sequence assembly

sequencing by primer walking
uses PCR products or mapped clones
good for known markers, bad for novel genomes
known location for sequence start => assembly known

shotgun sequencing
uses a library of overlapping fragments
‘randomly’ sheered or enzymatically cut
sequence position unknown
must locate matching overlapping fragments
(too) many programs
program and documentation quality varies greatly
source of data (homogenous versus heterogeneous)
type of data (read length, single end, paired end)
depth of coverage (== amount of RAM used)
handling of quality values (if not used in computation, pre–trimming and/or correction is necessary)
program choice depends on the ultimate goal
  short accurate versus longer less accurate contains
...assembly programs...

overlap (local alignment) and consensus
  e.g. Canu (a.k.a. Celera, wgs–assembler; ca. 1999); phrap (Green 1999), Edena (Hernandez et al. 2008), SOAP2/SOAP3 (Li et al. 2009), FALCON (ca. 2014)

prefix tree (partial hash)
  e.g. SSAKE (Warren et al. 2007), VCAKE (Jeck et al. 2007), JR-Assembler (Chu et al. 2013)

de Bruijn graphs
  e.g. Velvet (Zerbino and Birney 2008), SOAPdenovo (Li et al. 2010), Gossamer (Conway et al. 2012), SPAdes (Bankevich et al. 2012), ABYSS (Jackman et al. 2017)
...assembly programs

program choice depends on:

- objective
  - genome vs. RNAseq vs. ddRADseq etc.
- sequence type(s)
  - read length, paired and/or single end, error level, available quality values, etc.
- amount of data
- RAM efficiency
- quality of resulting output
overlap and consensus

e.g. phrap (Green 1999)
[0] remove end homopolymers
[1] find reads with matching words (user-defined size)
[2] local alignment of matching reads
[3] exclude vector (defined list) and chimeric reads
[4] compute matching score
[5] place in contigs (start with highest matching score)
[6] make consensus sequence using QV
prefix trees

e.g. JR-Assembler (Chu et al. 2013)
[0] filter out low complexity (e.g. Dust), low quality
[1] filter out duplicate reads (all orientations; save counts)
[2] select seeds (first quartile of read counts)
[3] extend reads with overlapping suffix/prefix (3' then 5')
[4] if extension is impossible, trim 1 base, retry until previous extension is reached
[5] break extension at repeat boundaries
We will describe the steps in detail in (Chu et al. 2013), for users to construct scaffolds. We will describe the stringent extension procedure to merge the assembled sequences. The three steps of seed selection, seed extension, and misextension de-tection are repeated until no unused seed remains. Finally, JR-Assembler uses back trimming to remove low-quality nucleotides at the end of a read to facilitate extension (Fig. 3). Fourth, any suitable seed exists? If an extension is terminated, JR-Assembler determines if a misassembly has occurred due to the existence of repeats. If it detects a misassembly, it identifies the boundaries of the repeat and breaks the sequence at each boundary. Steps (Chu et al. 2013) are repeated until no unused seed remains. (Fig. 1.)

We used the SRS data (10.3 million PE 101 bp reads, 450 × 450) of the genome, which is small, so we could efficiently scan a large parameter space to optimize each assembly. JR-Assembler had a higher ratio (89.1%) of no. NC_000913) using BLAST-like alignment tool (BLAT) comparisons of many assemblers using many datasets from different directions by jumping extension. (Fig. 4). The three assemblers all ran in a single thread (Velvet had not then pro-duced reads to the reference genome (retrieved from GenBank with accession no. NC_000913) using BLAST-like alignment tool (BLAT) in the calculation of genome coverage. The genome coverages by Taipan, Velvet, and ABySS conducted). The genome coverages by Taipan, Velvet, and ABySS were all under 30 min, whereas SOAPdenovo only required 7 min because it ran in multiple threads; the others required 11 min. JR-Assembler used only 5.1 gigabytes (GB) of memory, whereas the others required 22 GB. The run times of JR-Assembler, Velvet, and ABySS were all under 30 min, whereas SOAPdenovo required 11 min. JR-Assembler was the best in most assembly metrics, including the number of misassemblies, and the memory use. JR-Assembler performed better than most current SRS assemblers. JR-Assembler is freely available at http://jr-assembler.iis.sinica.edu.tw/.

Table 1 shows the assembly statistics by the six assemblers. JR-Assembler had a higher ratio (89.1%) of clean reads to the reference genome (retrieved from GenBank with accession no. NC_000913) using BLAST-like alignment tool (BLAT) in the calculation of genome coverage. The genome coverages by Taipan, Velvet, and ABySS were all under 30 min, whereas SOAPdenovo only required 7 min because it ran in multiple threads; the others required 11 min. JR-Assembler used only 5.1 gigabytes (GB) of memory, whereas the others required 22 GB. The run times of JR-Assembler, Velvet, and ABySS were all under 30 min, whereas SOAPdenovo required 11 min. JR-Assembler was the best in most assembly metrics, including the number of misassemblies, and the memory use. JR-Assembler performed better than most current SRS assemblers. JR-Assembler is freely available at http://jr-assembler.iis.sinica.edu.tw/.
de Bruijn graphs...

named for Nicolaas Govert de Bruijn (1946)
discovered independently Irving John Good (1946)
used by Camille Flye Sainte–Marie (1894)

$n$–dimensional graphs

nodes contain sequence, edges indicate overlap
de Bruijn graphs... often ‘compressed’ using alternative representations
block–sorting compression (Burrows–Wheeler transform)
Succinct de Bruijn graphs (Bowe et al. 2012)
Bloom filter (Bloom 1970)
...de Bruijn graphs...

e.g. ABySS 1.0 (Simpson et al. 2009)

[0] trim sequences, remove ambiguous reads

[1] make graph of $k$–mers

(usually slightly shorter than read length)

[2] create hash table of overlaps (i.e. the de Bruijn graph)

[3] simplify the graph

[a] delete dead ends

[b] use near matches if needed

[4] extract sequence by traversing the graph
block–sorting compression

^ACGTACGT$  $^ACGTACGT$
$^ACGTACGT$  $^ACGTACGT$
T$^ACGTACG  ACGT$^ACGT
GT$^ACGTAC  ACGTACGT$
CGT$^ACGTA  CGT$^ACGTA
ACGT$^ACGT  CGTACGT$^A
TACGT$^ACG  GT$^ACGTAC
GTACGT$^AC  GTACGT$^AC
CGTACGT$^A  T$^ACGTACG
ACGTACGT$^  TACGT$^ACG
block–sorting compression

count number of A, C, G, T, N
  if uniform read length, infer number of N
# block–sorting compression

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>$^</th>
<th>$^A</th>
<th>$^AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T$</td>
<td>T$^</td>
<td>T$^A</td>
<td>$^ACGTACGT</td>
</tr>
<tr>
<td>T</td>
<td>TA</td>
<td>TAC</td>
<td>TACG</td>
<td>ACGT$^ACGT</td>
</tr>
<tr>
<td>^</td>
<td>^A</td>
<td>^AC</td>
<td>^ACG</td>
<td>ACGTACGT$^</td>
</tr>
<tr>
<td>A</td>
<td>AC</td>
<td>ACG</td>
<td>ACGT</td>
<td>CGT$^ACGTA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>AC</th>
<th>ACG</th>
<th>ACGT</th>
<th>CGT$^ACGTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A -&gt; AC</td>
<td>ACG</td>
<td>ACGT</td>
<td>CGTACGT$^A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>CG</td>
<td>CGT</td>
<td>CGT$</td>
<td>GT$^ACGTAC</td>
</tr>
<tr>
<td>C</td>
<td>CG</td>
<td>CGT</td>
<td>CGTA</td>
<td>GTACGT$^AC</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT$</td>
<td>GT$^</td>
<td>T$^ACGTACG</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GTA</td>
<td>GTAC</td>
<td>TACGT$^ACG</td>
</tr>
</tbody>
</table>
block–sorting compression

first column == $T_z, A_v, C_w, G_x, N_y$
second column == $A_v, C_w, G_x, N_y, T_z$
calculate third etc. column
Bloom (1970) filter

an imprecise hash method of storing data

no false negatives, but false positives occur

useful when most queries will be negative

i.e. most reads do not have most $k$–mers

\{x, y, z\}
Bloom filters are space-efficient probabilistic data structures used to test whether an element is a member of a set.

They're surprisingly simple: take an array of $m$ bits, and for up to $n$ different elements, either test or set $k$ bits using positions chosen using hash functions. If all bits are set, the element probably already exists, with a false positive rate of $p$; if any of the bits are not set, the element certainly does not exist.

Bloom filters find a wide range of uses, including tracking which articles you've read, speeding up Bitcoin clients, detecting malicious websites, and improving the performance of caches.

This page will help you choose an optimal size for your filter, or explore how the different parameters interact.

- $n$: Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)
  - 4096
- $p$: Probability of false positives, fraction between 0 and 1 or a number indicating 1-in-$p$
  - 0.01
- $m$: Number of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)
- $k$: Number of hash functions

$n = 4,096$
$p = 0.010038487 \ (1 \text{ in 100})$
$m = 39,261 \ (4.79\text{KiB})$
$k = 7$

$n = \text{ceil}(m / (-k / \log(1 - \exp(\log(p) / k))))$
$p = \text{pow}(1 - \exp(-k / (m / n)), k)$
$m = \text{ceil}(n \times \log(p)) / \log(1 / \text{pow}(2, \log(2)))$
$k = \text{round}((m / n) \times \log(2))$
In light of our results, investigators may choose the most appropriate assembly tool(s) to use based on their specific read length. SOAPdenovo63mer and SOAPdenovo127mer that support kmer reads assembled on a desktop computer.

The table below provides recommendations for de novo tool selection under varying conditions:

<table>
<thead>
<tr>
<th>Read property</th>
<th>Small genome</th>
<th>Large genome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High N50</td>
<td>High SC</td>
</tr>
<tr>
<td>SE Low Short</td>
<td>Eu, SS</td>
<td>SS</td>
</tr>
<tr>
<td></td>
<td>SS, SO</td>
<td>SS</td>
</tr>
<tr>
<td></td>
<td>High Short</td>
<td>SO, Ed, AB, Ve</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>SO, SS, AB, Ve</td>
</tr>
<tr>
<td>PE Low Short</td>
<td>SO, SS, AB, Ve</td>
<td>AB, SS, Ve, SO</td>
</tr>
<tr>
<td></td>
<td>SO, SS</td>
<td>AB</td>
</tr>
<tr>
<td></td>
<td>High Short</td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>SO, AB, Ve</td>
</tr>
</tbody>
</table>

Requirements of assembly performance includes high N50, high sequence coverage (SC), low assembly error rate (AER). For different requirements, we recommend some de novo tools with order of priority according to properties of sequence reads, including single-end/paired-end, GC content, read length and sequence length. SE, single end reads; PE, paired end reads; Eu, Euler-sr; SS, SSAKE; Ed, Edena; AB, ABYSS; Ve, Velvet; SO, SOAPdenovo.

Table 7. Recommendations for de novo tool selection under varying conditions
De novo transcriptome assembly

**Figure 2**: Heat map showing for each data set (column) and each assembler (row) the calculated metric score (MS) (detailed definition in the Methods). The assembly tools are clustered based on their achieved MS over all data sets. The MS for 1 assembly tool and a single data set is based on 20 pre-selected metrics (see Table 4 and Methods for details) and is shown in 1 cell in the heat map (e.g., the MS for *E. coli* and *Trinity*\[10\] is 13.61). For each data set, an assembler’s MS is the sum of (0,1)-normalized scores of every single metric. The hierarchical clustering of the metric scores divides the assembly tools into 2 groups of generally high-ranked (upper half) and low-ranked (bottom half) tools. Except for Trans-ABySS\[9\], the MS reached for the largest human RNA-Seq data set is generally lower. Numbers in brackets next to the assembler names present the summarized metric scores (overall metric score, OMS) for all 9 data sets (see Methods). For the 3 similar human data sets infected with EBOV (Fig. 1), we added the mean MS value to the OMS. Details about the metric results for the human data set (no infection) can be found in Table 2 and for all other data sets Electronic Supplement Table S10.

**COs**, Trans-ABySS was among the best performing tools (Fig. 3). Trans-ABySS achieved one of the highest OMSs of 94.8 of all assembly tools (Fig. 2) and performed best for the large (human, mouse) data sets and the simulated data of human Chr1. By far, Trans-ABySS achieved the best MS (14.24) for the non-infected human data set. The lowest metric score was achieved for the bacterium data set (Fig. 2). Apart from the running time (Table 3), these results make Trans-ABySS one of the best-performing assembly tools in our comparison (besides Trinity and SPAdes).

**Bridger** In general, Bridger\[14\] assemblies resulted in high re-mapping rates between 87.35% (*E. coli*) and 96.72% (*C. albicans*; Fig. S4). For almost all TransRate metrics, the Bridger assemblies placed in the middle field of scores (Table S6). According to the RSEM-EVAL scores, Bridger generally performed well among the top tools (Table S9). Furthermore, Bridger performed well in the detection of complete BUSCOs with a moderate amount of duplicated hits. The amount of missing BUSCOs was comparably low (Fig. 3, Fig. S8). Based on a low duplication ratio and a low number of contigs, Bridger seems to produce very compact but also complete assemblies, especially for smaller data sets. The rate of mismatches per transcript was generally low (Table S10). Together, Bridger assemblies were of good quality and achieved among the top scores (OMS = 89.3).

**SOAPdenovo-Trans** The re-mapping rate of SOAPdenovo-Trans\[13\] was generally high (>85%), except for the *E. coli* data set (Fig. S4). SOAPdenovo-Trans performed quite well regarding most TransRate statistics and the calculated optimal score (Table S6). In most cases, only the Trinity and SPAdes assemblies could outperform SOAPdenovo-Trans on the TransRate metrics. The RSEM-EVAL scores varied depending on the assembled RNA-Seq (Holzer and Marz 2019).
trinity (Grabherr et al. 2011)

the ‘best’ transcriptome assembler
genome–guided de novo assembly
a perl script wrapper around many other programs
   jellyfish, inchworm, chrysalis, butterfly, TransDecoder, etc.
[0; inchworm] prefix tree assembly
[1; chrysalis] de Bruijn graph of [0]
   capture isoforms and paralogs
[2; butterfly] simplifies and extracts from [1]
   filters out ‘implausible’ isoforms
read mapping

placing reads onto reference sequences
to quantify expression
to find minor variants (re–sequencing)
assumes that reference sequences are (mostly) correct
a local alignment (with or without QV consideration)
  if QV is not used, pre–trimming and/or correction
should be used
many, many, programs
  speed and accuracy are highly variable
tuning is often required
STAR

[0] for each read
   search for maximal mappable prefix
      if match is interrupted (a ‘splice site’) => split read
[1] select seed reads (how?)
[2] join mapped reads to seeds (like a prefix tree)
The T3 libraries show a vast difference in recall, ranging from 12.5% (OLego) to 91.2% (CLC) for malaria data. Novoalign, GSNAP, CLC, STAR, and Subread have recall exceeding 50% on both organisms.

For results organized by class of misalignment (misaligned, intron gap, the shorter aligned segment is referred to as the anchor. A junction is where a read is spliced across an intron-sized gap that is typically thousands of bases long. When a read aligns across an intron gap, the anchor.

Figure 1 | Base-level precision and recall for human and malaria data sets.

Figure 2 | Recall (%)

Figure 3 | Junction-level precision and recall for human and malaria data sets.

(Baruzzo et al. 2016)
GenomeScope (Vurture et al. 2017)

genome size, heterozygosity, and repeat content

$k$–mer estimates (21–mers; counted via jellyfish)

>25x coverage required

<2% low error rate required (e.g. Illumina)

Statistical model using the $k$–mer histogram:

- excludes low frequency $k$–mers
- homozygous (Poisson) vs. heterozygous (bimodal)
  repeats are outside the ideal distribution

Genome size = normalized observed vs. mean coverage