Algorithms for Next-Generation Sequencing

Genome assembly
Sequence Assembly

- The sequencing machine can only produce sequence reads of limited length.
- Computational methods are needed to assemble the reads together to form a long DNA sequence.
Agenda

• Shot-gun sequencing
• Possible errors
  – Sequencing errors
  – Coverage is not uniform (Low coverage region)
  – Repeat
• How to evaluate the goodness of an assembly?
• Read correction
• Assembly approach
  – De-bruijn graph
  – Base-by-base extension
• Scaffolding
• Gap Filling
• Genome assembly using long reads
Shot-gun sequencing
Shot-gun sequencing

Genomes of the sample

- Sonication
- Size selection
- Sequencing
Mate pair sequencing

Size selection → Circularize, ligation, and cut → Sequencing
What is a paired-end read?

- A paired-end read is extracted from the sample genome
  - Each tag is of length *readlength*. (e.g. readlength = 100)
  - The span of the paired-end read is called its insert size (e.g. 300bp, 1000bp, 5000bp, 10000bp, or 20000bp)
Sequencing errors
Errors in DNA Sequencing

• Sequencing errors
  – Caused by ‘misreading’ bases by sequencing machine.

  - In most sequencing technologies, sequencing errors are more likely to occur towards end of the read.
Sequencing coverage

- We assume the reads have uniform coverage of the whole genome.

- However, some regions may have lower coverage while some regions may have higher coverage.
Size selection is not exact

- Sample fragment length distribution

![Graph 1: 300bp paired-end library](image1)

![Graph 2: 10,000bp mate pair library](image2)
Errors in DNA Sequencing

• Ligation errors
  – Occur in mate-pair libraries during library construction.
  – Two unrelated reads are paired together.
How to evaluate the goodness of an assembly?

- Number of contigs
- Number of contigs longer than 1000bp
- Maximum contig length
- N50
  - the contig length such that contigs of equal or longer than that length account for 50% of the combined total length of the assembly.

- If the assembly is nearly complete, number of contigs should be small while maximum contig length and N50 should be big.
Sequence Assembly Problem

- Given single-end (or paired-end) reads, we aim to assemble them to reconstruct the genome.
Steps

• Input: A set of single-end or paired reads

1. Read correction
2. Contig assembly (two approaches)
   – De-bruijn graph
   – Base-by-base extension
3. Scaffolding
4. Gap filling
Read correction
Read correction

• Input: A set of reads $R = \{R_1, ..., R_n\}$
• Aim: Correct the errors in the reads.

• E.g. $R = \{ R_1=\text{AAGTGAA}, R_2=\text{AGTGCAG}, R_3=\text{GTGAAGT}, R_4=\text{TGAAGTG} \}$
• $\rightarrow$
• $\hat{R} = \{ R_1=\text{AAGTGAA}, R_2=\text{AGTGAAG}, R_3=\text{GTGAAGT}, R_4=\text{TGAAGTG} \}$
k-mer frequency based error correction

• When a genome is sampled at high coverage, any k-mer in the genome will appear in multiple reads.
• For any k-mer $t$, denote $freq(t)$ be the number of reads and the reverse complements that contain $t$.
• If $freq(t)$ is small, it is likely that some error occurs.

$R_1=$ AAGTGAA
$R_2=$ AGTGCAG
$R_3=$ GTGAAGT
$R_4=$ TGAAGTG

$freq(GCAG)=1$. It is likely that it contains error.
Solid k-mer

- A k-mer t is said to be solid if freq(t) > M
- \( \mathcal{R} = \{ R_1 = \text{AAGTGAA}, R_2 = \text{AGTGCAG}, R_3 = \text{GTGAAGT}, R_4 = \text{TGAAGTG} \} \)
- \( \mathcal{S}_\mathcal{R} = \{ \text{AAGT}, ..., \text{TTCA} \} \)
- For \( M=1 \), \( \mathcal{T} = \{ \text{AAGT}, \text{ACTT}, \text{AGTG}, \text{CACT}, \text{CTTC}, \text{GAAG}, \text{GTGA}, \text{TCAC}, \text{TGAA}, \text{TGCA}, \text{TTCA} \} \).
- For \( M=2 \), \( \mathcal{T} = \{ \text{AAGT}, \text{ACTT}, \text{AGTC}, \text{CACT}, \text{TGAA}, \text{TTCA} \} \).

\[
\begin{array}{|c|}
\hline
4-mer & \text{freq}(t) \\
\hline
\text{AAGT} & 3 \\
\text{ACTT} & 3 \\
\text{AGTG} & 3 \\
\text{CACT} & 3 \\
\text{CTTC} & 2 \\
\text{CTGC} & 1 \\
\text{GAAG} & 2 \\
\text{GCAC} & 1 \\
\text{GCAG} & 1 \\
\text{GTGA} & 2 \\
\text{GTGC} & 1 \\
\text{TCAC} & 2 \\
\text{TGAA} & 3 \\
\text{TGCA} & 2 \\
\text{TTCA} & 3 \\
\hline
\end{array}
\]

\[
R_1 = \text{AAGTGAA} \\
R_2 = \text{AGTGCAG} \\
R_3 = \text{GTGAAGT} \\
R_4 = \text{TGAAGTG}
\]
Spectral alignment problem (SAP)

- SAP is a k-mer based read error correction method.
- Input: A set of reads $R$
- Let $T$ be the set of all correct k-mers in the sample genome (approximated by solid k-mers among all k-mers in $S_R$)
- A read $R$ is a $T$-string if every k-mer of $R$ is in $T$.
- Aim: Convert every read $R$ to $R'$ by the minimum number of mutations such that $R'$ is a $T$-string.
Example

- \( T = \{ \text{AAGT, ACTT, AGTG, CACT, CTTC, GAAG, GTGA, TCAC, TGAA, TGCA, TTCA} \} \).
- \( R = \{ R_1 = \text{AAGTGAA}, R_2 = \text{AGTGCAG}, R_3 = \text{GTGAAGT}, R_4 = \text{TGAAGTG} \} \).

- \( R_1, R_3 \) and \( R_4 \) are \( T \)-string. Hence, they are assumed to be correct.
- \( R_2 \) is not a \( T \)-string since \( \text{GTGC} \) and \( \text{GCAG} \) do not appear in \( T \). By mutating \( C \) (pos 5) \( \rightarrow \) A, \( R_2 \rightarrow \text{AGTGAAG} \), which is a \( T \)-string.
- Hence, by spectral alignment problem, we mutate \( C \rightarrow A \) at position 5 of \( R_2 \).

\[
\begin{align*}
R_1 &= \text{AAGTGAA} \\
R_2 &= \text{AGTGCAG} \\
R_3 &= \text{GTGAAGT} \\
R_4 &= \text{TGAAGTG}
\end{align*}
\]
Recursive formula

- For any read R, denote dist(i, t) be the minimum edit distance between R[1..i] and any T-string that ends at k-mer t.
- Assume there is no indel error in the first k bases of R.
- Let $\rho(x, y) = 0$ if $x = y$; and $\rho(x, y) = 1$ if $x \neq y$.
- Base case (i=k): dist(k, t) = Hamming(R[1..k], t) for $t \in T$ and dist(i, t) = $\infty$ for $t \not\in T$.

$\text{min}_{b \in \{A, C, G, T\}} \begin{cases} 
\text{dist}(i - 1, b \cdot t[1..k - 1]) + \rho(R[i], t[k]) & \text{match} \\
\text{dist}(i - 1, t) + 1 & \text{delete} \\
\text{dist}(i, b \cdot t[1..k - 1]) + 1 & \text{insert}
\end{cases}$

Our aim is to find $\min_{t \in T} \text{dist}(|R|, t)$. 

\[
\begin{array}{c}
\text{R} \\
\text{R'}  \\
\end{array}
\begin{array}{c}
\text{t}
\end{array}
\]
Solve this problem by DP?

- Naïvely, we may think the problem can be solved by DP.
- However, it is not possible since it has cycle.
- E.g. the right figure is the dependency graph for
  - \( T = \{AAGT, AGTG, GAAG, GTGA, TGAA, TGCA\} \).
  - \( R_2 = AGTGCAG \)
- It contains cycle.
Key Lemma

- Lemma: \( \text{dist}(i, t) = \) the length of the shortest path from \( v_s \) to \((i, t)\).
- Hence, our aim is to find the shortest path from \( v_s \) to \((|R|, t)\) for some \( t \in T \).
- Note that the dependency graph has \( O(|R| \cdot |T|) \) nodes and edges.

![Diagram of the dependency graph]

\( R_2[1..4] = \text{AGTG} \)

\( R[5] = \text{C} \)

\( R[6] = \text{A} \)

\( R[7] = \text{G} \)
Example

- $T = \{\text{AAGT, AGTG, GAAG, GTGA, TGAA, TGCA}\}$. $R_2 = \text{AGTGCAG}$.
- The minimum path length is 1.
- The corrected read is $\text{AGTG AAG}$.
- There are $|R|$ choices for $i$ and $|T|$ choices for $t$.
- The running time is $O(|R| \cdot |T|)$.
k-mer counting
k-mer counting

• Given a set \( Z \) of k-mers that appear in a set of reads \( R \),
  – We aim to count the number of occurrences of each k-mer.
• Assume \( |Z| = N \).

• This is the subroutine used by read error correction. It is also used in the assembly step.
k-mer counting solutions

• 1. Build a count table of size $4^k$.
• 2. Jellyfish
• 3. DSK
A count table of size $4^k$

- Build a table `Count[]` and initialize `Count[t]=0` for every k-mer `t`.
- For every k-mer `t` from the read,
  - `Count[t] = Count[t]+1;`
- Report every `t` and `count[t]` if `count[t]>0`.

- Running time: $O(4^k + N)$
- Space: $O(4^k)$

- When `k` is large, the space is too much.
Example

- AC
- CG
- AC
- GT
- CA
- GG
- AC
- GT

<table>
<thead>
<tr>
<th>i</th>
<th>2-mer</th>
<th>Count[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>AC</td>
<td>1+1+1=3</td>
</tr>
<tr>
<td>2</td>
<td>AG</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AT</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CA</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CG</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>CT</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>GA</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>GC</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>GG</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>GT</td>
<td>1+1=2</td>
</tr>
<tr>
<td>12</td>
<td>TA</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TC</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TG</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TT</td>
<td></td>
</tr>
</tbody>
</table>
Hash table

• Build a hash table \( H[1..\frac{N}{\alpha}] \) and a count table \( \text{count}[1..\frac{N}{\alpha}] \).
  – \( \alpha \) is the load factor (0<\( \alpha \)≤1)
  – \( \text{count}[i] \) stores the count of the k-mer \( H[i] \)
• We hash every k-mer into \( H[] \) using hash function \( h() \).
• Resolve collision by linear probing.

• When \( \alpha < 0.7 \), it is expected the number of collisions is low.
• The expected running time is \( O(N) \)
• The space is reduced to \( \frac{N}{\alpha}(2k + 32) \) bits.
  – \( H[] \) uses \( 2k\frac{N}{\alpha} \) bits and \( \text{count[]} \) uses \( 32\frac{N}{\alpha} \) bits.

• This approach is used by Jellyfish.
• Although it uses less space, the space depends on \( N \).
The Jellyfish algorithm

Algorithm \texttt{k-mer\_counting}(Z, \alpha, h)

\textbf{Require:} Z is a set of N's k-mers, \alpha is a load factor that controls the hash table size and \( h(.) \) is the hash function

\textbf{Ensure:} The count of every k-mer appearing in Z

1: Set \( H[1..\frac{N}{\alpha}] \) be a table where each entry requires 2k bits
2: Set \( \text{Count}[1..\frac{N}{\alpha}] \) be a table where each entry requires 32 bits
3: Initialize \( T \) to be an empty table
4: \textbf{for} each k-mer \( z \) in \( Z \) \textbf{do}
5: \hspace{1em} \( i = \text{hashEntry}(z, \frac{N}{\alpha}, h) \);
6: \hspace{1em} \text{if} \( H[i] \) is empty \text{ then}
7: \hspace{2em} \( H[i] = z \) and \( \text{Count}[i] = 1 \);
8: \hspace{1em} \text{else}
9: \hspace{2em} \( \text{Count}[i] = \text{Count}[i] + 1 \);
10: \hspace{1em} \text{end if}
11: \textbf{end for}
12: Output \( (H[i], \text{Count}[i]) \) for all non-empty entries \( H[i] \);

Algorithm \texttt{hashEntry}(z, h, size)

1: \( i = h(z) \mod \text{size} \);
2: \textbf{while} \( H[i] \neq z \) \textbf{do}
3: \hspace{1em} \( i = i + 1 \mod \text{size} \); /* linear probing */
4: \textbf{end while}
5: Return \( i \);
Example

- Table size is 7
- Let $h(z) = z \mod 7$

<table>
<thead>
<tr>
<th>$i$</th>
<th>H[$i$]</th>
<th>Count[$i$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>AC</td>
<td>1+1+1=3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>GG</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>GT</td>
<td>1+1=2</td>
</tr>
<tr>
<td>5</td>
<td>CA</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>CG</td>
<td>1</td>
</tr>
</tbody>
</table>

One collision: CA
If $N$ is large, the previous approach still use a lot of space.

Assume the memory limit is $M$ bits and the disk limit is $D$ bits.

We partition the $N$ k-mers into $n_{list} \left( = \frac{2kN}{D} \right)$ lists, each list contains $\frac{D}{2k}$ k-mers.

- Hence, we can store one list in the disk.

For each list with $\frac{D}{2k}$ k-mers, it is partitioned into $n_{sublist} \left( = \frac{D(2k+32)}{0.7(2k)M} \right)$ sublists, each sublist contains $\frac{0.7M}{(2k+32)}$ k-mers.

- We can count the k-mers in each sublist in memory using Jellyfish.
Solution

• For $i = 0$ to $n_{\text{list}} - 1$ do
  – Phase 1
    • Scans all $N$ k-mers and obtain the $i$th list.
    • Partition the elements in the $i$th list into sublists and write them into disk.
  – Phase 2
    • For $j = 0$ to $n_{\text{sublist}} - 1$ do
      – Use Jellyfish to count the k-mers and output their counts.
DSK algorithm

Algorithm DSK($Z, M, D, h$)

**Require:** $Z$ is a set of $N$'s $k$-mers, target memory usage $M$ (bits), target disk space $D$ (bits) and hash function $h(.)$

**Ensure:** The count of every $k$-mer appearing in $Z$

1: $n_{list} = \frac{2kN}{D}$
2: $n_{sublist} = \frac{D(2k+32)}{0.7(2k)M}$
3: for $i = 0$ to $n_{list} - 1$
4: Initialize a set of empty sublists \{\(d_0, \ldots, d_{n_{sublist}} - 1\)\} in disk;
5: for each $k$-mer $z$ in $Z$
6: if $h(z) \mod n_{list} = i$ then
7: $j = (h(z)/n_{list}) \mod n_{sublist}$;
8: Write $z$ to disk in the sublist $d_j$;
9: end if
10: end for
11: for $j = 0$ to $n_{sublist} - 1$
12: Load the $j$th sublist $d_j$ in memory;
13: Run $k$-mer\_counting($d_j, 0.7, h$) (see Figure 5.9) to output the number of occurrences of every $k$-mer in the sublist $d_j$;
14: end for
15: end for
Example

- \( n_{\text{list}} = 2; \) \( n_{\text{sublist}} = 2; \)
- Let \( h(z) = z \)

<table>
<thead>
<tr>
<th></th>
<th>List index</th>
<th>Sublist index</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC (1)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CA (4)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CG (6)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>GG (10)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>GT (11)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

List 0
- AC
- CG
- CA
- GG

List 1
- GT
- AC
- AC

Sublist 0
- CA
- CG
- GG

Sublist 1
- GT
- AC
- GT
Estimating genome size from k-mer frequency

- Let $C$ be the coverage
- Let $L$ be the mean read length.
- Let $\mu$ be the mean k-mer frequency (the peak in the k-mer distribution)
- We have
  - $C = \mu \cdot \frac{(L - k + 1)}{L}$

- Let $n$ be the genome size
- Let $N$ be the number of reads sequenced.
- Then, we have $n = N \cdot L / C$. 
Construct contigs

- After the reads are corrected, this step aims to stitch the reads to form long contigs.
- Two approaches:
  - Base-by-base extension approach
  - De Bruijn graph approach
Base-by-base extension approach

- E.g. SSAKE, VCAKE and SHARCGS
- Given a template, extend it on both 5' and 3' ends.

ACTACGTAGACCACACGTAGC
CACGTAGCTC
ACGTAGCTCT
ACGTAGCTCT
CGTAGCCCTG
CGTAGCTCTG
GTAGCCCTGA
TAGCTCTGAT

Feasible extension

Contig

Reads overlap with contig
Simple base-by-base extension Algorithm

Algorithm SimpleAssembler($\mathcal{R}$)

Require: $\mathcal{R}$ is a set of reads

Ensure: A set of contigs

1: while some read $R$ are not used for reconstructing contigs do
2: Set the template $T = R$;
3: repeat
4: Identify a set of reads that align to the 3’ end (or 5’ end) of the template $T$;
5: Identify all feasible extensions of the template;
6: If there is a consensus base $b$, set $T = Tb$ if the extension is at 3’ end and set $T = bT$ if the extension is at 5’ end;
7: Mark all reads with the consensus base $b$ as used;
8: until there is no consensus base;
9: Report the template as a contig;
10: end while
Utilizing paired-end info

- Standard algorithm can only give short contigs.
- By paired-end information, we can get longer contig.
De Bruijn graph approach

- Velvet (Zerbino and Birney 2008), Velvet-SC (Chitsaz et al 2011), Euler-SR (Chaisson and Pevzner 2008), Abyss (Simpson et al 2009), IDBA (Peng et al 2010), SOAPdenovo (Li et al 2010), ALLPATH (Bulter et al 2008), Edena (Hernandez et al 2008)
3 ideas in de Bruijn graph approach

• 1. De Bruijn graph with no error.
• 2. Resolve errors by
  – remove tips with length < 2k
  – Merge bubbles
• 3. Iterative try different k to resolve gaps and branches
What is a De Bruijn graph?

• Consider $S = \{ACG, CATC, CGC, GCA\}$.

• Transformation: Given the input $S$, creating a De Bruijn graph $H_k$ with
  – Each vertex represents a $k$-prefix or $k$-suffix of some length-$(k+1)$ substring in $S$
  – For each length-$(k+1)$ substring in $S$, creating an edge connecting the vertices representing the $k$-prefix and $k$-suffix.

• Report result: Read Euler paths from $H_k$
  – Generate the Euler path of $H_k$, we reconstruct the sequence ACGCATC!
De Bruijn Assembler (no sequencing error)

Input: A set of reads $S$
1. Fixed a $k$.
2. Generate the de Bruijn graph $H_k$ for $S$.
3. From $H_k$, we extract contigs.
   - A contig is a path where every node (except start and end nodes) has in-degree=1 and out-degree=1.
Example (k=3)

- Input:
  - AAGATC, GATCGAT, CGATGA, ATGATT, GATTT
- Construct de Bruijn graph for k=3

The constructed contigs:
- AAGAT, GATCGAT, GATGAT, GATTT

When k is small, the short repeats generate a lot of branches. The above example has two possible solutions:
- AAGATCGATGATT
- AAGATGATCGATTT

It is not possible to generate one single contig from this de Bruijn graph.
Example (k=4)

- Input:
  - AAGATC, GATCGAT, CGATGA, ATGATT, GATTT

- Construct de bruijn graph for k=4

```
AAGA → AGAT → GATC → ATCG → TCGA → CGAT
     ↓     |     |     |     |
      ATTT GATT TGAT ATGA GATG
```

- The constructed contigs:
  - AAGATCGATGATTT

For this case, we get one single contig from this de bruijn graph.
Example (k=5)

- Input:
  - AAGATC, GATCGAT, CGATGA, ATGATT, GATTT

- Construct de bruijn graph for k=5

```
AAGAT -> AGATC
GATCG -> ATCGA -> TCGAT
CGATG

GATTT -> TGATT -> ATGAT
GATGA
```

- The constructed contigs:
  - AAGATC, GATCGAT, CGATGA, ATGATT, GATTT

When k is large, some (k+1)-mers are missed due to uneven coverage. In the above example, we miss the following 6-mers: AGATCG, TCGATC, GATGAT, TGATTT
Difficult to choose k

- Based on the previous examples, we observe that de Bruijn graph is good when we know the correct k.

- When k is large, some k-mers are missing (especially for regions with low coverage).
- When k is small, there are many branches due to repeat regions. This results in many short contigs.

- We need to identify k to balance these two issues.
- We will discuss more on k selection.
Sequencing errors

- De Bruijn graph is prone to sequencing errors.
- One sequencing error can generate $k$ false $k$-mers.
- Sequencing errors make the de Bruijn graph huge and introduce many incorrect contigs.

- Velvet suggested two ways to reduce the errors.
  - Remove tips
  - Merge bubbles
**Tips**

- Consider multiple reads where one of them has sequencing error:
  - GACTCCGANN
  - NGACTCCGAN
  - NNGACTTCGA
  - NNNGACTCCG
  - NNNNGACTCC

- It generates a tip!
Remove Tips

• Tip is a path of length at most $k$, disconnected on one of its end (i.e. one end with 0 in-degree or out-degree) and whose vertices have low multiplicity.
• It represents a potential contig of length at most $k$. They are likely to be false positives.

• Note that removing a tip may generate more tips. The procedure need to remove tips recursively.
Bubbles

• Bubbles are two paths starting from the same vertex and ending at the same vertex, where these two paths represent different contigs differ by only one nucleotide.

• E.g.
  – NGACTCCGAGNNNNN
  – NNGACTCCGAGNNNN
  – NNNGACTTCGAGNNNN
  – NNNNGACTCCGAGNN
  – NNNNNNGACTCCGAGN

\[ \text{GACT} \rightarrow \text{ACTC} \rightarrow \text{CTCC} \rightarrow \text{TCCG} \rightarrow \text{CCGA} \rightarrow \text{CGAG} \rightarrow \ldots \]
Merge bubbles

- The top path represents GACTCCGAG.
- The bottom path represents GACTTCGAG.
- There is only one nucleotide different. We merge them.
TourBus algorithm

- TourBus searches through the graph for parallel paths that have the same starting and end node.
- If their sequences are similar enough, the path with lower coverage is merged into the path with higher coverage.
Tour Bus Algorithm

Algorithm Tour_Bus(H, s)
Require: H is the de Bruijn graph and s is an arbitrary node in H
Ensure: A graph formed after merging the bubbles
1: Set Q be a queue with one node s;
2: while Q ≠ ∅ do
3:   u = dequeue(Q);
4:   for each child v of u do
5:       if visited[v] = false then
6:         Set \( \pi(v) = u \); /* set u as v’s parent in the BFS tree */
7:         Set visited[v] = true;
8:         enqueue(Q, v);
9:       else
10:          Find the lowest common ancestor c of u and v by \( \pi() \);
11:          if the paths \( c \rightarrow u \) and \( c \rightarrow v \) are similar enough then
12:             Merge the two paths and keep the path with the highest path weight;
13:          end if
14:       end if
15:   end for
16: end while
Example

(a)

C5 → C5 → T3 → G2 → A2 → C2 → T2 → A5 → G2 → T2 → T3 → T5

(b)

r u' v c

C5 → C5 → T3 → G2 → A2 → C2 → T2 → A5 → G2 → T2 → T3 → T5

(c)

C' A5 → C5 → T5 → A5 → G2 → T2 → T3 → T5

(d)

C5 → C5 → T5 → T9 → A5 → C6 → T6 → A6 → G3 → T3 → T5
Trust k-mers with high multiplicity

• Assumption:
  – There are multiple reads cover any locus.
  – Sequencing error is random

• We only keep k-mer have multiplicity at least m (say 2).
• This approach can remove many noise k-mers.
Algorithm (Velvet)

- Input: a set \( S \) of reads and the parameter \( k \)
- Build a de Bruijn graph \( H_k \) using \( k \)-mers from \( S \)
- Remove Tips
- Merge bubbles
- Remove nodes with multiplicity < \( th \)
- Report all contigs in the final \( H_k \).
How to select k?

• IDBA observed that
  – When k is small, we can get high quality contigs. However, the contigs are short since there are many branches due to repeat regions.
  – When k is large, we can get long contigs. However, there are a lot of errors in the contigs.
• Instead of fixed a k, IDBA suggested to incrementally increase k.
  – When k is small, we can get high quality contigs, though they are short.
  – Using those high quality contigs, we can correct the errors in reads.
  – Then, we can incrementally try longer k.
Algorithm IDBA

\textbf{Algorithm IDBA}(\mathcal{R}, k_{min}, k_{max})

\textbf{Require:} \(\mathcal{R}\) is a set of reads and \(k_{min}\) and \(k_{max}\) are de Bruijn graph parameter

\textbf{Ensure:} A set of contigs

1: \textbf{for} \(k = k_{min}\) \textbf{to} \(k_{max}\) \textbf{do}
2: \hspace{1em} Generate the de Bruijn graph \(H_k\) for \(\mathcal{R}\);
3: \hspace{1em} Remove tips;
4: \hspace{1em} Merge bubbles;
5: \hspace{1em} Remove nodes with multiplicity \(\leq m\);
6: \hspace{1em} Extract all maximal simple paths in \(H_k\) as contigs;
7: \hspace{1em} All reads in \(\mathcal{R}\) are aligned to the computed contigs;
8: \hspace{1em} The mismatch in the read is corrected if 80\% of reads aligned to the same position has the correct base;
9: \hspace{1em} \textbf{end for}
10: \hspace{1em} Extract all maximal simple paths in \(H_{k_{max}}\) as contigs;
Limitation of de Bruijn graph approach

- De Bruijn graph is big. It requires big memory.

- It cannot use the connectivity of paired-end reads before scaffolding.
  - SPAdes used paired de Bruijn graph to capture such information

- If there are long repeats, this approach may fail to get long contigs.
Scaffolding
Scaffolding

- We obtain a set of contigs.
- The next question is to find the ordering and orientation of the contigs in the genome.
  - An arrangement of contigs is called a scaffold.
- We can verify the correctness of a scaffold by paired reads whose alignments span more than one contig.
- E.g. (1) is concordant which support the scaffold
- (2), (3) and (4) are discordant and cannot support the scaffold.
Scaffolding problem

• Input:
  – A set of contigs of some genome
  – A set of paired reads that link the contigs
• Scaffolding finds the ordering and the orientation of the contigs so that the number of discordant paired reads is minimized.
• In the following example, for (C, -B, A), only one paired read is discordant.
• Scaffolding problem is NP-complete.

• A number heuristics and approximation algorithms exist.
  – PE-assembler, SSAKE, Abyss and SOAPdenovo
  – Bambus, SSPACE, SOPRA, MIP Scaffolder, Opera, SCARPA.

• Below, we discuss a heuristics method.
Steps in scaffolding

• 1. Demarcates all repeat regions within assembled contigs.

• 2. Build the contig graph

• 3. Identify a linear order of the contigs
Demarcating all repeat regions within assembled contigs

- Map all paired-end reads onto the contigs.

- The median read density is assumed to be the expected read coverage across the genome.

- Any region with read density higher than 1.5 times of the median is considered as a repeat region.
Build contig graph

• Filter all paired reads whose both ends aligned to the same contig.

Filter paired reads align on repeat regions
Identify a linear order of the contigs

Case 1: 1 discordant edge

Case 2: 2 discordant edges
Greedy scaffolding (in PEassembler)

- Denote $S$ be a scaffold contains any random contig
- Repeat
  - Find all contigs $C_1, C_2, ..., C_r$ that are feasible right (or left) neighbors of $S$.
  - Identify the permutation $C_{i_1}, C_{i_2}, ..., C_{i_r}$ that have the highest score.
  - Set $S = S \cdot C_{i_1}$.
- Until either (1) cannot further extend or (2) the score is negative.

- Note: Score of $S$ is the number of concordant edges – the number of discordant edges. For the following example, score is $2 - 1 = 1$. 

![Diagram of scaffolding process]
Greedy Scaffolding

**Algorithm Greedy_Scaffolding**

**Require:** A contig graph \((\mathcal{C}, \mathcal{E})\) where \(\mathcal{C}\) is the set of contigs and \(\mathcal{E}\) is the set of paired reads

**Ensure:** A set of scaffolds that cover all contigs in \(\mathcal{C}\)

1. **while** \(\mathcal{C}\) is not empty **do**
2. Denote \(S\) be a scaffold consists of a random contig \(C\) in \(\mathcal{C}\) and remove \(C\) from \(\mathcal{C}\);
3. **repeat**
4. From \(\mathcal{E}\), we identify all contigs \(C_1, C_2, \ldots, C_r\) that are right feasible neighbors of \(S\) and all contigs \(C'_1, \ldots, C'_{r'}\) that are the left feasible neighbors of \(S\);
5. Identify the permutation \(C_{i_1}, \ldots, C_{i_r}\) that maximizes \(s_R = \text{score}(S \circ C_{i_1} \circ \ldots \circ C_{i_r})\);
6. Identify the permutation \(C'_{j_1}, \ldots, C'_{j_{r'}}\) that maximizes \(s_L = \text{score}(C'_{j_1} \circ \ldots \circ C'_{j_{r'}} \circ S)\);
7. **if** \(s_R > s_L > 0\) **then**
8. Set \(S = S \circ C_{i_1}\) and remove \(C_{i_1}\) from \(\mathcal{C}\);
9. **else if** \(s_L > s_R > 0\) **then**
10. Set \(S = C_{j_{r'}} \circ S\) and remove \(C_{j_{r'}}\) from \(\mathcal{C}\);
11. **end if**
12. **until** both ends of \(S\) cannot extend
13. Report the scaffold \(S\);
Gap filling

• From scaffolding, we identify adjacent contigs.
• Those gaps are usually generated by repeat regions.
• Since we have paired-end reads from both 5’ and 3’, we may be able to fill-in the gap.
Long reads
Assembler for first generation sequencing

- Sanger sequencing generates long reads with little sequencing errors.

- Two approaches for assembly:
  - De-bruijn graph assembler (discussed previously)
    - E.g. Euler
  - OLC assembler (Overlap, Layout, and Consensus)
    - E.g. Celera
Shortest Common Superstring (SCS)

• Problem: Given a set of fragments $F$, find a shortest string $S$ which is a superstring of all fragments $F$.

• For example

\begin{align*}
\text{ACGTCA} \\
\text{GTCACCATG} \\
\text{CATGCATTC} \\
\text{ACGTCACCATGCATTC}
\end{align*}
Shortest Common Superstring (II)

- The SCS problem is NP-hard.
- The best known approximation algorithm has approximation ratio $= 2.5$
- The superstring problem also cannot handle sequencing errors.
- Below, we discuss a greedy solution Celera. Celera is based on overlap/layout/consensus approach.
OLC assembler
(Overlap, Layout, and Consensus)

• Get many random reads with redundancy
• Detect overlaps
• Layout
• Consensus
Detect overlaps

• All-to-all alignment
  – Overlap all read pairs (R1, R2) such that the mismatch rate is low (say at most 6%) and the overlap region is long enough (say at least 100bp).
  – For each pair, we can align them.

• Examples (assume match=1, mismatch/gap=-1):

  ACTCGAGTAAC
  TCGA-TAACGT
  Score = 7

  ACTCGAGTAAC
  TGAACTCGA-TAACGT
  Score = 9
Create the overlap graph

- A graph is formed where each vertex is a read and each edge connects a pair of overlapping reads

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_1</td>
<td>R_3</td>
<td>R_6</td>
<td>R_7</td>
<td>R_9</td>
<td>R_11</td>
</tr>
<tr>
<td>R_2</td>
<td>R_4</td>
<td>R_6</td>
<td>R_8</td>
<td>R_{10}</td>
<td>R_{12}</td>
</tr>
<tr>
<td>R_8</td>
<td>R_9</td>
<td>R_{10}</td>
<td>R_{12}</td>
<td>R_{14}</td>
<td>R_{16}</td>
</tr>
<tr>
<td>R_{20}</td>
<td>R_{21}</td>
<td>R_{22}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```plaintext
A: R_1 → R_2 → R_3 → R_4
D: R_8 → R_9 → R_{10} → R_{11} → R_{12} → R_{13} → R_{14} → R_{15} → R_{16}
C: R_5 → R_6 → R_7
B: R_5 → R_6 → R_7
E: R_20 → R_21 → R_22
```
Remove redundant edges from the overlap graph

- If there is a path $R_4 \rightarrow R_5 \rightarrow R_{17}$ and an edge $R_4 \rightarrow R_{17}$, then $R_4 \rightarrow R_{17}$ is redundant.
- Each linear path is a layout.
Consensus

• A layout consists of a set of reads. We aim to build the MSA of the reads and obtain the consensus sequence.
• Example: $R_1=$GGTAATCGTC, $R_2=$AATCGACTTA, $R_3=$CTGACTTTACC

Choose $x$ that maximizes $|\{a_i : a_i=x\}|$
Consensus

• However, computing MSA is NP-hard. We use greedy merging:
  – The string pair with the highest alignment score are merged first.
  – Then, the next highest scoring pair is selected and merged, and so on.

• Example: \( R_1=GGTAATCGTC \), \( R_2=AATCGACTTA \), \( R_3=CTGACTTACC \)

\[
\begin{align*}
R_1 &= GGTAATCGTC \\
R_2 &= AATC-GACTTA \\
R_3 &= CTGACTTACC
\end{align*}
\]

\[
\begin{align*}
\text{Score} &= 7 \\
\text{Score} &= 5 \\
\text{Score} &= 2
\end{align*}
\]

\[
\begin{align*}
R_2 &= AATC-GACTTA \\
R_3 &= CTGACTTACC \\
R_1 &= GGTAATC-GTC
\end{align*}
\]

\[
\begin{align*}
R_1 &= GGTAATC-GTC \\
R_2 &= AATC-GACTTA \\
R_3 &= CTGACTTACC
\end{align*}
\]

;merge the highest scoring pair

;merge the next highest scoring pair

Layout takes \( O(\Sigma_i n_i) \) time.
Consensus

• After getting the MSA, we need to deduce the consensus. This is called the contig.

• The simplest approach is to report the most frequent base in each column. For example,

\[
\begin{align*}
\text{GGTAATC-GTC} \\
\text{AATC-GACTTA} \\
\text{CTGACTTACC} \\
\end{align*}
\]

Choose \( x \) that maximizes \(|\{a_i : a_i = x\}|\)
Greedy merging may not report the optimal alignment

- Example: $R_1$ = TAAAGCG, $R_2$ = TAAAGTGG, $R_3$ = AGTCGG, $R_4$ = TCGG, $R_5$ = AGCG

<table>
<thead>
<tr>
<th></th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
</tr>
</thead>
</table>
| $R_1$ | $R_1$ = TAAAGCG  
     | $R_2$ = TAAAGTGG  
          | Score = 5  
     | $R_3$ = AGTCGG  
          | Score = 3  
     | $R_4$ = TAAAGCG  
          | Score = 1  
     | $R_5$ = AGCG  
          | Score = 4  
     | $R_2$ = TAAAGTGG  
          | Score = 4  
     | $R_3$ = AGTCGG  
          | Score = 2  
     | $R_4$ = TAAAGTGG  
          | Score = 2  
     | $R_5$ = AGCG  
          | Score = 3  
     | $R_3$ = AGTCGG  
          | Score = 4  
     | $R_4$ = TCGG  
          | Score = 3  
     | $R_5$ = AGCG  
          | Score = 1  
     | $R_4$ = TCGG  
          | Score = 4  
     | $R_5$ = AGCG  
          | Score = 1  
     | $R_4$ = TCGG  
          | Score = 4  
     | $R_5$ = AGCG  
          | Score = 1  

Greedy solution:

- $R_1$ = TAAAGC-G
- $R_2$ = TAAAGT-GG
- $R_3$ = AGTCGG
- $R_4$ = TCGG
- $R_5$ = AGC-G

Optimal solution:

- $R_1$ = TAAAGC-G
- $R_2$ = TAAAGT-GG
- $R_3$ = AGTCGG
- $R_4$ = TCGG
- $R_5$ = AGCG
Realign

• Anson and Myers proposed a round-robin realignment algorithm (a heuristic algorithm) to improve the MSA by local realignment.
• The algorithm is as follows.
• Input: A MSA of $R_1, R_2, ..., R_k$
• Output: A MSA with higher sum of pairs of alignment score
• Repeat
  – For i = 1 to k
    • Q = {1, ..., k} – {i}
    • Update X to be an alignment between $R_i$ and $X_Q$ that improve the alignment score;
• until the alignment score does not decrease;
An example

- Below is an example

R1 = TAAAGC-G
R2 = TAAAGT-GG
R3 = AGTCGG
R4 = TCGG
R5 = AGC-G

Greedy solution

R1 = TAAAGC-G
R2 = TAAAGT-GG
R3 = AGTCGG
R4 = TCGG
R5 = AGC-G

Realign R5

R1 = TAAAG-CG
R2 = TAAAGT-GG
R3 = AGTCGG
R4 = TCGG
R5 = AG-CG

Realign R1

R1 = TAAAG-CG
R2 = TAAAGT-GG
R3 = AGTCGG
R4 = TCGG
R5 = AG-CG

Realign R5

R1 = TAAAG-CG
R2 = TAAAGT-GG
R3 = AGTCGG
R4 = TCGG
R5 = AG-CG

Cannot further improve
Long read by PacBio or Nanopore

• These two technologies enable us to sequence long reads of length >10kb
• Long reads generated by single-molecule sequencing currently suffer from low accuracy (error rate: 13-18% PacBio, 15-22% MinION)
• Although single-molecule sequencing may be error-prone, it exhibits less sequencing bias than previous technologies
Two approaches for genome assembly

- Hybrid approach
  - Use both long and short reads for genome assembly
  - E.g. ALLPATH-LG and PBcR

- Pure long read approach
  - Use long reads only for genome assembly
  - E.g. HGAP and MHAP
ALLPATH-LG

- Observation: Short reads are more accurate than long reads
- Idea: Construct assembly using short reads, then use mate-pair reads and long reads to scaffold and gap filling

PacBioToCA

**PBcR (PacBio corrected Reads)**

1. Align short Illumina reads on each long read and obtain the layout
2. The layout of the short reads on each long read is realigned.
3. Each long read is corrected by the consensus of the short reads
4. Using Celera assembler, reconstruct the genome using the corrected long reads

Hybrid error correction and *de novo* assembly of single-molecule sequencing reads. Nature Biotech 2012.
More on step 1 of PBcR

• Use BLASR, high-fidelity short reads are aligned on error-prone long reads.

• For each short read R,
  – R is allowed to align on multiple long reads.
  – Moreover, each R can align to exactly one location on each long read
  – Each R can align to the topmost C long reads (C is the estimated coverage of the long read)
Pure long read approach for genome assembly

- Use long read only
- Examples: FALCON, HGAP and MHAP

- Approach:
  1. Overlap the reads
  2. Correct the reads by the consensus
  3. Reconstruct the genome using the corrected reads

- Step 3 is just based on Celera. Below, we detail steps 1 and 2.
Step 1: Overlap the reads

- This step performs all-to-all pairwise alignment among all reads
- There are three approaches:
  - 1. HGAP: Long reads are selected as seed reads; then, align other reads on the seed reads using BLASR
  - 2. FALCON: Use Daligner (by Gene Myer) to perform all-to-all alignments
  - 3. MHAP: Use minHash to perform all-to-all alignment
All-pairs overlaps detection problem

- PacBio can generate long reads of length about 10k bp
- The accuracy is about 82-87%

- To perform overlap-layout-consensus assembly, we need to determine all-pairs overlaps of the long reads.
- Input: A set of long reads
- Output: All pairs of reads \((R_1, R_2)\) such that \(R_1\) and \(R_2\) have good similarity score
- This step consumes a big portion of the total runtime (e.g. for D. melanogaster SMRT assembly, this step takes 95% of the total time)
- There are a few solutions:
  - minHash, minimap, BLASR, Daligner
- Below, we describe the probabilistic algorithm based on minHash to efficiently detect the overlaps between noisy reads.
Naïve solution

• Suppose there are $N$ reads, each read is of length $L$.
• For every pairs of reads $R_1$, $R_2$,
  – Run Smith-Waterman alignment of $R_1$ and $R_2$
  – Report $(R_1, R_2)$ if the score is high

• Running time: $O(N^2L^2)$. 
Finding number of shared k-mers

• Input: Each read is represented by a set of k-mers.
• Output: Report all \((R_1, R_2)\) if they share at least \(w\) k-mers.

\[
R_1 = \text{ACTGCTTACG} \quad R_2 = \text{ACTCCTTATG}
\]

They share 3-mers: \(\text{ACT}, \text{CTT}, \text{TTA}\).
Hashing

• Build hash table for all k-mers.
• K is selected to be big enough so that there is little by-chance hit. (For human, the expected number of by-chance hits is \( \frac{4^K}{3 \times 10^9} \). If K=16, the expected by-chance hits < 1.)
• Building hash table takes O(N * (L-k+1)) = O(NL) time.

• For each read R1,
  – Maintain a count for other reads and initialize them to zero
  – For each k-mer X of R1,
    • get the list of reads in the hash entry X.
    • Increment the counts for the list of reads.
  – For every read R2 with count \( \geq w \), report (R1, R2).
All-to-all alignment in MHAP

• For every read $R_i$, we collect all the k-mers $kmer(R_i)$.
• Using $K$ hash functions ($h_1$, ..., $h_K$), we compute the hash values for all k-mers.
• $\text{sig}(R_i)$ is a length-$K$ vector where the $j^{th}$ entry equals $\min\{ h_j(\alpha) \mid \alpha \in \text{sig}(R_i) \}$
• By default, $k=16$, $K=512$(fast) or 1256(sensitive)
• To determine overlap offset between $R_i$ and $R_j$, we find the medium different in the positions for all k-mers in $\text{sig}(R_i) \cap \text{sig}(R_j)$.

$$\text{sim}(R_1, R_2) = \frac{2}{4} = 0.5$$

$R_1$: GATCAACGGACCCA  
  ATC  ACG  ACC  TCA  CGG  CCC  CAA  GGA  CCA  

$R_2$: TCACGACCCCATGTC  
  TCA  GAC  CAT  CAG  ACC  ATG  ACG  CCC  TGT  CGA  CCA  GTC  

$\text{sig}(R_1)$  $\text{sig}(R_2)$

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_2$</th>
<th>$h_3$</th>
<th>$h_4$</th>
<th>$h_5$</th>
<th>$h_6$</th>
<th>$h_7$</th>
<th>$h_8$</th>
<th>$h_9$</th>
<th>$h_{10}$</th>
<th>$h_{11}$</th>
<th>$h_{12}$</th>
<th>$h_{13}$</th>
<th>$h_{14}$</th>
<th>$h_{15}$</th>
<th>$h_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>26</td>
<td>21</td>
<td>GAT</td>
<td>TCA</td>
<td>CAG</td>
<td>ACC</td>
<td>GAC</td>
<td>ATC</td>
<td>CGG</td>
<td>CCC</td>
<td>CAA</td>
<td>ACG</td>
<td>ACC</td>
<td>CAA</td>
<td>CCA</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>31</td>
<td>TCA</td>
<td>8</td>
<td>33</td>
<td>15</td>
<td>ATC</td>
<td>2</td>
<td>31</td>
<td>51</td>
<td>CAA</td>
<td>22</td>
<td>13</td>
<td>9</td>
<td>AAC</td>
</tr>
<tr>
<td>31</td>
<td>18</td>
<td>10</td>
<td>51</td>
<td>CAA</td>
<td>14</td>
<td>2</td>
<td>9</td>
<td>27</td>
<td>ACG</td>
<td>CCC</td>
<td>25</td>
<td>39</td>
<td>33</td>
<td>GGA</td>
<td>27</td>
</tr>
<tr>
<td>25</td>
<td>39</td>
<td>53</td>
<td>33</td>
<td>GGA</td>
<td>27</td>
<td>34</td>
<td>19</td>
<td>63</td>
<td>GAC</td>
<td>CAT</td>
<td>17</td>
<td>ATG</td>
<td>10</td>
<td>53</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>24</td>
<td>11</td>
<td>ACC</td>
<td>9</td>
<td>27</td>
<td>45</td>
<td>39</td>
<td>CCC</td>
<td>23</td>
<td>58</td>
<td>12</td>
<td>61</td>
<td>41</td>
<td>61</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>27</td>
<td>45</td>
<td>39</td>
<td>CCC</td>
<td>23</td>
<td>58</td>
<td>12</td>
<td>61</td>
<td>CAT</td>
<td>41</td>
<td>61</td>
<td>18</td>
<td>57</td>
<td>10</td>
<td>53</td>
</tr>
<tr>
<td>23</td>
<td>42</td>
<td>12</td>
<td>61</td>
<td>CCA</td>
<td>23</td>
<td>58</td>
<td>12</td>
<td>61</td>
<td>CAT</td>
<td>41</td>
<td>61</td>
<td>18</td>
<td>57</td>
<td>10</td>
<td>53</td>
</tr>
</tbody>
</table>

$\text{sim}(R_1, R_2) = \frac{2}{4} = 0.5$
minHash estimates Jaccard Similarity accurately

1256 hashes

512 hashes

5000 red dots (non-overlapping reads) and 5000 blue dots (20% overlapping reads)
Step 2: Correct the reads by consensus

• Consider a template read T. Given a set of reads that overlap with T, this step corrects the errors in T.
  – HGAP: Use PBDAG-Con
  – FALCON and MHAP: Use FALCON Sense
FalconSense

• Input: a set of reads \{R_1, \ldots, R_n\} and a template T

1. Align \(R_x\) on T allowing matches or indels (no mismatches)

2. For each read \(R_x\),
   – let \(A_x\) be the alignment between \(R_x\) and \(T[p..q]\).
   – For each aligned position in \(A_x\),
     • if it is a match between \(R_x[i]\) and \(T[j]\), we include a tuple \(((j, 0), R_x[i])\)
     • If it is “-” aligned with \(T[j]\), we include a tuple \(((j, 0), “-”))\)
     • If it is a base \(R_x[i]\) aligned with “-”, we scan to the left and find the first match, says, between \(R_x[i']\) and \(T[j]\), then we include a tuple \(((j, (j-i')), R_x[i])\)

3. Sort all tuples \(((p, d), b)\) by increasing order of \(p, d\) and followed by the alphabetic order of \(b\).

4. Generate the consensus:
   – For each position \(j\), let \(C_j = \sum_{b \in \{A,C,G,T, -\}} count((j, 0), b)\).
   – If \(\text{count}((j,d),b) > C_j/2\), then output “b”; otherwise, output “-”.
FalconSense example

| R1 = TAA-G-CA | T = TAAAGT-AG | p = 12345667 | d = 00000010 |
| R2 = AGTCAG | T = TAAAGT-AG | p = 456678 | d = 000100 |
| R3 = GTCAG | T = TAAAGT-AG | p = 56678 | d = 00100 |
| R4 = TAA--TAG | T = TAAAGT-AG | p = 12345678 | d = 00000000 |

<table>
<thead>
<tr>
<th>Tuples</th>
<th>Count</th>
<th>Consensus</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, T)</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>(2, 0, A)</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>(3, 0, A)</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>(4, 0, A)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>(4, 0, -)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>(5, 0, G)</td>
<td>3</td>
<td>G</td>
</tr>
<tr>
<td>(5, 0, -)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>(6, 0, T)</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>(6, 0, -)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>(6, 1, C)</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>(7, 0, A)</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>(8, 0, G)</td>
<td>3</td>
<td>G</td>
</tr>
</tbody>
</table>

T = TAAAGT-AG
R1 = TAA-G-CA
R2 = AGTCAG
R3 = GTCAG
R4 = TAA--T-AG

========
TAA-GTCAG
End!