1 Algorithms

FM-index searching algorithms are closely related to performing searches using a suffix array. Below we briefly review nomenclature and previously established algorithms (1.1), provide an algorithm to find inexact overlaps (1.2), then we describe our error correction methods (1.3 and 1.4). In section 1.5 we provide additional details on our method of constructing an FM-index for large data sets.

1.1 FM-index search and overlap algorithms

The suffix array is a compact representation of the lexicographic ordering of the suffixes of a text [1]. Each element of the array is an index into the original string; \( \text{SA}_X[i] = j \) indicates that the suffix starting at position \( j \) in \( T \) is the \( i \)-th lowest suffix in \( X \). As an example consider the string \( T = \text{AGATCGATA} \). The suffix array of \( T \) is \( \text{SA}_T = [10, 9, 1, 7, 3, 5, 6, 2, 8, 4] \). As the suffix array is a sorted data structure, suffixes that start with a common string will occur in contiguous intervals. Such intervals are termed \textit{suffix array intervals} and exploited when performing pattern matches. For our example string \( T \) the substring \textit{GAT} occurs twice, at positions 2 and 6. In this case the suffix array interval is \([7, 8]\) which indicates that the 2nd and 6th suffixes of \( T \) start with \textit{GAT}. We can find suffix array intervals for a given a query string \( Q \) using \( \text{SA}_T \) and \( T \) with a binary search.

While the suffix array can efficiently perform pattern matches [2], it requires \( n \log(n) \) bits of storage space where \( n = |T| \). Alternatively, the FM-index can be used to calculate suffix array intervals while requiring far less memory [3]. Let \( C_T(a) \) be the number of symbols in \( T \) that are lexicographically lower than symbol \( a \) and \( \text{Occ}_T(a, i) \) be the number of occurrences of symbol \( a \) in \( B_T[1, i] \). Let \([l, u]\) be the suffix array interval for a string \( W \). We can compute the suffix array interval for a string one symbol longer, \( aW \), using \( C_T \) and \( \text{Occ}_T \):

\[
l' = C_T(a) + \text{Occ}_T(a, l - 1)
\]  

(1)
Using these equations, we can define an efficient algorithm to compute suffix array intervals for any query string $Q$:

\begin{equation}
u' = C_T(a) + \text{Occ}_T(a, u) - 1
\end{equation}

**Algorithm 1** computeInterval($T, Q$) - find the suffix array interval for query $Q$ in text $T$

```python
i ← |Q|
l ← C_T(Q[i])
u ← C_T(Q[i] + 1) - 1
i ← i - 1
while $l \leq u$ and $i \geq 1$ do
    $l ← C_T(Q[i]) + \text{Occ}_T(Q[i], l - 1)$
    $u ← C_T(Q[i]) + \text{Occ}_T(Q[i], u) - 1$
    $i ← i - 1$
return $[l, u]$
```

If the interval returned by computeInterval is invalid ($l > u$) then $Q$ does not occur in $T$ otherwise the number of occurrences of $Q$ in $T$ is given by the size of the interval ($u - l + 1$). This algorithm can be used to count the number of occurrences of a given substring (and its reverse complement) in a set of reads.

**Algorithm 2** countOccurrences($R$, $Q$) - count the number of times $Q$ and its reverse complement occurs in $R$

```python
c ← 0
[l, u] ← computeInterval($R$, $Q$)
if $l \leq u$ then
    $c ← u - l + 1$
[l, u] ← computeInterval($R$, $\overline{Q}$)
if $l \leq u$ then
    $c ← c + u - l + 1$
return $c$
```

The locations of the occurrences $Q$ in $T$ can be recovered using the suffix array. Typically it is too expensive to load the entire suffix array in memory so a subsample of suffix array positions are directly stored and the remainder are computed from the samples (for details see [4] or [5]). In our application, we are typically only interested in prefix-suffix overlaps between members of a set of reads. In this case, we obtain a suffix array interval $[l, u]$ corresponding to a prefix string $P$. These prefix string intervals correspond to lexicographic ranks (in the entire read set) of the reads with the prefix $P$. We store an array of size $n = |R|$ of the read indices sorted into lexicographic order by the read sequence. Using the suffix array intervals of the suffix-prefix matches and this array, we can recover the identifiers of the matching reads.
To construct an overlap or string graph, we must determine which members of a read set \( \mathcal{R} \) overlap by at least \( \tau \) bases. In [6] we provided an algorithm to do this, which we describe here. For simplicity, we encapsulate equations (1) and (2) into a function to update a suffix array interval:

**Algorithm 3** updateBackwards(\( \mathcal{R} \), \( [l, u] \), \( a \)) - update a suffix array interval using the FM-index for read set \( \mathcal{R} \)

\[
\begin{align*}
l &\leftarrow C_{\mathcal{R}}(a) + \text{Occ}_{\mathcal{R}}(a, l - 1) \\
u &\leftarrow C_{\mathcal{R}}(a) + \text{Occ}_{\mathcal{R}}(a, u) - 1 \\
\text{return } [l, u]
\end{align*}
\]

Using updateBackwards, we can define a procedure similar to computeInterval to find overlapping reads.

**Algorithm 4** findOverlaps(\( X \), \( \mathcal{R} \), \( \tau \)) - compute which reads in \( \mathcal{R} \) overlap a suffix of \( X \) by at least \( \tau \) symbols

\[
\begin{align*}
i &\leftarrow |X| \\
l &\leftarrow C_{\mathcal{R}}(X[i]) \\
u &\leftarrow C_{\mathcal{R}}(X[i] + 1) - 1 \\
i &\leftarrow i - 1 \\
\text{while } l \leq u \text{ and } i \geq 1 \text{ do} \\
&\quad o \leftarrow |X| - i + 1 \\
&\quad \text{if } o \geq \tau \text{ then} \\
&\quad \quad [l_s, u_s] \leftarrow \text{updateBackwards}(\mathcal{R}, [l, u], \$) \\
&\quad \quad \text{if } l_s \leq u_s \text{ then} \\
&\quad \quad \quad \text{outputOverlaps}(X, [l_s, u_s]) \\
&\quad \quad \quad [l, u] \leftarrow \text{updateBackwards}(\mathcal{R}, [l, u], X[i]) \\
&\quad \quad i \leftarrow i - 1 \\
&\quad \text{if } l \leq u \text{ then} \\
&\quad \quad \text{outputDuplicates}(X, [l, u])
\end{align*}
\]

The findOverlaps algorithm calculates suffix array intervals of strings that have a prefix matching a suffix of \( X \). The search is initialised to the last symbol in \( X \) and extended by one symbol in each iteration of the while loop. Once the match is at least \( \tau \) symbols in length, we check if the match is a valid prefix by querying the string \( \$P \) where \( P \) is the matched string. If this interval is non-empty, the interval \([l_s, u_s]\) provides the lexicographic ranks of the reads with the matched prefix. The corresponding read IDs can be recovered using the lexicographic index of the set \( \mathcal{R} \) and output by the subroutine outputOverlaps. At the end of the loop, the suffix array interval contains a match for the entire string \( X \) and any other reads that are identical in sequence to \( X \). These reads are output by outputDuplicates. This procedure calculates suffix-prefix overlaps. To calculate other types of overlaps, for instance suffix-suffix overlaps, we can use a similar procedure except complement the sequence of \( X \) and use the FM-index of the set of reversed reads. See [6] for further details.
1.2 FM-index inexact overlap algorithm

The overlap algorithm can be extended to allow mismatches in the overlaps. Let \( \epsilon \) be the maximum allowed mismatch rate between two overlapping strings (for example \( \epsilon = 0.05 \) would allow 1 mismatch in a 20bp overlap). At each stage of the extension, we can branch to each possible base \( A, C, G, T \). If the base that we extend to is different from the current position in \( X \), we increment a mismatch counter \( d \). If the value of \( d \) exceeds the maximum number of mismatches for an overlap of length \( |X| - 1 \) the current search path is terminated as a valid overlap cannot possibly be found. When an overlap of length at least \( \tau \) is found and the mismatch rate is at most \( \epsilon \) we output overlaps as in \texttt{findOverlaps}. We then recursively branch the search, updating the mismatch counter as needed.

The pseudocode for this algorithm is presented below in \texttt{findOverlapsInexact} and \texttt{findOverlapsInexactExtend}. While this matching algorithm will return the complete set of \( \tau, \epsilon \)-overlaps, it is inefficient. This naive search will branch excessively at the beginning of the search (when \( i \) is close to \( |X| \)) as the overlap lengths are not large enough to exclude strings that are matched by chance. Once the overlap length becomes long enough (i.e. for \( |X| - i \approx 16 \)) then most branches will not form valid matches (and hence have empty suffix array intervals) and therefore stop the recursion.

\begin{algorithm}
\caption{findOverlapsInexact(\( X \), \( R \), \( \tau \), \( \epsilon \)) - find all reads overlapping \( X \) by at least \( \tau \) bases with error rate at most \( \epsilon \)}
\begin{algorithmic}
\State \( i \leftarrow |X| \)
\ForAll{\( b \in [A, C, G, T] \)}
\State \( l \leftarrow C_R(b) \)
\State \( u \leftarrow C_R(b + 1) - 1 \)
\If{\( X[i] \neq b \)}
\State \texttt{findOverlapsInexactExtend}(\( X, i, 1, [l, u], R, \tau, \epsilon \))
\Else
\State \texttt{findOverlapsInexactExtend}(\( X, i, 0, [l, u], R, \tau, \epsilon \))
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}
Algorithm 6 findOverlapsInexactExtend($X$, $i$, $d$, [$l$, $u$], $R$, $\tau$, $\epsilon$) - perform one round of extension of the inexact search. The current suffix array interval is given by $l$ and $u$ which corresponds to a string from the end of $X$ to base $i$ with $d$ mismatches.

// check if the number of mismatches exceeds the maximum
// possible for a valid overlap
$m \leftarrow \lceil(\lvert X \rvert - 1) \times \epsilon \rceil$
if $d > m$ then
    return

// check if overlaps should be output
$o \leftarrow \lvert X \rvert - i + 1$
$r \leftarrow d/o$
if $o \geq \tau$ and $r \leq \epsilon$ then
    $[l, u] \leftarrow$ updateBackwards($R$, [$l$, $u$], $\$e$
    if $l \leq u$ then
        outputOverlaps($X$, [$l$, $u$])

// perform branched extension
if $i > 1$ then
    $i \leftarrow i - 1$
for all $b \in \{A, C, G, T\}$ do
    $[l', u'] \leftarrow$ updateBackward($R$, [$l$, $u$, $b$, $R$]
    if $l' \leq u'$ then
        if $X[i] \neq b$ then
            findOverlapsInexactExtend($X$, $i$, $d + 1$, $[l'$, $u']$, $R$, $\tau$, $\epsilon$
        else
            findOverlapsInexactExtend($X$, $i$, $d$, $[l'$, $u']$, $R$, $\tau$, $\epsilon$

We can design a more efficient algorithm using the seed-and-extend method of sequence alignment. This method is based on the principle that if we want to align a string $X$ to a text $T$ with up to $d$ mismatches, we can create $d + 1$ seeds over the sequence of $X$, one of which must be matched exactly to $T$. The seed matches can then be extended allowing for mismatches. We have adapted this method of alignment to finding inexact overlaps with the FM-index. We must take care when choosing the seed length to ensure that at least one seed matches exactly between any two reads that have a $\tau, \epsilon$-overlap. Let $d_\tau = \lceil \epsilon \tau \rceil$ be the maximum number of differences between two reads that overlap by the minimum amount. We define the minimal seed region of the read as $r_{min} = \lfloor d_\tau / \epsilon \rfloor$ and calculate the seed length $l_{seed} = \lceil r_{min} / (d_\tau + 1) \rceil$. This value of $l_{seed}$ is small enough such that for all overlap lengths $i \geq \tau$ we are guaranteed to have $\lceil i\epsilon \rceil + 1$ seeds covering the suffix of $X$ of length $i$ and hence we can find all $\tau, \epsilon$-overlaps.

Once the seed positions of $X$ have been calculated, we can find the suffix array intervals for the seeds using an exact match with the FM-index. The seeds can then be extended with a branching algorithm similar to that of findOverlapsInexact. We note that in this case, the extensions are not a strict right-to-left search as in findOverlapsInexact as some seeds start in
the middle of the read. We use the bidirectional search procedure outlined in [6, 7] to extend the seeds both left-to-right and right-to-left. We refer to these papers for details of this extension. See also [8] for a discussion of other inexact prefix-suffix matching algorithms.

1.3 \textit{k-mer} based error correction algorithm

The primary error correction algorithm in SGA is based on \textit{k-mer} frequencies. Assuming base-calling errors are a random process that occur independently, \textit{k}-mers covering a position in a read with an error will occur in the entire data set with low frequency (typically \textit{k}-mers covering an error will form unique strings). Our correction algorithm follows from other \textit{k}-mer based correctors [9, 10, 11] in that it attempts to identify positions in the read that are incorrect, then searches for a suitable correction. The algorithm scans each read left-to-right to identify bases that are not present in a \textit{k}-mer of frequency at least \textit{c}. We iterate over the potentially incorrect bases and change the base in the left-most \textit{k}-mer covering the position to the 3 other possibilities. If one of the possibilities yields a \textit{k}-mer with frequency at least \textit{c}, the change is made. If no correction can be found using the left-most covering \textit{k}-mer, the right-most covering \textit{k}-mer is tested. If this test also fails the procedure terminates and returns the original read sequence. If a set of corrections is found that makes all bases in the read trusted, the procedure terminates and returns the modified sequence. This algorithm is both time and space efficient as it only requires calls to \texttt{countOccurrences} to look up the \textit{k}-mer frequencies using the FM-index.

The correction algorithm has two parameters - the \textit{k}-mer size and the minimum required coverage, \textit{c}. The \textit{k}-mer value is typically found via experimentation with small subsets of the data. We select a subset of reads (approximately one million), run error correction for a particular choice of \textit{k} (using the FM-index of the entire data set) then evaluate the correction performance with the \texttt{sga stats} subprogram, which estimates the error rate in a set of reads. Iterating over this procedure allows a suitable choice of \textit{k} to be identified without requiring correction of the entire data set. The \textit{c} parameter can be manually provided or SGA can infer it by finding a trough in the \textit{k}-mer frequency histogram to differentiate low-frequency from high-frequency \textit{k}-mers. The \textit{c} parameter can optionally use the base quality scores to require more support for low quality bases (by default we consider quality scores less than phred-20 to be low).

1.4 Inexact overlap based error correction

The second error correction algorithm in SGA is based on finding inexact overlaps between reads in \textit{R}. Let \textit{X} be a read in \textit{R} that we want to correct. We use the seed-and-extend algorithm presented in section 1.2 to find all reads in \textit{R} with a \textit{τ,ϵ}-overlap with \textit{X}. This set of reads forms a multiple alignment with respect to \textit{X}. Let \textit{C}[i] be a 4 element vector of the counts for each base call in column \textit{i} of the multiple alignment. A simple consensus-based correction algorithm would be to set \textit{X}[i] to the element of \textit{C}[i] with the highest value. However, we must
take care to avoid collapsing variation (if the sequenced genome is diploid) or distinct copies of a repeat. We filter the multiple alignment by excluding reads that have consistent mismatches with respect to \( X \). If two elements of \( C[i] \) have a count of at least \( v \), we label position \( i \) as conflicted. The reads that match \( X \) at all conflicted columns are kept and the remainder excluded; \( C[i] \) is re-calculated from the filtered multiple alignment. We correct \( X[i] \) to be the consensus base indicated by \( C[i] \) if there is a single base in \( C[i] \) that occurs more than \( c \) times. This condition helps avoid setting \( X[i] \) to an incorrect base in the situation that multiple well-supported bases remain in the multiple alignment. The values of \( v \) (the conflict threshold) and \( c \) (the minimum base call support required) are command line parameters (the default values are 5 and 3, respectively).

1.5 Distributed construction of the FM-index
Our algorithm to construct the FM-index for very large read sets can be distributed over a cluster of computers. We use a modified version of the suffix array construction algorithm of Nong-Zhang-Chan [12] (see [6]) to construct the suffix array (and hence BWT) for subsets of the read collection (typically 2-4 million reads per subset). Once the initial indices are constructed, they are merged together using the BWT merging algorithm described in [13].

2 SGA Implementation
SGA is implemented in C++ as a set of modular subprograms. In this section, we briefly describe the major subprograms of SGA.

- \texttt{sga index} - constructs the FM-index for a set of sequence reads.
- \texttt{sga merge} - merges two indices together into a single index.
- \texttt{sga correct} - performs the error correction routines given in section 1.3 and 1.4.
- \texttt{sga filter} - removes duplicate reads and reads that contain low-frequency \( k \)-mers from the read set.
- \texttt{sga stats} - infers the error rate for a set of reads.
- \texttt{sga overlap} - constructs a string graph from the FM-index using the algorithm described in [6]
- \texttt{sga fm-merge} - detects and merges non-branching chains of reads
- \texttt{sga assemble} - simplifies the string graph by removing sequence variation bubbles and outputs contigs
- \texttt{sga scaffold} - reads in a scaffold graph and constructs linear scaffolds with gap size estimates
• sga scaffold2fasta - attempts to fill in scaffold gaps and outputs the scaffolds in FASTA format

Help for each subprogram is available from the command sga subprogram --help. Source code and further documentation is available at http://github.com/jts/sga.

3 Supplemental Methods

3.1 Human genome assembly and analysis

The sequence reads were downloaded from the 1000 Genomes FTP site as an aligned BAM file which included unaligned reads. We removed the alignment information using the RevertSAM command of the Picard toolkit. We separated the reverted BAM file into FASTQ files of 20 million reads each. As we chose to assemble half the data, we used 62 of the FASTQ files for the assembly. We built an FM-index of each FASTQ file then iteratively merged indices in pairs until a single index of the entire data remained. We corrected the reads using a k-mer size of 55. After correction, exact-match duplicate reads were removed and reads that contain a 27-mer seen fewer than 2 times in the entire read set were filtered out. Non-branching chains of reads were merged with a minimum overlap parameter of 65. The string graph was constructed from the merged sequences with a minimum overlap parameter of 77 bp. We realigned the original sequence reads to the contigs using bwa [4]. We required a minimum of 5 read pairs linking two contigs to create an edge in the scaffold graph. During scaffolding, we considered a contig to be unique if the A-statistic [14] (calculated from the read alignments) was at least 25. Only contigs at least 200bp in length were included in the scaffolds. Version 0.9.13 of SGA was used.

The complete SGA commands used to run the assembly can be found on the SGA website:


The following SOAPdenovo commands were run to perform the assembly:

• KmerFreq -q 0 -i correction.input -o correction
• Corrector -i correction.input -r correction.freq -t 4
• SOAPdenovo-63mer pregraph -s soap.config -K 61 -p 8 -o soap-k61 -R
• SOAPdenovo-63mer contig -g soap-k61 -M 2 -R
• SOAPdenovo-63mer map -p 8 -s soap.config -g soap-k61
• SOAPdenovo-63mer scaff -F -g soap-k61

In the SOAPdenovo configuration file, the avg_ins parameter was set to 400.
3.1.1 Mismatch Analysis

We downloaded a VCF file containing SNP calls for NA12878 from the Broad Institute (http://www.broadinstitute.org/gsa/wiki/index.php/Data_sets_for_a_framework_for_variation_discovery_and_genotyping_using_next-generation_DNA_sequencing_data). As the Broad calls were made for build 36 of the reference genome and we used build 37 for our analysis, we lifted the Broad coordinates to build 37 using the UCSC liftOver tool.

We constructed a BAM file for the SGA and SOAPdenovo assemblies containing only the full-length contig alignments to the autosomes and chromosome X of reference as described in the main text. We parsed the output of samtools mpileup [15] and classified each contig base aligned to the reference as either being a reference match, matching a VCF record for a SNP call or a mismatch. Ambiguous reference bases (“N”) were skipped. The mismatch rate was calculated by dividing the total number of aligned bases by the number of mismatches. We performed the same calculation for positions assembled by both SGA and SOAPdenovo by skipping reference positions that were not covered by a contig from both assemblies. We generated a masked version of the human reference genome by changing bases identified by RepeatMasker or annotated as a segmental duplication in the UCSC genome browser to “N”. The mismatch rate calculation was performed again, skipping the masked positions.

3.1.2 Detailed resource usage

Table 1: Running time and memory summary for the SGA human genome assembly

<table>
<thead>
<tr>
<th>Stage</th>
<th>Processes</th>
<th>Wall time</th>
<th>CPU time</th>
<th>Max Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build index (raw)</td>
<td>123</td>
<td>23 hr</td>
<td>187 hr</td>
<td>45 GB</td>
</tr>
<tr>
<td>Correct reads</td>
<td>63</td>
<td>1 hr</td>
<td>355 hr</td>
<td>28 GB</td>
</tr>
<tr>
<td>Build index (corrected)</td>
<td>123</td>
<td>32 hr</td>
<td>355 hr</td>
<td>44 GB</td>
</tr>
<tr>
<td>Filter reads</td>
<td>1</td>
<td>33 hr</td>
<td>167 hr</td>
<td>54 GB</td>
</tr>
<tr>
<td>Merge reads</td>
<td>1</td>
<td>15 hr</td>
<td>105 hr</td>
<td>48 GB</td>
</tr>
<tr>
<td>Assemble reads</td>
<td>3</td>
<td>23 hr</td>
<td>41 hr</td>
<td>16 GB</td>
</tr>
<tr>
<td>Align to contigs</td>
<td>62</td>
<td>6 hr</td>
<td>210 hr</td>
<td>10 GB</td>
</tr>
<tr>
<td>Build scaffolds</td>
<td>4</td>
<td>7 hr</td>
<td>7 hr</td>
<td>13 GB</td>
</tr>
<tr>
<td>All stages</td>
<td>-</td>
<td>140 hr</td>
<td>1427 hr</td>
<td>54 GB</td>
</tr>
</tbody>
</table>

Table 2: Running time and memory summary for the SOAPdenovo human genome assembly

<table>
<thead>
<tr>
<th>Stage</th>
<th>Processes</th>
<th>Wall time</th>
<th>CPU time</th>
<th>Max Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct reads</td>
<td>2</td>
<td>92 hr</td>
<td>312 hr</td>
<td>40 GB</td>
</tr>
<tr>
<td>Assemble reads</td>
<td>4</td>
<td>29 hr</td>
<td>167 hr</td>
<td>118 GB</td>
</tr>
<tr>
<td>All stages</td>
<td>-</td>
<td>121 hr</td>
<td>479 hr</td>
<td>118 GB</td>
</tr>
</tbody>
</table>
3.2 *C. elegans* assembly comparison

We used version 0.9.13 of SGA, version 1.0.14 of Velvet, version 1.2.6 of ABySS and version 1.05 of SOAPdenovo for the *C. elegans* assembly. For the de Bruijn graph based assemblers, the assembly was performed over all odd \( k \)-mers between 51 and 73. For Velvet, the `cov_cutoff` and `exp_cov` parameters were set to `auto` and the paired-end read insert size was set to 250. For ABySS, the minimum number of read pairs linking two contigs (\( n \) parameter to `abyss-pe`) was set to 5, quality-based trimming was disabled (\( q=0 \)) and the minimum contig seed length for the paired-end module was set to 500bp (\( s=500 \)). The SOAPdenovo assemblies were performed with the same commands used in the human genome assembly. In the configuration file, the `avg_ins` parameter was set to 250.

For SGA, the \( k \)-mer length used for correction was 41. The minimum coverage required was automatically determined from the data using the `--learn` flag to `sga correct`. After correction, we filtered the reads by requiring every 27-mer in each read was present in the entire data set at least twice. The minimum overlap parameter when constructing the string graph was 75. When building scaffolds, the minimum number of read pairs required to link two contigs was 5. Only contigs of length at least 200bp were included in the scaffolding process. A shell script implementing the SGA *C. elegans* assembly is provided in the SGA source repository:


3.2.1 Contig analysis

We first corrected differences between the sequenced individual and the *C. elegans* reference genome by calling a consensus sequence. We mapped the reads to the *C. elegans* reference (build WS222) using `bwa` [4]. We then called a consensus sequence using the following command as specified in the samtools documentation:

```
samtools mpileup -uf ref.fa aln.bam | bcftools view -cg -
```

The resulting consensus sequence was used for the contig analysis. The contigs alignments were analyzed using the same method as the human assembly. The consensus-corrected reference sequence contains ambiguity codes for bases that could not be confidently assigned a unique base call. These ambiguous bases were skipped when calculating the mismatch error rate of the contig sequences.

For the substring coverage analysis, we randomly selected 10,000 strings for each read length (50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000) from the consensus-corrected reference. The strings were aligned to the contigs using `bwa` requiring an exact match (`bwa aln -o 0 -n 0 -k 0`). Figure 2 in the main text plots the fraction of strings for each read length that mapped exactly to a contig.
3.2.2 SOAPdenovo with GapCloser

SOAPdenovo has an optional module for filling intra-scaffold gaps, called GapCloser. We performed SOAPdenovo assemblies of the *C. elegans* data with and without using GapCloser. Table 3 compares the assembly before and after using GapCloser. While the aligned contig N50 and the genome coverage increased, the contig accuracy substantially decreased.

Table 3: SOAPdenovo assembly of the *C. elegans* data before and after running GapCloser.

<table>
<thead>
<tr>
<th></th>
<th>Before GapCloser</th>
<th>After GapCloser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaffold N50 size</td>
<td>31.1 kbp</td>
<td>31.1 kbp</td>
</tr>
<tr>
<td>Aligned contig N50 size</td>
<td>16.0 kbp</td>
<td>25.0 kbp</td>
</tr>
<tr>
<td>Mean aligned contig size</td>
<td>5.6 kbp</td>
<td>7.7 kbp</td>
</tr>
<tr>
<td>Sum aligned contig size</td>
<td>95.4 Mbp</td>
<td>97.4 Mbp</td>
</tr>
<tr>
<td>Reference bases covered</td>
<td>95.1 Mbp</td>
<td>95.7 Mbp</td>
</tr>
<tr>
<td>Reference bases covered by contigs ≥ 1kb</td>
<td>92.3 Mbp</td>
<td>93.9 Mbp</td>
</tr>
<tr>
<td>Mismatch rate at assembled bases</td>
<td>1 per 26,585 bp</td>
<td>1 per 4,204 bp</td>
</tr>
<tr>
<td>Mismatch rate at bases covered by all assemblies</td>
<td>1 per 81,025 bp</td>
<td>1 per 5,278 bp</td>
</tr>
<tr>
<td>Contigs with split/bad alignment (Sum size)</td>
<td>483 (4.4 Mbp)</td>
<td>1194 (25.3 Mbp)</td>
</tr>
</tbody>
</table>

3.3 Error Correction Comparison

We used version 0.9.14 of SGA, version 0.2.2 of Quake (http://www.cbcb.umd.edu/software/quake/) and the latest version of HiTEC (http://www.csd.uwo.ca/~ilie/HiTEC/) as of February 12th, 2011.

SGA and Quake both require selection of the k-mer parameter. For SGA, we tried all odd k-mers between 17 and 31 (inclusive) for both the 20x and 50x data set. For Quake, we tried k-mers of length 15,16,17,18. We were unable to use larger k-mers for Quake as the memory requirement is 4k in the version that we used. Both SGA and Quake require a minimum coverage parameter, which both programs can infer from the input data. For Quake we used the provided script cov_model.py to select the minimum cutoff for correction and also used manually-selected values of 1,2,3,4,5. For SGA, the minimum coverage was automatically determined by enabling the --learn flag to sga_correct. The only input parameters to HiTEC are the expected genome size and error rate, which we set to 4,600,000 and 1.0%, respectively.

After correction, we performed assembly of the corrected reads using the SGA assembly module. No read filtering was performed, except for removing exact-match duplicate and contained reads. Read pairing information was not used. For the 20X data set, we required a minimum overlap of 45bp. For the 50X data set, we required a minimum overlap of 55bp. For each program and
data set we selected the error correction run which yielded the highest assembly N50 for further analysis. For SGA setting \( k = 17 \) for both the 20X and 50X data sets provided the largest N50. For Quake, the selected parameters were \( k = 16, c = 1 \) and \( k = 15, c = 4 \) for the 20X and 50X data sets, respectively.

Using \texttt{bwa} with default parameters in unpaired mode, we mapped the raw (uncorrected) reads and the selected corrected sets to the \textit{E. coli} K12 MG1655 reference genome. As this is the substrain sequenced in the experiment, we considered the reference genome to be the ground truth and therefore any mismatches in the read alignments to be sequencing errors. From the sequence alignments, we calculated the total number of bases aligned, the total number of mismatches in the alignments and the total number of perfect reads (those that aligned without errors). These numbers are presented in Table 2 and Table 3 in the main text. To report the CPU time and memory usage of each program, we parsed the output logs of the cluster environment that we used, LSF (load sharing facility). We assessed the contig mismatch rate as in the \textit{C. elegans} and human genome assemblies.

4 Data availability

The human genome and \textit{C. elegans} assemblies are available for download at 
ftp://ftp.sanger.ac.uk/pub/js18/
5 Supplementary Figures

Figure 1: The number of bases of the *C. elegans* reference genome covered as a function of minimum contig alignment length.

References


