Chapter 3

The SGA Assembler

3.1 Introduction

In the previous chapter, I described an algorithm to construct an assembly string graph Myers [2005] for a set of error-free sequence reads using the FM-index. In this chapter, I expand upon the algorithms to build a fully-functional sequence assembly program. I will describe algorithms to correct base calling errors, remove duplicate and contained sequences and construct contigs and scaffolds for real sequencing data. These algorithms are implemented in my software called SGA (String Graph Assembler)[1]. SGA is implemented as a modular pipeline, which allows it to be easily extended as improved algorithms are developed or sequencing technology changes.

3.1.1 Publication Note

The work described in this chapter was previously published in [Simpson and Durbin, 2012]. Sections 3.2.7, 3.3.1 and 3.3.5 describe currently unpublished results. The work described is the sole work of the author, under the supervision of his PhD supervisor, Richard Durbin.

[1] Source code available at www.github.com/jts/sga

3.1.2 Algorithm Overview

The SGA algorithm is based on performing queries over an FM-index constructed from a set of sequence reads. The SGA pipeline begins by preprocessing the sequence reads to filter or trim reads with multiple low-quality or ambiguous base calls. The FM-index is constructed from the filtered set of reads and base-calling substitution errors are detected and corrected using k-mer frequencies. The corrected reads are re-indexed then duplicated and contained sequences are removed, remaining low-quality sequences are filtered out and a string graph is built. Contigs are assembled from the string graph and constructed into scaffolds if paired end or mate pair data is available. Figure 3.1 depicts the flow of data through the SGA pipeline. I discuss the major components of SGA below.

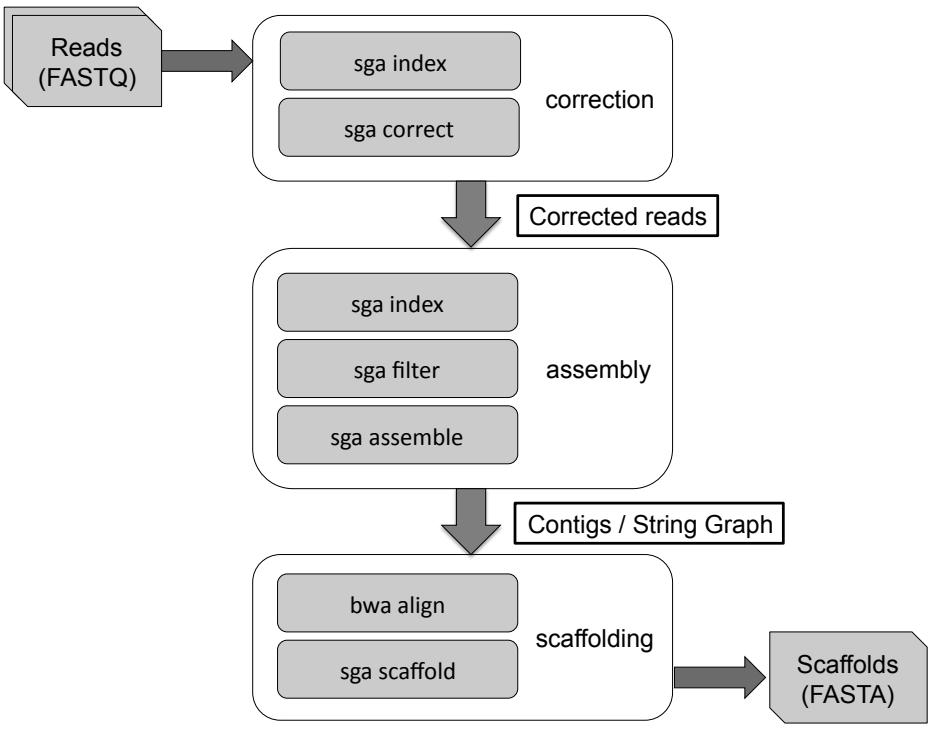

Figure 3.1: Schematic of the flow of data through SGA.

3.2 SGA Algorithms

3.2.1 Construction of the FM-index for large read sets

The algorithm begins with the construction of the FM-index for the complete set of reads. In Chapter 2, I described a modified version of the Nong-Zhang-Chan algorithm [Nong et al., 2009]. This algorithm has the drawback that to compute the Burrows-Wheeler transform of a set of reads, $\mathcal{R}$, it must first construct the full suffix array for a read set. The full suffix array requires $N \log(N)$ bits of memory, where N is the total number of bases in the read set. For a human genome sequenced to 30X coverage, this would require over 400GB of memory. As using this amount of memory during index construction would eliminate any benefit of using a compressed data structure for assembly, I have taken a different approach. I have implemented a distributed construction algorithm that builds an FM-index for subsets of $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$,..,$\mathcal{R}_m$. I then iteratively merge pairs of the intermediate indices together using a BWT merging algorithm [Ferragina et al., 2010] until a single index of the entire data set is obtained. As the space occupancy of the FM-index is typically less than an order of magnitude smaller than that of a suffix array, this indexing strategy allows us to efficiently build the FM-index for very large sequence collections. This construction strategy can be easily parallelized as the construction of the FM-index for each read subset, and most merging operations, can be computed independently.

Recently Bauer, Cox and Rosone designed an algorithm specifically tailored to the problem of computing the BWT for a very large collection of short ($\approx$ 100bp) sequence reads [Bauer et al., 2011]. Their algorithm directly computes the BWT of a read set without the need to first construct a suffix array. Their algorithm has two variants. Let n be the number of reads and l be the read length. The first variant, named `BCR`, uses $O(n \log(nl))$ bits of working space, and $O(l\texttt{sort}(n))$ time, where `sort(n)` is the time required to sort n integers. The second variant, `BCRext`, uses external memory (disk storage) for most data structures. It requires $O(ln)$ time with constant memory usage and overall I/O volume $O(l^2 n)$. Both algorithms work by progressively building partial BWTs, starting from the last base of each read. At each iteration j, the position to insert the next base of

each read into the partial BWT can be calculated from the previous iteration, $j-1$. In both variants of the algorithm the partial BWTs are stored on disk to save memory. As `BCR` and `BCRext` use disk-based storage to store intermediate files, their performance is dependent on the I/O and seek times of the underlying harware[1]. For this reason, I implemented a variant of `BCR` within SGA that uses in-memory storage of the partial BWT files. I compare the performance of the indexing algorithms in section 3.3.1. All algorithms can be used by SGA - the Nong-Zhang-Chan and in-memory BCR algorithms are natively implemented in SGA. `BCR` and `BCRext` are available by running the authors' reference implementation (BEETL[2]) then running a script to convert the output into SGA's BWT file format.

3.2.2 k-mer based error correction algorithm

Real sequencing data contains base calling errors. SGA's error corrector is currently designed to handle substitution errors, which are the dominant error mode in the Illumina sequencing platform [Bentley et al., 2008]. I have implemented two error correction methods. The first is a k-mer frequency-based corrector, which has been successfully used in other sequence assemblers [Kelley et al., 2010; Li et al., 2010c; Pevzner et al., 2001]. The second algorithm is based on finding inexact overlaps between sequence reads. In my tests the k-mer based corrector is faster than the overlap-based corrector and is therefore the default method of correction using SGA. Both correction methods have an option to use per-base quality scores of the read being corrected to vary the coverage threshold required to support a base call.

The primary error correction algorithm in SGA is based on k-mer frequencies. Assuming base-calling errors are a random process that occur independently, k-mers covering a incorrectly-called base in a read will occur in the entire data set with low frequency (typically k-mers covering an error will form unique strings). This is illustrated in figure 3.2, which plots a histogram of k-mer frequencies for simulated error-free data and simulated data with 1% error rate. For both data

[1] The authors of BCRext found that Solid State Drives offered the fastest indexing performance

[2] www.github.com/BEETL/BEETL

sets, 30X coverage of 100bp reads from the *E. coli* genome was generated. In the perfect data, there are almost no k-mers in the read set that are seen less than 5 times. In contrast, for the 1% error rate data, there is a large proportion of k-mers are low frequency (<5 occurrences). These are k-mers that are very likely to contain sequencing errors. We can therefore use k-mer occurrence counts to distinguish between correct and erroneous k-mer strings.

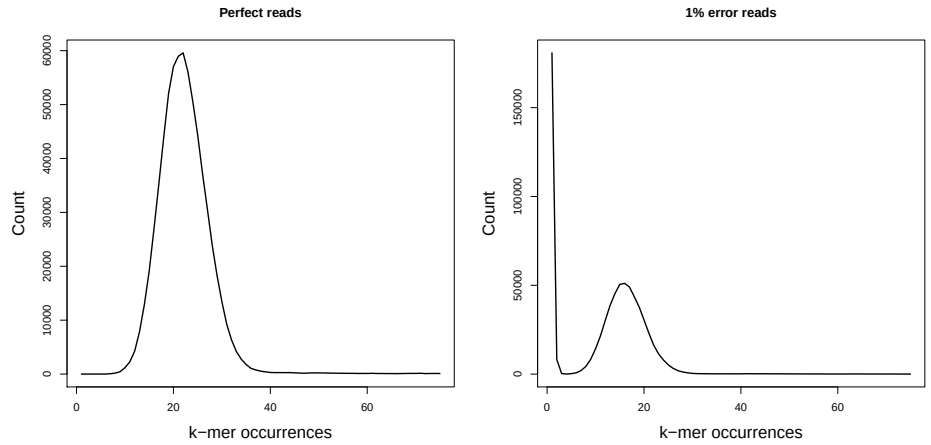

Figure 3.2: k-mer occurrence histogram for simulated perfect data (left) and simulated data with 1% uniform base calling errors (right). The y-axis records the number of times a k-mer with frequency x occurs in samples of the data set. For example, there are 57,059 k-mers seen 20 times in the perfect data set. The histogram was calculated by sampling 10,000 random reads.

Our correction algorithm follows from other k-mer based correctors [Kelley et al., 2010; Li et al., 2010c; Pevzner et al., 2001] 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 k-mer of frequency at least c. We iterate over the potentially incorrect bases and change the base in the left-most k-mer covering the position to the 3 other possibilities. If exactly one of the possibilities yields a k-mer with frequency at least c, the correction is made. If no correction can be found using the left-most covering k-mer, the right-most covering 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 (frequency $\geq c$), then the procedure terminates and returns the modified sequence.

The minimum coverage parameter c is conservatively chosen to avoid collapsing SNPs (if the genome is diploid) or distinct copies of a repeat. This parameter can either be manually provided or automatically selected by SGA by finding the trough in the k-mer frequency histogram.

Unlike previous k-mer based error correctors, the k-mer frequencies are not stored in a lookup or hash-table but rather directly calculated from the FM-index. Each k-mer frequency lookup in the FM-index only requires $O(k)$ time when using the algorithm `countOccurrences`.

Algorithm 9 countOccurrences($\mathcal{R}$, Q) - count the number of times Q and its reverse complement occurs in $\mathcal{R}$

$c \leftarrow 0$
$[l, u] \leftarrow$ `backwardsSearch`$(\mathcal{R}, Q)$
if $l \leq u$ **then**
  $c \leftarrow u - l + 1$
$[l, u] \leftarrow$ `backwardsSearch`$(\mathcal{R}, \overline{Q})$
if $l \leq u$ **then**
  $c \leftarrow c + u - l + 1$
return c

In addition to being computationally efficient, this has the advantage of using comparatively little memory and allowing greater flexibility in the parameter choices as the FM-index can support any value of k, unlike a hash table which must be reconstructed for each choice of k.

3.2.3 Overlap based error correction

The second error correction algorithm in SGA is based on finding inexact overlaps between reads. In the next section, I describe the algorithm to compute overlaps. In the following section, I describe how these overlaps are used to correct reads.

3.2.3.1 Finding Inexact Overlaps with the FM-Index

The overlap algorithm from 2.5.2 can be extended to allow mismatches in the overlaps. Let ϵ 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 overlap 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 τ is found and the mismatch rate is at most ϵ we output overlaps as in `findOverlaps`. We then recursively branch the search, updating the mismatch counter as needed. The pseudocode for this algorithm is presented below in `findOverlapsInexact` and `findOverlapsInexactExtend`. While this matching algorithm will return the complete set of τ, ϵ-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.

Algorithm 10 findOverlapsInexact(X, $\mathcal{R}$, τ, ϵ) - find all reads overlapping X by at least τ bases with error rate at most ϵ

```
i ← |X|
for all b ∈ [A, C, G, T] do
    l ← C_R(b)
    u ← C_R(b + 1) − 1
    if X[i] ≠ b then
        findOverlapsInexactExtend(X, i, 1, [l, u], R, τ, ε)
    else
        findOverlapsInexactExtend(X, i, 0, [l, u], R, τ, ε)
```

Algorithm 11 findOverlapsInexactExtend(X, i, d, $[l, u]$, $\mathcal{R}$, τ, ϵ) - 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 ← ⌊(|X| − 1) ∗ ε⌋
if d > m then
  return
// check if overlaps should be output
o ← |X| − i + 1
r ← d/o
if o ≥ τ and r ≤ ε then
  [l$, u$] ← updateBackwards(R, [l, u], $)
  if l$ ≤ u$ then
    outputOverlaps(X, [l$, u$])
// perform branched extension
if i > 1 then
  i ← i − 1
  for all b ∈ [A, C, G, T] do
    [l′, u′] ← updateBackward(R, [l, u], b, R)
    if l′ ≤ u′ then
      if X[i] ≠ b then
        findOverlapsInexactExtend(X, i, d + 1, [l′, u′], R, τ, ε)
      else
        findOverlapsInexactExtend(X, i, d, [l′, u′], R, τ, ε)
```

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 τ, ϵ-overlap. Let $d_\tau = \lfloor \epsilon\tau \rfloor$ 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 $r_{min} = \lceil d_\tau/\epsilon \rceil$ and calculate the seed length $l_{seed} = \lfloor r_{min}/(d_\tau + 1) \rfloor$. This value of l_{seed} is small enough such that for all overlap lengths $i \geq \tau$ we are guaranteed to have $\lfloor i\epsilon \rfloor + 1$ seeds covering the suffix of X of length i and hence we can find all τ, ϵ-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 Chapter 2 to extend the seeds both left-to-right and right-to-left. See also [Välimäki et al., 2010] for a discussion of other inexact prefix-suffix matching algorithms.

3.2.3.2 Overlap Based Error Correction Algorithm

Let X be a read in $\mathcal{R}$ that we want to correct. We use the seed-and-extend algorithm presented in the previous section to find all reads in $\mathcal{R}$ with a τ, ϵ-overlap with X. This set of reads forms a multiple alignment with respect to X. Let $C[i]$ be a 4 element vector of the counts for each base call in column i of the multiple alignment. A simple consensus-based correction algorithm would be to set $X[i]$ to the element of $C[i]$ 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).

As the number of overlaps for a given read is dependent on sequence depth, the runtime of the overlap based algorithm is dependent on the depth of sequencing. The run time of k-mer based algorithm presented in the previous section does not depend on the sequence depth. For this reason, it tends to be much faster than the overlap-based algorithm presented here, and therefore the k-mer based algorithm is the default method of correction used in SGA.

3.2.4 Read filtering

To construct the string graph we require a subset of $\mathcal{R}$ consisting of unique reads. We achieve this by removing contained and duplicated reads. To compute this subset, we use the FM-index to calculate full-length matches for each read in $\mathcal{R}$. If a read $\mathcal{R}_i$ has a full length match (including reverse complements) to some other read $\mathcal{R}_j$ we keep $\mathcal{R}_i$ iff $i < j$, otherwise $\mathcal{R}_i$ is discarded. Once the unique subset $\mathcal{U}$ of $\mathcal{R}$ has been calculated, we do not need to re-compute the FM-index of $\mathcal{U}$ from scratch. The BWT of $\mathcal{U}$ can be derived from the FM-index of $\mathcal{R}$ by marking the positions in $\mathbf{B}_{\mathcal{R}}$ that correspond to reads that were discarded and exporting only the unmarked positions as $\mathbf{B}_{\mathcal{U}}$ [Sirén, 2009].

Some reads remain uncorrected after error correction. To prevent these sequences from impacting the assembly, we remove sequences with unique k-mers. By default, this filter requires all 27-mers in a read to be seen at least twice.

3.2.5 Read merging and assembly algorithm

After correction and filtering, the vast majority of the remaining reads do not contain errors. We could directly apply our string graph construction algorithm (section 2.5.3) to these, however the resulting graph would have a vertex for every read and therefore require a substantial amount of memory when assembling very large genomes (as demonstrated in table 2.1). The majority of reads in the initial graph are simply connected (that is, without branching) to two other reads - one matching a prefix of the read and one matching a suffix. Such chains of reads, referred to as *unipaths*, can be unambiguously merged to reduce the size of the

graph. We have developed an algorithm to merge unipaths by locally constructing the assembly graph around each read. For each read, we find the predecessor and successor vertices in the graph by querying the FM-index for its irreducible edge set using `findIntervals` and `extractIrreducible` from section 2.5.3. If the read connects to its neighbors without branching, we continue the search from the neighboring reads. This search stops when a branch in the graph, or no possible extension, is found. This procedure will discover all non-branching chains in the graph and allow the chain to be replaced by a single merged sequence. As the predecessor/successor queries only require the FM-index, not the complete structure of the graph, this merging step requires comparatively little memory. Once we have performed this merging step, we build an FM-index for the merged sequences and use this FM-index to construct the full string graph. We then perform the standard assembly graph post-processing step of removing tips (see section 1.2.2.1) from the graph where a vertex only has a connection in one direction [Chaisson and Pevzner, 2008; Li et al., 2010c; Simpson et al., 2009; Zerbino and Birney, 2008].

To account for heterozygosity in a diploid genome, we have developed an algorithm to find and catalog variation described by the structure of the graph, similar to the "bubble-popping" approaches taken by de Bruijn graph assemblers. Let v be a vertex in the graph which branches (the prefix or suffix of v has multiple overlaps). Following each branch, we search outwards from v for a set of walks, $\mathcal{W}$, which meets the following conditions: 1) all walks terminate at a common vertex u and 2) no vertex visited in any walk between v and u has an edge to a vertex that is not present in a walk in $\mathcal{W}$. The first condition ensures that the walks describe equivalent sequence in G - any assembly of G that visits v and u must use one of the found walks. The second condition ensures that the induced subgraph of G described by the walks is self-contained - we can remove any walk in $\mathcal{W}$ without breaking any walk in $G \setminus W$. Once a set of walks meeting these conditions has been found, we select one of the walks to remain in the graph. We align the sequence described by the other walks to the sequence of the selected walk and, if the sequence similarity is within tolerance (by default 95%) in all cases, the non-selected walks are removed from the graph. We retain the sequences of the removed walks in a FASTA file to allow the heterozygous

variation present in the genome to be analyzed after assembly.

3.2.6 Paired end reads/Scaffolding

The final stage of the assembly is to build scaffolds from the contigs using paired-end or mate-pair data. Similar to other approaches to scaffolding [Pop et al., 2004], our method is based on constructing a graph of the relationships between contigs. We begin by re-aligning the paired reads to the contigs using `bwa` [Li and Durbin, 2009]. The copy number of each contig in the source genome is estimated from the number of reads aligned to the contig using Myers' A-statistic which approximates the log-odds ratio between the contig being unique and a collapsed repeat [Myers, 2005]. By default, we classify contigs with an A-statistic ≥ 20 as unique and the remainder as repetitive. We construct a scaffold graph where each unique contig is a vertex. Contigs linked with read pairs are connected by a bidirected edge labeled with the estimated gap size separating the contigs. Paths through this scaffold graph describe layouts of the contigs into scaffolds. The gap sizes are estimated using the `DistanceEst` subprogram from ABySS [Simpson et al., 2009].

Our scaffolder first removes ambiguous or likely erroneous edges from the graph. For each contig in the graph with more than one edge in a particular direction, we test whether the linked contigs have an ordering that is consistent with each pairwise distance estimate. An ordering of contigs $C_1, C_2, ..., C_n$ is called consistent if no pair of contigs has an overlap (implied by their positions in the layout) greater than α bases ($\alpha = 400$ by default). If the contigs cannot be consistently ordered, we break the graph by removing all edges of the affected contigs.

Once the graph has been cleaned of inconsistent edges, we find and isolate any directed cycles then compute the connected components of the graph. For each connected component, we find the terminal vertices of the component (vertices that have an edge in only one direction) and find all paths between each pair of terminal vertices. The path containing the largest amount of sequence is retained as the primary layout of the scaffold. The SGA scaffolder supports multiple libraries of different sizes.

The scaffolds are represented as an alternating list of contigs and gaps, $C_1, g_1, C_2, g_2, ..., C_n$. We attempt to fill in the gaps through a three-stage process. Let C_i and C_j be two adjacent contigs separated by a distance of g_i. As C_i and C_j are vertices in the string graph we previously constructed, we search the string graph for a walk connecting these vertices with the constraint that the total walk length can be no larger than $|C_i| + |C_j| + g_i + \theta_i$ where θ_i allows for the inexact distance estimate (by default 3 times the standard error of the distance estimate). If a single walk is found to meet this constraint, we replace C_i, g_i, C_j in the scaffold by the walk string. If no walk can be found connecting the vertices and g_i is negative (the contigs are predicted to overlap), we align the ends of C_i and C_j. If the predicted overlap is confirmed to exist, the sequences of C_i and C_j are merged. If the gap cannot be resolved, we simply fill the sequence between C_i and C_j with g_i ambiguity ("N") symbols.

As described in section 2.6, we can use the FM-index as a representation of the de Bruijn graph for all k. In the latest version of SGA, we can use this feature to optionally fill in scaffold gaps by finding walks through a de Bruijn graph. Let C_i, g_i, C_j be two contigs in a scaffold separated by a gap. Starting at $k = 91$, we use the last k-mer of C_i to seed a breadth-first search through the 91-mer de Bruijn graph. If a path through the graph ending at the first k-mer of C_j can be found, and the length of the path is within 100bp of the estimated size of the gap, the gap is replaced by the string corresponding to the path. To avoid searching very dense regions of the graph, the breadth-first search aborts if more than 2000 vertices have been visited, if the search branches more than 50 times or if more than 20 branches are being simultaneously following. If the gap cannot be successfully filled, k is decreased by 10 and the procedure restarts. This continues until the gap is filled or k is less than 51. The starting and stopping k are parameters to the program. There also exists an option to ignore k-mers that are seen less than t times in the reads ($t = 3$ by default). This method is typically able to fill 10-20% of the scaffold gaps, leading to slightly increased contig N50. This de Bruijn graph gap-filling procedure is a standalone component of SGA - it can be used on scaffolds from any assembler.

3.2.7 Implementation Details

3.2.7.1 FM-Index Implementation

SGA relies on pattern searches over the FM-index for both error correction and the construction of the string graph. While the FM-index provides optimal $O(|P|)$ queries for a pattern P, the implementation of the data structure has a significant impact on the performance of these queries. In this section I describe the implementation of the FM-index within SGA.

As described in section 2.4, the FM-index of a string X over an alphabet Σ consists of three data structures:

- $\mathbf{B}_X$ - the Burrows-Wheeler transform of X
- $\mathbf{Occ}_X(a, i)$ - the number of occurrences of the symbol a in $\mathbf{B}_X[1, i]$.
- $\mathbf{C}_X(a)$ - the number of symbols in X that are lexicographically lower than the symbol a

$\mathbf{C}_X$ only requires storing $|\Sigma|$ integers. If we explicitly stored $\mathbf{Occ}_X(a, i)$ for all $i \in \{1, |X|\}$ the amount of memory required would be $|X||\Sigma| \log_2(|X|)$. In our case with an alphabet of size 5, $|\Sigma| \log_2(|X|) \approx 40$ bytes. This huge memory cost would offset any benefit of using the FM-index. A common method to reduce the memory usage is to only store $\mathbf{Occ}_X(a, i)$ for i which is a multiple of a fixed value d. When a value $\mathbf{Occ}_X(a, j)$ is requested that is not explicitly stored, the closest stored value to j, $\mathbf{Occ}_X(a, k)$ is looked up and the requested value is calculated by explicitly counting the symbols in $\mathbf{B}_X$ between j and k. This allows the memory usage to be reduced to $40|X|/d$ bytes. The value of d offers a tradeoff between space and time. Larger values of d will use less memory but require longer stretches of $\mathbf{B}_X$ to be traversed during counting. In SGA, we use a two-tier encoding of the occurrence array similar to the encoding used in [Healy et al., 2003][1]. We store the absolute number of times each symbol have been seen in $\mathbf{B}_X[1, i]$ using an 8 byte integer for all i divisible by 8192. We call these absolute counts *large markers*. For all i divisible by d, we store a two-byte integer *small marker* which is the symbol count relative to the previous large marker.

[1] This implementation of the occurrence array was suggested to us by Travis Wheeler

This encoding requires an extra addition when calculating a value of $\mathbf{Occ}_X$, but lowers the memory usage to $40|X|/8192 + 10|X|/d$ bytes. The d parameter is chosen by the user at runtime and defaults to 128.

A naive encoding of $\mathbf{B}_X$ would require $|X| \log_2 |\Sigma|$ bits. However, as the Burrows-Wheeler transform sorts substrings of X, $\mathbf{B}_X$ contains long runs of repeated symbols. To account for this, we use run length encoding for $\mathbf{B}_X$. Each run is a byte encoding a <symbol, length> pair. We use 3 bits for the symbol and 5 bits for the length of the run. This encoding scheme is efficient for high-coverage data and requires ≈ 1.3 bits per base on average for high-depth data. However, when the run lengths are short due to lack of coverage or sequencing errors, this encoding scheme is inefficient. During the development of SGA I experimented with different methods of encoding $\mathbf{B}_X$, including Huffman and Golomb coding. Each of these encodings offered greater space efficiency than the <symbol, length> pair encoding, however they were all slower to decode during the critical $\mathbf{Occ}_X$ counting procedure. This issue remains to be further investigated as significant space savings could be made in SGA. Ideally, the FM-index would be implemented as a separate software library, allowing the time/space tradeoff to be selectable by the user.

3.2.7.2 Progam Design, Implementation and Libraries

SGA is implemented in C++. It uses zlib (`www.zlib.net`) to read compressed files and BamTools [Barnett et al., 2011] to read SAM/BAM files. It is multi-threaded using pthreads and the hoard parallel memory allocator [Berger et al., 2000]. The source code is licensed under GPLv3 and freely available online `www.github.com/jts/sga`.

SGA is designed to be modular, so that components of the assembler can be replaced as improved algorithms are available. The major components of SGA are listed below.

- `sga index` - constructs the FM-index for a set of sequence reads.
- `sga merge` - merges two indices together into a single index.

- `sga correct` - performs the error correction routines described in section 3.2.2 and 3.2.3.
- `sga filter` - removes duplicate reads and reads that contain low-frequency k-mers from the read set.
- `sga stats` - infers the error rate for a set of reads.
- `sga overlap` - constructs a string graph from the FM-index using the algorithm described in Simpson and Durbin [2010].
- `sga fm-merge` - detects and merges unipaths in the string graph.
- `sga assemble` - simplifies the string graph by removing sequence variation bubbles and outputs contigs.
- `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.
- `sga gapfill` - standalone gapfiller based on de Bruijn graphs.

3.3 Results

In this section, I demonstrate the use of SGA on a variety of real data sets. I begin by benchmarking the indexing performance of the algorithms discussed in section 3.2.1. I then perform an assembly of a medium sized genome to compare the performance of SGA against widely used de Bruijn graph assemblers. I also use SGA to assemble a human genome, demonstrating the benefits of using a compressed data structure on memory usage. Finally, I describe the assembly of 104 *Schizosaccharomyces pombe* yeast strains.

Program	CPU time	Walltime	Max Memory
sga-sais	12581s	12567s	17.0 GB
sga-bcr	5393s	5367s	9.3 GB
BCR	9553s	18184s	1.1 GB
BCRext	4761s	7953s	0.05 GB

Table 3.1: Running time and memory usage for four different methods of constructing the BWT for 66.7 million 100bp reads.

3.3.1 Index construction results

I profiled the SGA implemention of Nong-Zhang-Chan (`sga-sais`), `BCR`, `BCRext` and the SGA implementation of in-memory `BCR` (`sga-bcr`) on 66.7 million 100bp reads from the *C. elegans* genome. In this test, I used version 0.9.20 of SGA and version 0.0.2 of the BCR authors' non-commerical reference implementation, named BEETL[1]. To conserve memory in `sga-sais`, the read set was broken into 4 subsets. `sga-sais` was used to calculate the BWT of each subset, then 2 BWT merging rounds were performed to compute the final result. For the other three algorithms the BWT was directly constructed from the full read set.

Each test was run on an Intel Xeon X5650 CPU (2.67GHz) with 38GB of available memory. The input, temporary and output files were stored on a Lustre parallel distributed file system. All programs were run with a single thread. The results are summarized in table 3.1. The wall-clock, cpu time and memory usage was all measured by our cluster computing environment, LSF (Load Sharing Facility).

The BCRext algorithm required negligable memory and was the fastest program when measured by CPU time. My in-memory implementation of BCR was the fastest program when measured by wall-clock time, however it required 9.3GB of memory. The Nong-Zhang-Chan algorithm implemented in SGA required the most CPU time and the most memory.

[1] `www.github.com/BEETL/BEETL`

3.3.2 *C. elegans* Assembly

To assess the performance of SGA I performed assemblies of the nematode *C. elegans* using SGA and three other assemblers. The Velvet assembler [Zerbino and Birney, 2008] was one of the first de Bruijn graph-based assemblers for short reads and has become a standard tool for assembling small to medium sized genomes. The ABySS assembler [Simpson et al., 2009] was developed to handle large genomes by distributing a de Bruijn graph across a cluster of computers. SOAPdenovo is also based on the de Bruijn graph and designed to assemble large genomes [Li et al., 2010b,c].

C. elegans provides a good real-world test case for assembly algorithms because it has a complete and accurate reference sequence [C. elegans Sequencing Consortium, 1998], it propagates as a hermaphrodite so the genome of an individual (or strain) is homozygous and essentially free of SNPs and structural variants, and the genome is a reasonable size for evaluation (100 Mbp). I downloaded *C. elegans* sequence reads (strain N2) from the NCBI SRA (accession SRX026594). The data set consists of 33.8M read pairs sequenced using the Illumina Genome Analyzer II. The mean DNA fragment size is 250 bp from which reads of length 100 bp were taken from both ends of the fragment. To reduce the impact of differences between the sequenced individual and the reference sequence, I called a new consensus sequence for the *C. elegans* reference genome (build WS222, www.wormbase.org) based on alignments of the reads to the reference using `samtools` as specified by the documentation[1].

As sequence assemblers are often sensitive to the input parameters, I performed multiple runs with each assembler. The de Bruijn graph assemblers were run for all odd k-mer sizes between 51 and 73 (inclusive). The k-mer size providing the largest scaffold N50 was selected for further analysis (67 for ABySS, 61 for Velvet, 59 for SOAPdenovo). Similarly, for SGA the k-mer size used for error correction and the minimum overlap parameter for assembly were selected to provide the largest scaffold N50 (k=41 for error correction, τ=75 for the minimum overlap). I also performed a SOAPdenovo assembly using their `GapCloser` program after scaffolding. `GapCloser` was able to fill in many gaps within scaffolds,

[1] The command run was `samtools mpileup -uf ref.fa aln.bam | bcftools view -cg -`

which increased the contig N50 and genome coverage. However, these increases came at the cost of substantially lowered accuracy. In the following analysis, I use the SOAPdenovo assembly without using GapCloser.

I broke the assembled scaffolds into their constituent contigs by splitting each scaffold whenever a run of "N" bases was found. I filtered the contig set by removing short contigs ($< 200bp$ in length). The remaining contigs were aligned to the consensus-corrected reference genome using bwa-sw [Li and Durbin, 2010] with default parameters. I considered a number of different assessment criteria, which are described below and summarized in table 3.2.

	SGA	Velvet	ABySS	SOAPdenovo
Scaffold N50 size	26.3 kbp	31.3 kbp	23.8 kbp	31.1 kbp
Aligned contig N50 size	16.8 kbp	13.6 kbp	18.4 kbp	16.0 kbp
Mean aligned contig size	4.9 kbp	5.3 kbp	6.0 kbp	5.6 kbp
Sum aligned contig size	96.8 Mbp	95.2 Mbp	98.3 Mbp	95.4 Mbp
Reference bases covered	96.2 Mbp	94.8 Mbp	95.9 Mbp	95.1 Mbp
Reference bases covered by contigs $\geq$ 1kb	93.0 Mbp	92.1 Mbp	93.9 Mbp	92.3 Mbp
Mismatch rate at all assembled bases	1 per 21,545 bp	1 per 8,786 bp	1 per 5,577 bp	1 per 26,585 bp
Mismatch rate at bases covered by all assemblies	1 per 82,573 bp	1 per 18,012 bp	1 per 8,209 bp	1 per 81,025 bp
Contigs with split/bad alignment (Sum size)	458 (4.4 Mbp)	787 (7.2 Mbp)	638 (9.1 Mbp)	483 (4.4 Mbp)
Total CPU time	41 hr	2 hr	5 hr	13 hr
Max Memory usage	4.5 GB	23.0 GB	14.1 GB	38.8 GB

Table 3.2: Assessment of various assembly programs on the C. elegans data set.

3.3.2.1 Substring coverage

For the first assessment, I sampled strings from the consensus sequence and tested whether they were exactly present in the contigs. I sampled 10,000 strings of length from 50 bp up to 5,000 bp. This assessment combines three measures; the contigs must be accurate (as exact matches are required), complete (as the string must be present in the contig) and contiguous (as strings broken between multiple

contigs will not be found). Figure 3.3 plots the proportion of strings found in the contigs as a function of the string length. All assemblers perform well for short strings (50 to 100 bp). For longer string lengths, SGA slightly outperforms the other three assemblers.

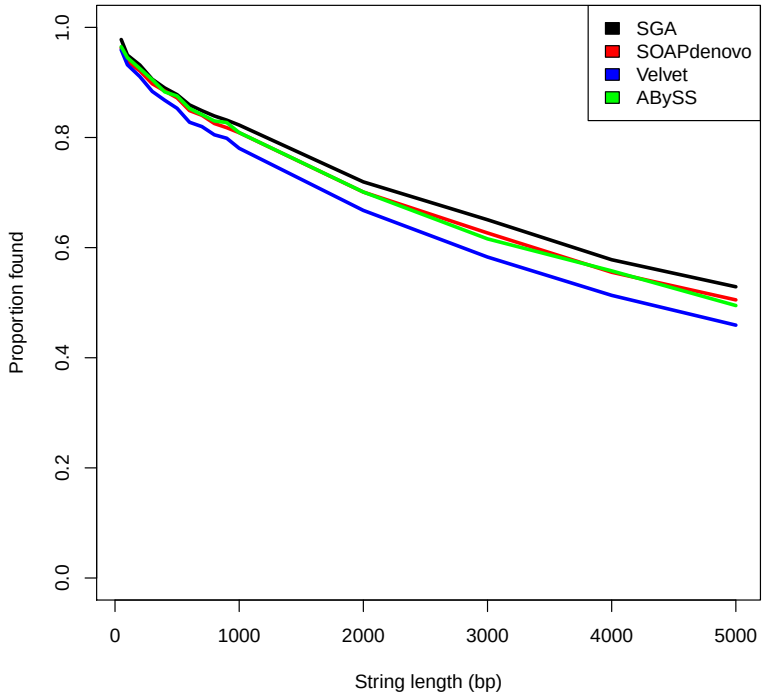

Figure 3.3: Reference string coverage analysis for the C. elegans N2 assembly. For string lengths from 50bp up to 5,000bp, 10,000 strings were sampled from the consensus-corrected C. elegans reference genome. The proportion of the strings found in the SGA, Velvet, ABySS and SOAPdenovo assemblies is plotted.

3.3.2.2 Assembly Contiguity

I assessed the contiguity of the assemblies by calculating the contig alignment length N50. By analyzing the contig alignment lengths, as opposed to the length of contigs themselves, I account for misassembled contigs that can inflate the assembly statistics. For SGA, contig alignments 16.8 kbp and greater covered

50% of the reference genome (50 Mbp). ABySS, SOAPdenovo and Velvet had contig alignment N50s of 18.4 kbp, 16.0 kbp and 13.6 kbp, respectively.

3.3.2.3 Assembly Completeness

The contigs assembled by SGA covered 95.9% of the reference genome. The ABySS assembly covered 95.6%, Velvet covered 94.5% and SOAPdenovo covered 94.8%. Figure 3.4 plots the reference genome coverage as a function of minimum contig alignment length. In this assessment ABySS generated the best assembly as it covered more of the reference genome with long contigs.

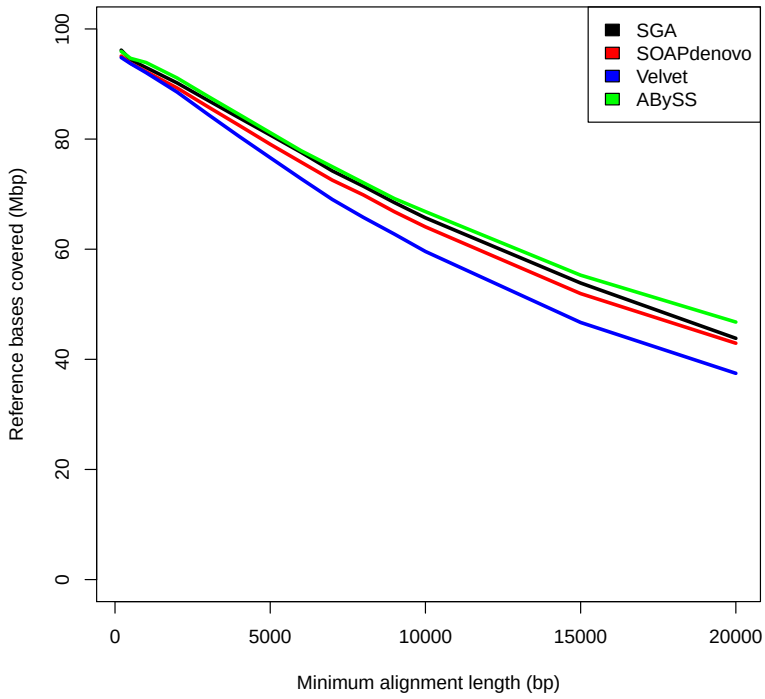

Figure 3.4: The number of bases of the C. elegans reference genome covered as a function of minimum contig alignment length.

3.3.2.4 Assembly Accuracy

I assessed both the structural accuracy and the per-base mismatch rate of the contigs. First, I categorized the contig alignments into three groups. The first group ("full-length") contains contigs that had a single alignment to the reference containing at least 95% of the contig length. The second group ("split") contained contigs that had two alignments to the same chromosome in close proximity (<10,000bp). These split contigs may either contain local assembly errors, or structural variation (for example a large insertion or deletion) with respect to the reference. All remaining alignments ("bad") were partially aligned (< 95% of the contig aligned to the reference), aligned to multiple chromosomes, aligned in greater than 2 pieces or did not align to the reference at all. For all assemblies a substantial proportion of the contigs were found to match the *E. coli* genome. As *C. elegans* eat *E. coli*, this is an expected contaminant and one might suspect other bacterial sequences to also be present. For this reason contigs that did not align to the *C. elegans* reference were not included in this analysis.

For the first measure of assembly accuracy, I counted the number and total size of contigs with split or bad alignments. The accuracy of the SGA and SOAPdenovo contigs was similar - 458 contigs for SGA (totaling 4.4 Mbp) and 483 contigs for SOAPdenovo (4.4 Mbp) had split or bad alignments. Velvet and ABySS had 787 contigs (7.2 Mbp) and 638 contigs (9.1 Mbp) with split or bad alignments, respectively.

For the second accuracy assessment, I calculated the rate at which aligned contig bases did not match the reference. In this assessment, I used the fully-aligned contigs only. I evaluated each assembly at all reference positions covered by its contigs, and also at the subset of positions that were covered by all assemblies. The latter case provides a fairer basis for comparison, removing the effect of differences of coverage of repetitive or complex sequence between the four assemblies. The results are summarized in table 3.2. Again, the accuracy of the SGA and SOAPdenovo assemblies was comparable, and both were more accurate than Velvet and ABySS. The mismatch rate of the SGA assembly at reference positions assembled by all four programs was approximately 1 mismatch per 83 kbp. SOAPdenovo, Velvet and ABySS had error rates at shared positions of 1

per 81 kbp, 1 per 18 kbp and 1 per 8 kbp, respectively.

3.3.2.5 Computational Requirements

Of the four assemblers, SGA used the least memory (4.5 GB vs 14.1 GB, 23.0 GB and 38.8 GB for ABySS, Velvet and SOAPdenovo, respectively). The de Bruijn graph assemblers were considerably more computationally efficient however as the SGA assembly required 8 times more CPU hours than ABySS, 20 times more than Velvet and 3 times more than SOAPdenovo. This speed difference is largely due to the time required to build the FM-index. However, we can reuse one FM-index for multiple runs of SGA, for instance to try different error correction or assembly parameters, whereas the de Bruijn table for ABySS, Velvet and SOAPdenovo must be re-calculated for each choice of k.

3.3.3 Human Genome Assembly

As a second demonstration, I assessed the ability of SGA to scale to very large data sets by assembling a human genome. I downloaded 2.5 billion reads (252 Gbp of sequence) for a member of the CEU HapMap population (identifier NA12878) sequenced by the Broad Institute[1]. The reads are 101bp in length from a paired-end insert library of 380 bp mean separation. As the total sequence depth is 84x, I chose to only assemble half the data to reflect typical coverage depths seen for human shotgun sequence data sets.

I constructed an FM-index for subsets of 20 million reads at a time (using the `sga-sais` variant of our indexing algorithm), then iteratively merged the sub-indices in pairs to obtain a single FM-index for the entire data set. I ran the error correction process using a cluster of computers. Each process used the full FM-index to correct 20 million reads. An FM-index was constructed for the corrected reads, duplicate and low-quality reads were removed, and non-branching chains of reads were merged together. A string graph was constructed from the merged sequences using a minimum overlap parameter $\tau = 77$. I re-aligned the reads to the resulting contig set using bwa [Li and Durbin, 2009] and constructed

[1] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20101201_cg_NA12878/NA12878.hiseq.wgs.bwa.raw.bam

scaffolds.

In total, the assembly took 1,427 CPU hours across 140 wall clock hours, just under 6 days. The most compute intensive stages were error correcting the reads and building the FM-index of the corrected reads, which each required 355 CPU hours. However these stages were distributed across a cluster of computers by simply splitting the input data, substantially reducing the elapsed (wall clock) time. I ran 123 indexing/merging processes and 63 correction processes; the elapsed time for these stages was 32 hours and 1 hour, respectively. The post correction read filtering stage - where duplicate and low quality reads are discarded - was the memory high-water mark, requiring 54 GB of memory. Complete details of the number of processes, running time and memory usage for each stage of the assembly can be found in table 3.3.

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

Table 3.3: Running time and memory summary for the SGA human genome assembly

I also assembled the data with SOAPdenovo [Li et al., 2010c]. I first error corrected the reads using the SOAPdenovo error correction tool then performed three assemblies, with k-mer sizes 55, 61 and 67. The 61-mer assembly had the largest scaffold and contig N50 and was used for the subsequent analysis. The 61-mer SOAPdenovo assembly (including error correction) required 479 CPU hours across 121 wall clock hours. The maximum amount of memory used was 118 GB. As with the *C. elegans* assembly described above, I did not use the SOAPdenovo GapCloser.

I evaluated the assemblies in terms of contiguity, completeness and accuracy. Note that unlike for the *C. elegans* assembly, in this case the sequenced sample

differs from the reference genome. As in the *C. elegans* analysis, I broke the assembled scaffolds into their constituent contigs, filtered out contigs less than 200bp in length then aligned the remaining contigs to the human reference genome (build GRC 37) using bwa-sw [Li and Durbin, 2010].

The SGA contig alignments cover 2.69 Gbp of the human reference autosomes and chromosome X (95.0% of the non-N portions of these chromosomes). The SOAPdenovo contigs cover 2.65 Gbp of the human reference (93.6%). The SGA contig alignment N50 is 9.4 kbp and the SOAPdenovo contig alignment N50 is 6.6 kbp. The corresponding raw contig N50s are 9.9 kbp and 7.2 kbp. Figure 3.5 plots the amount of the reference genome covered by each assembly as a function of the minimum contig alignment length. Across all contig alignment lengths, the SGA assembly covered more of the reference genome than SOAPdenovo. In contrast, SOAPdenovo gave larger scaffolds (N50 length of 34.8 kbp compared to 25.1 kbp for SGA), though the single short insert library for this data set limits the ability to build larger scaffolds.

The overall assembly accuracy for both SGA and SOAPdenovo was high; 94.5% of SGA contigs (totaling 2.64 Gbp) had full-length alignments to the reference genome, 1.1% (68 Mbp) had split alignments and 4.3% (50 Mbp) had low-quality alignments or did not align at all. 96.8% of the SOAPdenovo contigs had a full-length alignment to the reference (totaling 2.60 Gbp), 1.0% had split alignments (53 Mbp) and 2.2% (33 Mbp) had low-quality alignments or did not align to the reference at all. This is consistent with the SGA assembly being a little larger, covering a little more of the reference but also containing a little more additional material.

I also calculated the per-base mismatch rate of the contigs using the same methodology as the *C. elegans* assembly. In this case, I used the human reference genome combined with SNP calls produced by the Broad Institute in the same individual from the same data set by a mapping rather than assembly based approach [DePristo et al., 2011]. I only counted mismatches at positions that did not match the reference and did not match a Broad SNP call. I also calculated the mismatch rate at the subset of positions assembled by both SGA and SOAPdenovo. As both SNP calling and assembly can be confused by genomic repeats and segmental duplications, I also calculated the per-base accuracy at positions

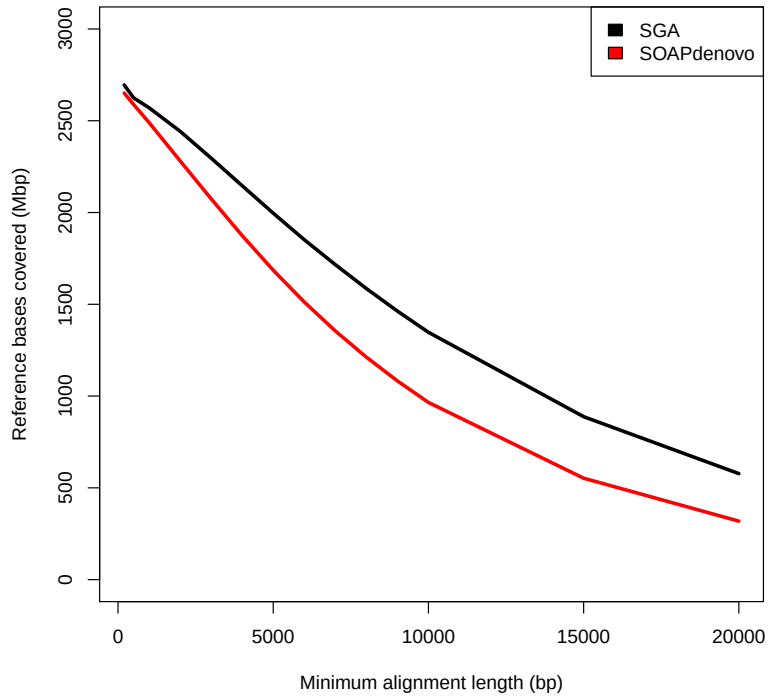

Figure 3.5: The amount of the human reference genome covered by a contig as a function of the minimum contig alignment length. For each length L on the x-axis, contig alignments less than L bp in length were filtered out and the amount of the reference genome covered by the remaining alignments was calculated.

of the reference that are not masked by RepeatMasker[1] and not annotated as segmental duplications (1.3 Gbp of the reference genome remains after this filter). Both assemblies were highly accurate. The mismatch rate for SGA over all covered positions of the reference was 1 per 3,574 bp. For SOAPdenovo, the mismatch rate was 1 per 4,285 bp. If I only consider reference positions covered by a contig from both assemblies, the mismatch rates are 1 in 4,325 bp for SGA and 1 per 5,041 bp for SOAPdenovo. When restricting the analysis to positions not masked by RepeatMasker and not annotated as segmental duplications, the mismatch rate is 1 per 52,464 bp for SGA and 1 per 51,125 for SOAPdenovo. At

[1] http://www.repeatmasker.org

positions assembled by both programs and not masked as repeats or segmental duplications, the mismatch rates are 1 per 59,884 bp and 1 per 60,511 bp, for SGA and SOAPdenovo, respectively.

I note that both the contig mismatches and the mapping-based SNP calls will contain false-positive variants due to mapping errors between the contig or read sequences and the reference. These false positives will have an opposing effect; if the contig sequence is misaligned to the reference, we may count a mismatch in the assembly that is not truly present. This will cause the error rate in the assembly to be overestimated. It is also possible that false positives from misalignments in the mapping-based call set may overlap errors in the assembly. This would lead to an underestimate of the assembly error rate. As I cannot assess the magnitude of these effects it is difficult to accurately estimate the true base-level error rate in the assemblies. However, if we conservatively consider all remaining mismatches to be assembly errors it would indicate the per-base accuracy of the SGA and SOAPdenovo assemblies are very similar and better than 1 error in 50 kbp in non-repetitive regions. The accuracy of SGA is supported by an independent assessment of our assemblers performed during the Assemblathon competition, which is described in the next section.

3.3.4 The Assemblathon

In 2010, a community organized project was launched with the goal of providing a simulated data set to benchmark and evaluate assembly software. This project was organized by UC Davis and UC Santa Cruz. They simulated a diploid genome derived from human chromosome 13. The organizers sampled simulated sequence reads from this genome from both short insert (200-300bp paired end separation) and long insert (3kb and 10kb) libraries. With the goal of modelling real sequence data, the organizers introduced base-calling errors, PCR duplications and bacterial contamination [Earl et al., 2011]. The sequence reads were openly released to the community and the developers of assembly software were invited to submit assemblies of the data. The organizers performed the analysis of the submitted assemblies, providing an unbiased comparison of assemblers on simulated data. I entered SGA into the competition. In the assessment, SGA had the largest scaf-

fold path NG50 (a measure of scaffold length, corrected for assembly errors), the lowest number of substitution errors, and the second lowest number of structural errors [Earl et al., 2011], highlighting the accuracy of my software. Overall, SGA placed 3rd out of 17 groups, behind ALLPATHS-LG [Gnerre et al., 2011] and SOAPdenovo [Li et al., 2010c].

3.3.5 *Schizosaccharomyces pombe* assemblies

As a final assessment of SGA, I assembled 104 strains of the fission yeast, *Schizosaccharomyces pombe*. These strains were sequenced as part of a project to determine the genetic diversity across the *S. pombe* population. There are two groups of strains. The first group (97 strains) had 65X sequence depth on average (range 29-91X). The second group (7 strains) were sequenced much deeper (mean 210X, range 107-436). I assembled each strain with SGA using a minimum overlap parameter (τ) of 65. Additionally for the $\geq$ 100X strains, I set a fixed threshold of 5 k-mer occurrences when running error correction. For the other strains, this parameter was automatically learned from the data.

With such a high number of samples, I am able to evaluate the impact of sequence coverage depth on contig N50, as well as run time and memory usage. The relationship between coverage and contig N50 is shown in figure 3.6. Contig N50 increases with coverage up to 100X, likely due to being able to use a longer overlap at higher coverage. Beyond 100X, adding additional coverage does not help and may actually be detrimental to assembly contiguity at very high depth (>200X). The relationship between coverage and CPU time is almost perfectly linear (figure 3.7).

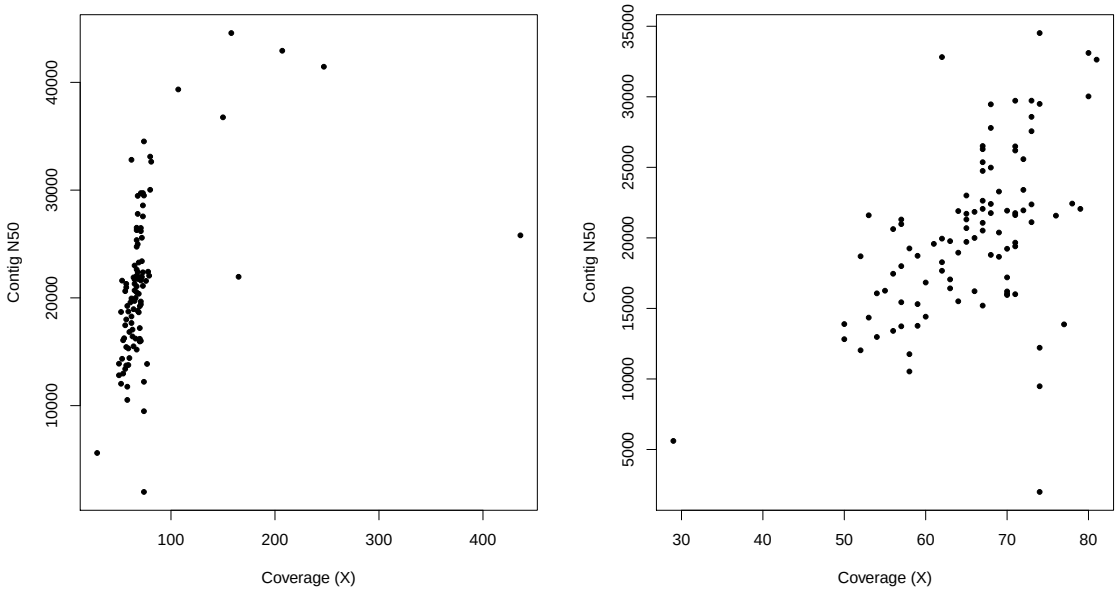

Figure 3.6: The relationship between sequence coverage and contig N50 for the S. pombe data set. The plot in the left panel displays the complete data set. The plot in the right panel only shows strains that have <100X coverage.

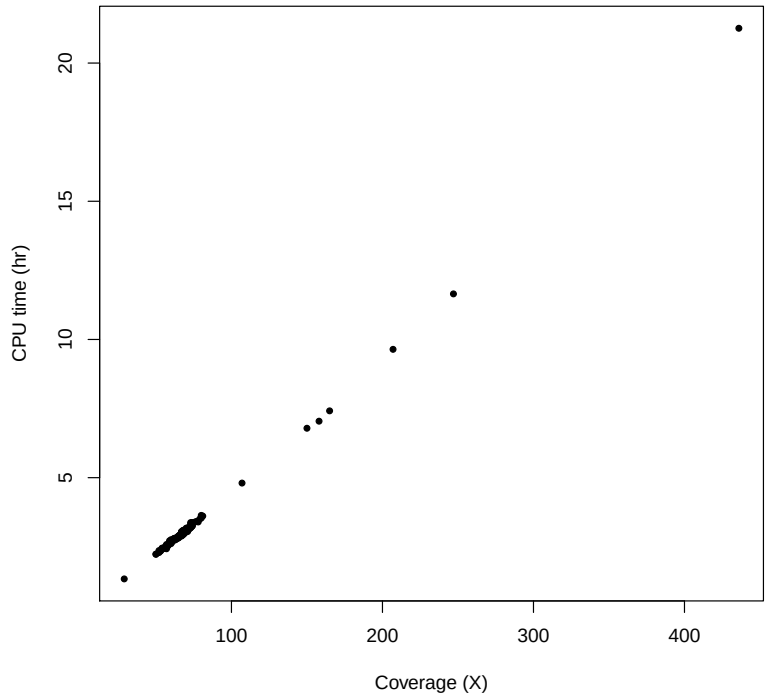

Figure 3.7: The relationship between sequence coverage and CPU time for the S. pombe data set.