T A T
T A T G G C T
C G T A G T A T G G C T

Other Assembly Algorithms

- HMM based
- Majority based
- Etc.

- Long reads: string graphs

Other Assembly Algorithms

K.R. Bradnan et al. Assemblathon 2: evaluating *de novo* methods of genome assembly in three vertebrate species
(http://gigascience.biomedcentral.com/articles/10.1186/2047-217X-2-10, 2013)

"Many current genome assemblers produced useful assemblies, containing a significant representation of their genes and overall genome structure.

However, the high degree of variability between the entries suggests that there is still much room for improvement in the field of genome assembly and that:

approaches which work well in assembling the genome of one species may not necessarily work well for another."

Other Assembly Algorithms

Salzberg SL, Phillippy AM, Zimin A, Puiu D, Magoc T, Koren S, Treangen TJ, Schatz MC, Delcher AL, Roberts M, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. Genome Res. 2012; 22(3):557–67.

Three conclusions:

1.Quality: data quality, rather than the assembler itself, has a dramatic effect on the quality of an assembled genome

2.Variability: the degree of contiguity of an assembly varies enormously among different assemblers and different genomes

3.Correctness: the correctness of an assembly also varies widely and is not well correlated with statistics on contiguity.

Other Assembly Algorithms

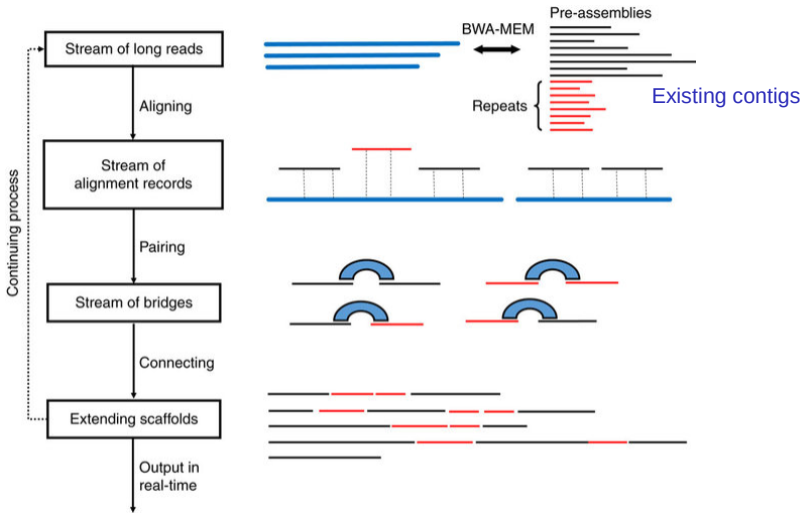
Scaffolding and completing genome assemblies in real-time with nanopore sequencing

By Minh Duc Cao, Son Hoang Nguyen, Devika Ganesamoorthy, Alysha G. Elliott, Matthew A. Cooper & Lachlan J. M. Coin

Nature Communications 8, Article number: 14515 (2017)

"Long read sequencing technologies, for example Pacific Biosciences' (PacBio) and Oxford Nanopore MinION sequencing, allow users to generate reads spanning most repetitive sequences, which can be used to close gaps in fragmented assemblies."

Figure 1: Workflow of the real-time algorithm.



Stream of long reads are aligned to the existing contigs to create alignment records. Bridges connecting contigs are formed, and are used for extending scaffolds. These steps are performed in a streaming fashion.

Image from reference [8] (2017).

De Novo Assembly of Viral Quasispecies using Overlap Graphs

Jasmijn A. Baaijens, A. Z. El Aabidine, E. Rivals, and A. Schönhuth, De Novo Assembly of Viral Quasispecies using Overlap Graphs. Genome Research, 27:835–848, 2017

Viruses HIV, Zika, and Ebola:

- Ensemble of genetically related but different mutant strains, viral quasispecies.
- Each strains, each characterized by its own haplotypic sequence: high mutation and recombination rates

Goal: a viral quasispecies assembly

- presenting all of the viral haplotypes, and
- their abundance rates.

Viral Quasispecies Assembly

Viruses HIV, Zika, and Ebola:

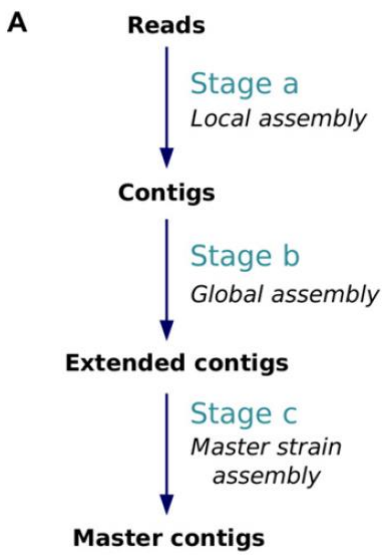
Goal: a viral quasispecies assembly

- presenting all of the viral haplotypes, and
- their abundance rates.

Problems:

- The number of different strains is usually unknown.
- Different strains can differ by only minor amounts of distinguishing mutations.
- Abundance rates can be as low as the sequencing error rates
- Reference genomes representing high-quality consensus genome sequences can be obsolete (as a result of great diversity and high mutation rates)

SAVAGE Overview



The three stages of SAVAGE. Each assembles sequences into longer sequences.

Output by each stage:

a) Contigs
b) maximally extended contigs
c) master contigs

Overlap Graphs

Given a collection ***R*** of sequences over the alphabet {A,C,G,T, N}
(N is unknown nucleotide),

The overlap graph G = (V,E) is a directed graph, where
- vertices v ∈ V correspond to reads/contigs R ∈ ***R*** and
- directed edges connect reads/contigs R_i, R_j ∈ ***R*** whenever
  - a suffix of R_i of sufficient length matches a prefix of R_j and
  - $QS(R_i, R_j) \geq \delta$ where $QS: V \times V \rightarrow R$ is a quality score

- For local and global assembly of contigs, the statistical model of Töpfer et al. (2014) is used: $QS(R_i, R_j) \geq \delta$ reflects the quality of the overlaps of reads R_i and R_j , i.e. with high probability locally identical haplotypic sequence.

- It includes a refined version of the (Phred-scaled (see next slide)) error profiles of the sequencing process.

- For the assembly of the master strain, $QS(R_i, R_j) \geq \delta$ indicates that two contigs share only a limited amount of mismatches in their overlaps => likely emerge from identical master strains sequences.

Phred Quality Score

Definition [edit]

Phred quality scores Q are defined as a property which is logarithmically related to the base-calling error probabilities P.[2]

$$Q = -10 \log_{10} P$$

or

$$P = 10^{\frac{-Q}{10}}$$

For example, if Phred assigns a quality score of 30 to a base, the chances that this base is called incorrectly are 1 in 1000.

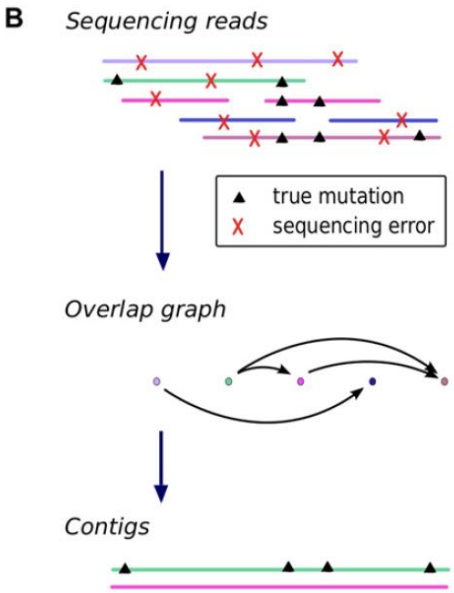
Phred quality scores are logarithmically linked to error probabilities

Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10,000	99.99%
50	1 in 100,000	99.999%
60	1 in 1,000,000	99.9999%

The phred quality score is the negative ratio of the error probability to the reference level of $P = 1$ expressed in Decibel (dB).

Source: Wikipedia.

Overlap Graphs

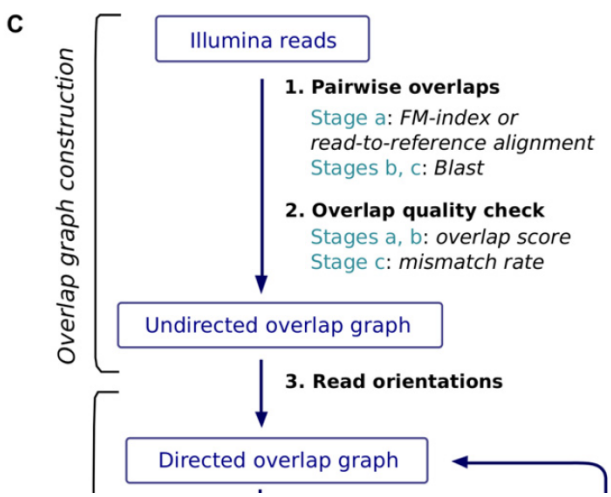


Principle of overlap graph construction

Distinction among the reads between:

- errors and
- shared mutations

Note: FM-index: Compressed self index as used in BWT.
FM = Full-text index in Minute space.

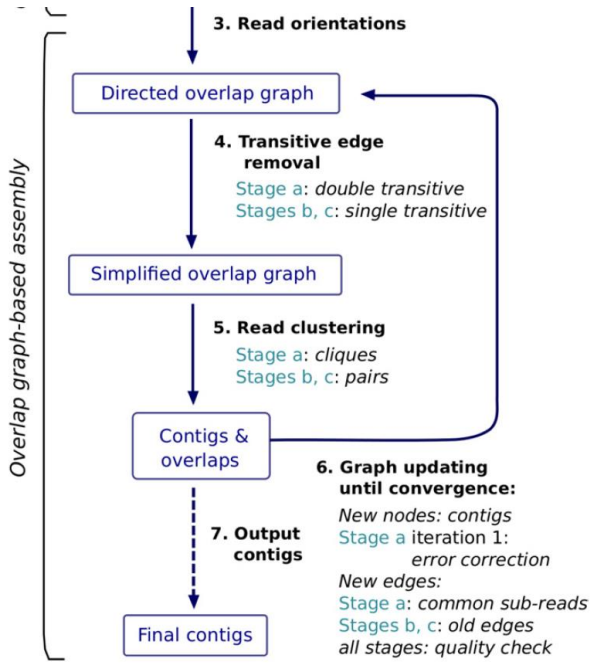


Each stage has two steps:

1. the overlap graph construction
2. assembly.

This panel summarizes the differences in each step between the three stages.

During overlap graph-based assembly, steps 4 to 6 are repeated iteratively until there are no edges left in the overlap graph.



Each stage has two steps:

1. the overlap graph construction
2. assembly.

This panel summarizes the differences in each step between the three stages.

During overlap graph-based assembly, steps 4 to 6 are repeated iteratively until there are no edges left in the overlap graph.

Life Sciences & Health CWI

SAVAGE

Disentangling the genetic diversity of virus infections

Benchmark Data Sets

Table 1. Characteristics of benchmarking data sets

Data set	Virus type	Genome length (bp)	Average coverage	Strain count	Strain abundance	Pairwise divergence
600× HIV mix	HIV-1	9478–9719	600×	5	20%	1%–6%
5-strain HIV mix	HIV-1	9478–9719	20,000×	5	5%–28%	1%–6%
10-strain HCV mix	HCV-1a	9273–9311	20,000×	10	5%–19%	6%–9%
3-strain ZIKV mix	ZIKV	10,251–10,269	20,000×	3	16%–60%	3%–10%
15-strain ZIKV mix	ZIKV	10,251–10,269	20,000×	15	2%–13%	1%–10%
Lab mix	HIV-1	9478–9719	20,000×	5	10%–30%	1%–6%

For each benchmark, we specify virus type, genome length, average coverage, strain count, relative abundance, and pairwise divergence. For the 600× HIV mix, the strains were homogeneously distributed with a relative abundance of 20% each.

Results

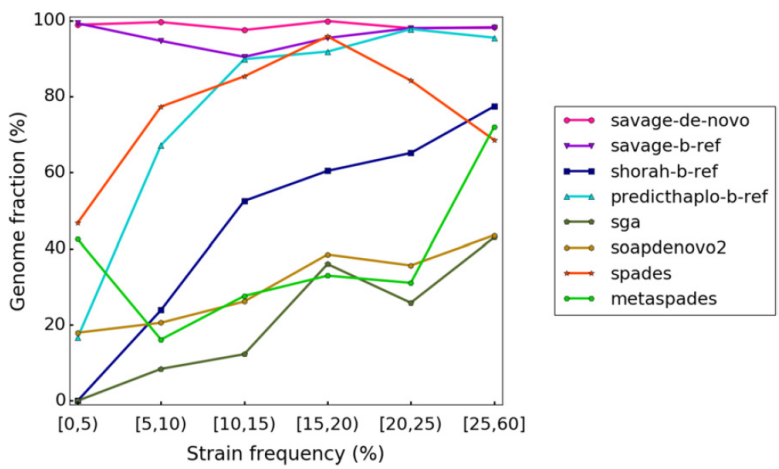


Figure 2. Target genome fraction recovered per strain for all 20,000× benchmarks, stratified by strain frequency.

Bibliography

[1] Ben Langmead, Cole Trapnell, Mihai Pop and Steven L. Salzberg. Ultrafast and memory-effcient alignment of short DNA sequences to the human genome. Genome Biology, 2009.

[2] Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[3] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using phred. II. Error probabilities. Genome Research, 1998.

[4] Paulo Ferragina and Giovani Manzini. Opportunistic data structures with applications. FOCS '00 Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 2000.

[5] Heng Li, Jue Ruan and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Research, 2008.

[6] Daniel R. Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Research, 2008.

[7] Ron Shamir, Computational Genomics Fall Semester, 2010 Lecture 12: Algorithms for Next Generation Sequencing Data, January 6, 2011 Scribe: Anat Gluzman and Eran Mick

[8] Minh Duc Cao, Son Hoang Nguyen, Devika Ganesamoorthy, Alysha G. Elliott, Matthew A. Cooper & Lachlan J. M. Coin. Scaffolding and completing genome assemblies in real-time with nanopore sequencing. Nature Communications 8, Article number: 14515, 2017.