Variant Genotyping with Gap Filling

Riku Walve

M. Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, November 23, 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Riku Walve

Työn nimi — Arbetets titel — Title
Variant Genotyping with Gap Filling

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M. Sc. Thesis	November 23, 2016	59

Tiivistelmä — Referat — Abstract

Although recent developments in DNA sequencing have allowed for great leaps in both the quality and quantity of genome assembly projects, *de novo* assemblies still lack the efficiency and accuracy required for studying individual genomes. Thus, efficient and accurate methods for calling and genotyping structural variations are still needed.

Structural variations are variations between genomes that are longer than a single nucleotide, i.e. they affect the structure of a genome as opposed to affecting only the content. Structural variations exist in many different types. By finding the structural variations between a donor genome and a high quality reference genome, genotyping the variations becomes the only required genome assembly step.

The hardest of the structural variations to genotype is the insertion variant, which requires assembly to genotype; genotyping the other variants require different transformations of the reference genome. The methods currently used for constructing insertion variants are fairly basic; they are mostly linked to variation calling methods and are only able to construct small insertions.

A subproblem in genome assembly, the gap filling problem, provides techniques that are very applicable to insertion genotyping. Yet there are currently no tools that take full advantage of the solution space. Gap filling takes the context and length of a missing sequence in a genome assembly and attempts to assemble the sequence.

This thesis shows how gap filling can be used to assemble the insertion variants by modeling the problem of insertion genotyping as finding a path in de Bruijn graph that has approximately the estimated length of the insertion.

ACM Computing Classification System (CCS):

Applied computing – Sequencing and genotyping technologies

Theory of computation – Graph algorithms analysis

Avainsanat — Nyckelord — Keywords
structural variations, gap filling, variation calling

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

DNA is sequenced by machines by cutting strands of DNA into short segments and reading only the segments. The sections are generally around 100 basepairs long as the probability of correctly reading segments longer than that gets impractical with current sequencing machines. The reads can be complemented by cutting the strands of DNA into longer segments and reading parts from both ends of the segment. These are called paired-end reads.

Recently, the number of reads sequenced by a single machine has been increasing fast enough that it has been becoming possible to construct *de novo* genome assemblies of different sizes. Though, for now, the dream of accurate *de novo* assemblies is not quite a reality [B+13]. Thus, for larger genomes, such as human, we need to use more efficient ways to assemble a donor genome.

Given a reference genome and reads of a donor genome, i.e. any individual, instead of having to attempt to fully construct the donor genome, it suffices to find the differences between the genomes directly from the reads. These differences, or variations, are generally split into two groups based on size. The larger variations are called structural variations.

Structural variations are often defined as any variations between two genomes that are longer than a single nucleotide. This follows from the observation that differences of single nucleotides are much easier to find from the sequenced reads by aligning the reads and ruling out noise with enough reads. In practice, insertions ands deletions of up to 30 nucleotides are reliably found by standard aligners [MHS13].

There exists a fairly large array of different approaches to structural variation finding [CWM+09, LHAB09, HAES09, YSL+09, ESW+12, RGCL14]

with all having their own sets of pros and cons. This problem has been somewhat sidestepped by biologists, by combining the results of multiple tools rather than simply deciding on one [FMP+14].

It should be noted that the problems of finding the structural variations and telling what the variations actually are, called *genotyping*, are two different problems that are often very intertwined. Most tools do their best at answering both problems, but in the case of insertions in the donor genome, the genotyping problem gets closer to genome assembly as the length of the insertion grows.

A similar problem faced in genome assembly is the *gap filling* problem. In the gap filling problem, we attempt to construct a sequence of the donor genome from the sequenced reads, such that it fills a gap of estimated length between two known sequences. The problem of genotyping insertions can be defined essentially equivalently to the gap filling problem; genotyping insertions by assembling a sequence of estimated length is gap filling with different expectations of difficulty.

Gap2Seq [SSMT15] is an implementation of an algorithm for solving the gap filling problem by reducing it to the exact path length problem, i.e. finding a path of a given length between two vertices in a graph. It manages to fill gaps better than most other tools according to their results. However, it fails to scale up to human data. The failure to scale up is largely due to the amount of reads needed to cover large, human-sized, genomes.

In this thesis, we show that using read pair information of the reads we can filter the reads down to a useful subset for a single gap and that we can use the resulting workflow to efficiently genotype insertions that were previously impossible.

2 Preliminaries

Strings are sequences $S = s_1 s_2 \dots s_n$ of symbols s_i from an alphabet $\Sigma = \{c_1, c_2, \dots c_\sigma\}$, where $\sigma = |\Sigma|$. The alphabet is assumed to be ordered, i.e. $c_1 < c_2 < \dots < c_\sigma$. Though the alphabet can technically be anything that can be ordered, within the context of DNA strings it comprises of the symbols for the four nucleotides, $\Sigma = \{A, C, G, T\}$. We will also use the notation $c + 1$ and $c - 1$ to mean the next character larger and smaller than c in the alphabet respectively.

We will use 1-indexed strings throughout, i.e. they start at position 1. The notation $S[i..j]$ is used to mean the substring of S that starts from i and ends in j, i.e. $S[i..j] = s_i s_{i+1} \dots s_j$. We will call substrings that starts from the beginning of the string $S[1..i]$, prefixes and substrings that end at the end of the string $S[j..n]$, suffixes. Strings can also be decomposed into overlapping *k-mers*. k-mers are substrings of length k.

For example, the suffixes of ATGCATGC are,

A T G C A T G C
T G C A T G C
G C A T G C
C A T G C
A T G C
T G C
G C
C

.

As the alphabet is ordered, we can compare any two strings S and T and thus even order a set of strings. Sorting strings into the order of the alphabet is called sorting into *lexicographic* order. String S is said to be smaller than T if and only if, either S is a prefix of T, or at the first position i where the two differ $s_i < t_i$. A set of strings $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ is ordered if for all strings $S_i < S_{i+1}$.

For example, the sorted set of suffixes of ATGCATGC is,

A T G C
A T G C A T G C
C
C A T G C
G C
G C A T G C
T G C
T G C A T G C .

2.1 Burrows-Wheeler transform

The Burrows-Wheeler transform was originally defined with sorted cyclic permutations of a string [BW94]. A cyclic permutation is a permutation of a string where the first symbol is moved to the last position. Sorting all the n cyclic permutations of S and taking the last character of every permutation gives us the Burrows-Wheeler transform of S.

Since we have to sort the n cyclic permutations of length n each, doing this naively would take $O(n^2 \log n)$ time. We can also define the transform using a *suffix array* of a string.

Suffix arrays are used as the basis for many tasks involving string processing. As such, they are very well studied and there exist multiple linear time construction algorithms for them [PST07]. Entries in a suffix array $\texttt{SA}_S[1..n]$ define the starting positions of the lexicographically sorted set of suffixes of S.

Definition 1. Let $S = s_1 s_2 \cdots s_n$ be a string. The i-th element in the suffix array $\texttt{SA}_S[i] = j$ corresponds to the lexicographically i-th suffix $S[j..n]$.

When adding a special character \$ to the end of the string, sorting all the cyclic permutations reduces down to sorting the suffixes of the string. Taking the last symbol from the permutations translates to taking the

symbol preceding the start of the suffix. More formally, we have the following definition.

Definition 2. Let $S = s_1 s_2 \cdots s_n$ be a string, such that $s_i \in \Sigma$ for all $1 \leq i < n$ and $s_n = \$$ where $\$ < s$ for all $s \in \Sigma$. The Burrows-Wheeler transform of S is then,

$$\texttt{BWT}_S[i] = \begin{cases} S[\texttt{SA}_S[i] - 1], & \text{if } \texttt{SA}_S[i] > 1 \\ S[n], & \text{otherwise} \end{cases}$$

As only a single linear pass over the suffix array, which takes $O(n)$ time to construct, is needed, this takes only $O(n)$ time in total.

For example, the Burrows-Wheeler transform of $T =$ ATGCATGC\$ is,

$\texttt{SA}_T$	$\texttt{BWT}_T$	$T[\texttt{SA}_T[i]..]$
9	C	\$
5	C	ATGC\$
1	\$	ATGCATGC\$
8	G	C\$
4	G	CATGC\$
7	T	GC\$
3	T	GCATGC\$
6	A	TGC\$
2	A	TGCATGC\$.

2.2 BWT-index

As string ordering is solved at the string differences, if $X \leq Y$ lexicographically then $cX \leq cY$ for any character $c \in \Sigma$. If the i-th character in the Burrows-Wheeler transform is $\texttt{BWT}[i] = c$ is the j-th occurrence of c in BWT, then cX is j-th suffix starting with c and X is the suffix starting at position $\texttt{SA}[i]$.

Finding the range $[s..e]$ of suffixes that start with c_1 consists of finding the first and last suffixes that start with c_1. Finding the first suffix is done by finding the last suffix that starts with a character smaller than c_1.

This could be done by counting the number of occurrences of characters smaller than c_1 in the text, but we can precompute the number for all characters in an array C. Equivalently we can define $C[c]$ to be the sum of frequencies over the set of characters $\{c_1c_2 \cdots c-1\}$ smaller than c. Note that we will use $C[c_1] = 0$ and $C[c_{\sigma+1}] = n$

Now the first suffix starting with c_1 is at position $s = C[c_1] + 1$ in `BWT`. The last suffix starting with c_1 is at position $e = C[c_1 + 1] - 1$. The range of suffixes that start with c_1 is thus $[s..e]$.

Finding the range of suffixes that start with $P = c_2c_1$ then means finding the positions of the first and last suffixes starting with c_2. The range is then offset from both sides by the number of suffixes that are smaller or larger, respectively, than P.

As all the suffixes smaller than c_1X have to be before s in `BWT`, we only need to find suffixes smaller than c_2c_1X in $\texttt{BWT}[1..s-1]$. We do this by counting the number of occurrences of c_2 in $\texttt{BWT}[1..s-1]$.

We will use the notation $rank_c(\texttt{BWT}, i)$ to mean the *rank* of c up to position i in `BWT`, i.e. the number of occurrences of c in $\texttt{BWT}[1..i]$. The rank queries can be answered in $O(\log \sigma)$ time when the Burrows-Wheeler transform is stored in a *wavelet tree* structure [MBCT15].

We can thus update the range $[s..e]$ of suffixes starting with c_1 to the range of suffixes starting with P by saying $s = C[c] + rank_c(\texttt{BWT}, s-1) + 1$ and $e = C[c] + rank_c(\texttt{BWT}, e)$. The range of suffixes that start with P is then the range $[s..e]$. With this information we can say that the number of occurrences of the pattern P is $e-s+1$. This is also described in Algorithm 1.

This technique of finding occurrences of a pattern $P = c_1c_2 \cdots c_m$ in `BWT` is called *backward search* and is named for the fact that the pattern needs to

be searched right-to-left. The technique is the basis for a lot of the tricks that are used with Burrows-Wheeler transforms as we will see in the next section. We will also use the *BWT-index* structure to mean we are given the transform `BWT` and the array C.

Input : Burrows-Wheeler transform `BWT` of text $S = s_1 s_2 \cdots s_n$, count array $C[0..\sigma]$, and a pattern $P = p_1 p_2 \cdots p_m$
Output : The number of occurrences of pattern P in text T

```
i ← m;
(s, e) ← (1, n);
while s ≤ e and i ≥ 1 do
    c ← p_i;
    s ← C[c] + rank_c(BWT, s − 1) + 1;
    e ← C[c] + rank_c(BWT, e);
    i ← i − 1;
end
if e < s then
    return 0;
else
    return e − s + 1;
end
```

Algorithm 1: Backward search for a pattern with a BWT-index

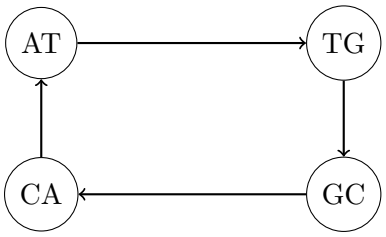
3 De Bruijn graphs

DNA cannot be sequenced in its entirety with a single read, as sequencers can only read short sections while maintaining accuracy. Further, splitting the DNA into short sections using a chemical process makes the splits at random positions. This makes it difficult to tell where any single short section came from. The well-studied problem of assembling the short sequences into the full genome is called *genome assembly.*

Genome assembly is usually abstracted as the problem of reconstructing a string from a set of its k-mers [IW95]. This is often done by way of *de Bruijn graphs.* They are directed graphs $G = (V, E)$ where vertices $v \in V$

correspond to k-mers present in the set of reads R and edges $(v, v') \in E$ correspond to observed $(k+1)$-mers in the reads starting with v and ending with v'.

For example, the de Bruijn graph of $T = ATGCATGC$, and $k = 2$ is



Note that this definition is strictly speaking a subset of a traditional de Bruijn graph; de Bruijn graph as used in other contexts refers to a graph containing all possible k-mers. In genome assembly it makes sense to use this subset of the graph to represent a given genome; not only is it more efficient to work with the subgraph, it also makes solutions to problems specific to the genome.

In practice, choosing a value of k is an act of careful balance; a large value of k creates an untangled graph and reduces repeat collapsing, while a small k avoids fragmentation of the graph. Some genome assembly algorithms attempt to tackle this by using multiple different values of k [PLYC10].

Recently, there has been work on a few different approaches on generalizing de Bruijn graphs around its parameter of k [BBG$^+$15, LP14]. Mainly the *variable-order* de Bruijn graphs, which allow for changing k on the fly, have been shown to be practical to construct, but traversing them requires further restrictions.

3.1 Bloom filter-based representations

The *Bloom filter* [KM06] is a space-efficient hash-based data structure, designed to test whether an element is in a set. It consists of an array of m bits, initialized as zeroes, and h hash functions. In order to insert an element into

a Bloom filter, h positions in the array are computed with the hash functions and all positions are set to 1. Testing whether an element is in a set with the Bloom filter is done by testing whether all of the h corresponding positions in the array are set to 1.

Using Bloom filters is clearly faster than going through an entire set of elements to test whether an element is in the set, as the number of elements in the set n usually dominates over the number of hash functions h. The main drawback is the fact that Bloom filters can give false positives, i.e. saying an element is in the set when it is not, when encountering collisions with the hash functions.

Considering hash functions that yield equally likely positions in the bit array, the false positive rate $\mathcal{F}$ is [CR13]

$$\mathcal{F} \approx \left(1 - e^{-h\frac{n}{m}}\right)^h.$$

With a fixed bits per element ratio $r = \frac{n}{m}$, minimizing the equation gives the optimal number of hash functions $h \approx 0.7r$, for which $\mathcal{F} \approx 0.6185^r$. Assuming the optimal h and solving the equation for m gives an optimal array size of $m \approx 1.44 \log_2(\frac{1}{\mathcal{F}})n$.

We will now look at two different approaches to representing a de Bruijn graph with a Bloom filter. The first is a simpler method of inserting k-mers in the text to a Bloom filter and querying for implicit overlaps of the k-mers. The second one removes false positives from the overlap querying by building a structure to hold all false positives overlaps of the k-mers.

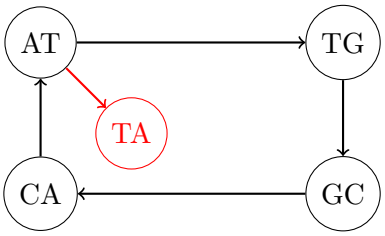
3.1.1 Probabilistic de Bruijn graph

Constructing a probabilistic de Bruijn graph [PHCK$^+$12] consists of inserting all k-mers into a Bloom filter. Edges (u, v) between k-mers are then deduced by querying the Bloom filter for the k-mer v that is prefixed by the $k-1$ length suffix of u.

Note that as the reads contain sequencing errors at random positions, we cannot guarantee all k-mers from the reads are genomic, i.e. exist in the genome. A commonly used strategy to reduce the number of incorrect k-mers in the de Bruijn graph is to count the occurrences of each k-mer and only include ones that have a frequency above some threshold. The reasoning is that as the read coverage grows the frequency of genomic k-mers grows.

Querying for (in-/out-) neighbors of any one k-mer can be done by querying for all σ possible neighboring k-mers. This clearly takes $O(\sigma h)$ time, thus does not theoretically slow down with growing graph sizes. This is also a space-efficient representation of a de Bruijn graph, as it only requires m bits of space, where m is a pre-determined size for a Bloom filter as previously discussed.

For example, the 2-mers for $T = ATGCATGC$, a single simple hash function, and a small array size $m = 8$ give the following probabilistic de Bruijn graph,



$c_1 \dots c_k$	$\sum_{i=1}^{k} c_i^i \mod 8$
AT	5
TG	4
GC	3
CA	2
TA	5

Even with this trivial example, the problem of over-approximation of the graph becomes noticeable. The Bloom filter allows for following an edge from AT to TA which does not exist in the graph. Handling the false positives in a succinct manner is thus critical for using a Bloom filter-based de Bruijn graph.

3.1.2 Exact de Bruijn graph

Removing the false positives from the probabilistic de Bruijn graph was studied in [CR13]. This is done by extending the probabilistic representation by adding another structure cFP which marks all false positive edges in the de Bruijn graph.

Querying for neighbors in the graph is modified to only return neighboring k-mers that the Bloom filter returns a true value for and that are not in the cFP structure.

The cFP structure is constructed by enumerating all possible extensions from k-mers in the graph, i.e. all the implicit edges for which the Bloom filter gives a positive answer. For each such extensions, we check if it exists in the actual graph. If an implicit edge in the graph leads to a non-existing k-mer, the edge is false positive and is added to the cFP structure. This algorithm and how it can partition the work sets to keep a low memory usage

are detailed in Algorithm 1.

Input : The set S of all vertices in the graph, the Bloom filter B constructed from S, and the maximum number M of elements in each partition
Output : The set of false positive edges cFP

$i \leftarrow 0$;
for *each k-mer* $m \in S$ **do**
  for *each extension* n *of* m **do**
    if $n \in B$ **then**
      $D_i \leftarrow D_i \cup \{n\}$;
    end
  end
end
while *end of* S *is not reached* **do**
  $P_i \leftarrow \emptyset$;
  while $|P_i| < M$ **do**
    $P_i \leftarrow P_i \cup \{$ next k-mer in $S\}$;
  end
  for *each k-mer* $m \in D_i$ **do**
    if $m \notin P_i$ **then**
      $D_{i+1} \leftarrow D_{i+1} \cup \{m\}$;
    end
  end
  delete D_i, P_i;
  $i \leftarrow i + 1$;
end
$cFP \leftarrow D_i$;
return cFP;

Algorithm 1: cFP structure construction. The memory usage is entirely dependent on the number of elements allowed for each partition, as the Bloom filter B can be freed after finding all extensions for which B answers *yes*.

3.2 BWT-index-based representations

Using the previously described BWT-index, we can not only store de Bruijn graphs efficiently, we can also efficiently construct the additional required structures to represent the graphs.

In this subsection we will look at two different BWT-index-based approaches to representing the de Bruijn graphs. The first one is a straightforward structure that adds a bit vector to the BWT-index to mark repeated runs of k-mers in the transform. The second approach, while actually unrelated, can be seen as an optimization over the first as it collapses all the k-length repeats in the suffixes to more compactly store the de Bruijn graph.

3.2.1 Frequency-aware de Bruijn graph

Presented originally as a *Compressed Gk array* by Välimäki and Rivals [VR13] the structure was later shown to be functionally usable as a de Bruijn graph [MBCT15].

The structure exploits the fact that when we are sorting suffixes, we are implicitly also sorting all the prefixes of the suffixes. When we consider all the k-mers in the text to be the k length prefixes that are common between the suffixes where the k-mers start, all the unique k-mers correspond to intervals in the suffix array and thus the Burrows-Wheeler transform.

We simply mark all the starting positions of the intervals in a bit vector $first[1..n]$, where n is the length of the Burrows-Wheeler transform. There is no need to mark the ending positions in the bit vector, as all intervals end only when the next interval starts. Finding the starting positions for all the k-mers is trivial to do in $O(\sigma^k)$ time; we use backward searching on all possible unique k-mers to find the interval of the transform they correspond to, and mark the starting position in the bit vector.

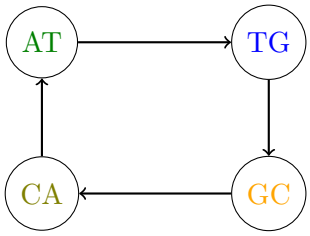
This construction time can be improved by only searching for the k-mers that exist in the text. The basic idea is that at each step of the backward searching, all the possible extensions to the current pattern are the unique characters in the search interval. With a wavelet tree representation of the Burrows-Wheeler transform, the unique character query for an interval $[i..j]$ can be answered in $O(\sigma \log \sigma)$ time by computing the ranks for all

characters at the starting and ending positions of the interval and noting that if $rank_c(\texttt{BWT}, i) \neq rank_c(\texttt{BWT}, j)$ then c appears in the interval at least once.

This shrinks the construction time to only $O(n \log \sigma)$ as all overlapping k-mers are searched for only once and the combined length of the k-mers is bounded by the length of the text. Note that the algorithm also simulates a traversal on the suffix link tree [MBCT15].

Querying for (in-/out-) neighbors of a k-mer given the corresponding interval can be done in $O(\sigma \log \sigma)$ time by taking all σ possible backward steps on the Burrows-Wheeler transform.

For example, the frequency-aware de Bruijn graph for $T = ATGCATGC$ and $k = 2$ is,



first	$\texttt{BWT}_T$	$T[\texttt{SA}_T[i]..]$
1	C	\$
1	C	ATGC\$
0	\$	ATGCATGC\$
1	G	C\$
1	G	CATGC\$
1	T	GC\$
0	T	GCATGC\$
1	A	TGC\$
0	A	TGCATGC\$.

Using the full Burrows-Wheeler transform also allows us to answer queries on the number of occurrences of each k-mer in the original text as the length of the interval is the frequency of the k-mer. These queries are less often used though, thus not supporting them also makes sense as it allows to have a more compact representation of the graph.

3.2.2 Frequency-oblivious de Bruijn graph

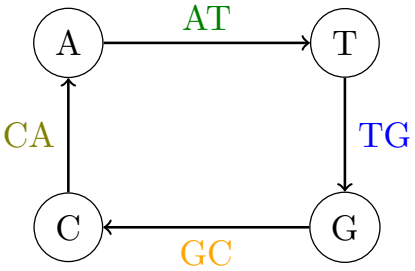
Similar to the frequency-aware de Bruijn graph, the BOSS [BOSS12] (from the authors' initials) structure is based on a structure similar to the BWT-index. Unlike a BWT-index, where we store characters preceding sorted suffixes, here we store the characters following each sorted k-mer, effectively storing the unique *edges* of the de Bruijn graph. It thus, compared to the frequency-aware representation, throws away the frequency information of the k-mers in order to save space.

Since the structure stores the edges of the graph, we need to technically consider the $(k+1)$-mers of the text during the construction to maintain similarity with the other methods. Structures that store edges are called *edge-centric* in general.

To construct the BOSS structure, we need to find all the k-mers that have a frequency above a given threshold and sort them into *co-lexicographic* order. In a co-lexicographic order, the strings are sorted in lexicographic order from right to left. Note that we can use either the Bloom filter based k-mer counting or the frequency-aware de Bruijn graph of the text to count and sort the k-mers.

For each of the sorted k-mers v, we will need to take the character following it in the text and store it in an array as $W[v]$. We will also need an bit vector $L[0..m]$ to mark all unique vertices, i.e. $L[e] = 1$ means e is the first edge from the corresponding k-mer, and a position array $F[0..\sigma]$ encoding the positions for the first vertex of ending with each character.

For example,



L	k-mer	W
1	\$\$	A
1	\$A	T
1	CA	T
1	GC	A
1	TG	C
1	AT	G

The space required by the structure is m bits for L, $\sigma \log_2 m$ bits for F and $m \log_2 \sigma$ for W, and given the DNA alphabet this equals a total of $4+o(1)$ bits for each of m edges. Thus, even though of the four representations of a de Bruijn graph, this is the most complex, it is also the most succinct.

4 Gap filling

Gap filling is the process of reconstructing the missing sequence between contiguous sections, called *contigs*, of a genome assembly that have a gap of either an estimated or an unknown length between them.

The gaps are simply sections that proved difficult for the genome assembler to assemble. The difficulty comes mainly from two things, either the section has been sequenced with a low coverage or contains too much repetitive sequences to unambiguously assemble.

Many genome assemblers, such as Allpaths-LG [GMP+11] and ABySS [SWJ+09], include a gap filling module in their pipelines. There are also standalone gap filling tools available, e.g. SOAPdenovo's GapCloser [LLX+12], GapFiller [BP12], Gap2Seq [SSMT15], MindTheGap [RGCL14] and Sealer [PWV+15].

All these tools attempt to do a *local genome assembly* with a set of reads

of the donor genome. The methods used in the tools vary somewhat between using overlaps within a subset of the reads in Allpaths-LG, using a k-mer based method in GapFiller, and the de Bruijn graph based methods used in GapCloser, Gap2Seq, MindTheGap and Sealer. As the commonly used implementations of de Bruijn graphs have improved in recent years, the graph based methods tend to be more efficient and yield better sequences.

The major issue with the de Bruijn graph based methods is how to choose a value of k for the graph. Gaps stemming from low read coverage can generally be assembled with shorter k-mers, as the graph is more likely to contain all required k-mers. Repetitive sequences are easier to fill with longer k-mers as the graph is less tangled. To get around the problem, Sealer finds paths using multiple graphs with different values of k.

In this section, we present the formal definition for the gap filling problem used by Gap2Seq, and show how it can be efficiently implemented with the succinct de Bruijn graphs previously shown. Gap2Seq is chosen here for the robustness of its problem definition. Two key reasons for which are as follows.

First, rather than directly using a de Bruijn graph to fill the gap, MindTheGap attempts to construct a graph of assembled contigs from which it finds a path over the gap. The problem of this is that the contigs should have already been considered by the genome assembly method leading up to the gap filling. Gap2Seq and Sealer both find paths in the de Bruijn graph directly instead.

Second, MindTheGap and Sealer both discard the gap length estimate information. Sealer considers all paths between the two gap-flanking sequences and builds a consensus of those. Whereas Gap2Seq finds only paths with a length close to the estimated gap length.

Although robust, the path finding method employed by Gap2Seq does not scale up to large de Bruijn graphs. We will consider using read filtering

to construct smaller de Bruijn graphs later in this thesis.

4.1 Problem definition

With a general graph of strings, the gap filling problem can be defined as finding a path from a starting string s to an ending string t, such that the length of the path is approximately the known length of the gap. The strings s and t are chosen such that they correspond to parts of the contig right before and after the gap respectively.

Definition 3. *Gap Filling problem.* Given a directed graph $G = (V, E)$, two vertices $s, t \in V$, a cost function $c : E \to \mathbb{Z}_{\geq 0}$, and an interval of path costs $[d'..d]$, find a path $P = v_1 v_2 \cdots v_n$ such that $v_1 = s$, $v_k = t$, and

$$\text{Cost}(P) = \sum_{i=1}^{k-1} c(v_1, v_{i+1}) \in [d'..d].$$

With a de Bruijn graph, we can define the cost function to be 1 when two vertices have edge, as they overlap by $k-1$ symbols, and ∞ everywhere else, as such an edge in the path is impossible. This makes the cost of the path the number of vertices in the path that are not s or t. An example of gap filling with a de Bruijn graph is given in Figure 4.

This can be solved in $O(d\,|V|)$ time with a simple dynamic programming pattern. This is a pseudo-polynomial time complexity as it depends on the cost of the path which could be arbitrarily large but as we will see later in the results, the size of the graph clearly dominates the runtime.

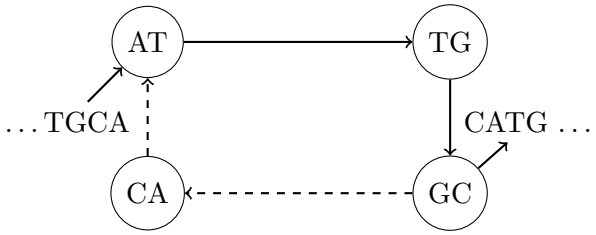


Figure 4: An example of filling a gap using a path through a de Bruijn graph. The path CA-AT-TG-GC-CA, where CA on both sides of the path are the flanks, connects the two contigs with a cost of 3.

We need to fill a matrix $M[v][i]$, where $v \in V$ and $i \in [1..d]$, such that $M[v][i]$ tells the number of paths that reach v from s, by following i edges. This can be done with a breadth-first search in G where we increment $M[v][i]$ when we reach v from any vertices v' where $M[v'][i-1] > 0$. We then trace some path back from any $M[t][i] > 0$ where $i \in [d'..d]$ and output the labels of the vertices in the path.

4.2 Space complexity

We will add an artificial limit to the number of paths to a vertex for two reasons. First, we are only interested in biologically viable sequences, thus abusing a cycle in the de Bruijn graph to produce an arbitrarily lengthy sequence is not useful. Second, a simple limit will greatly reduce the complexity of the following analysis of the space requirements.

Storing the dynamic programming matrix, M naively is going to take $d \log_2 m|V|$ bits of space, where m is the maximum number of paths to any vertex. Since the vertices in used in a path is going to be far less than the vertices in the overall graph, we know that the matrix is going to be very sparse, so we can store it in a more compact way. We can store only the non-zero rows of the matrix by associating each row with the corresponding k-mer. With the BWT-index based de Bruijn graphs the k-mers are ordered,

so we can use the k-mer ordering to associate the row to the k-mer. With the Bloom filter de Bruijn graphs, we need to use hashing.

Note that rather than storing the number of paths, we could store boolean values answering only whether we can reach a vertex v with i edges. In fact, assuming the rows are sparse as well, we could even store the booleans more compactly by storing only the indices of all 1s. This would take $dn \log_2 d$ bits of space, where n is the number of non-zero rows in the matrix. Though, now we cannot use the number of path information to filter out highly cyclic paths.

Since we are filling the matrix in a breadth-first search, we are only ever accessing either the number of paths to a vertex in $i - 1$ edges when filling the i-th column of the matrix, or whether the number of paths is non-zero when tracing back the path. Thus we only need to store the number of paths for the previous and current columns of the matrix at each step. This would get us down to $dn \log_2 d + n \log_2 m$ bits of space.

Each row in the matrix can be stored more efficiently by noting that they consist of strictly increasing integers. Thus it suffices to store only the differences between every two elements. The rows are also only being read or appended to at the end; the rows can be represented as stacks and each integer can be decoded from the differences by keeping track of the top most integer and subtracting the last difference from the top gives the previous integer.

Still, storing the differences as such does not really help as the largest possible difference is still $d - 1$ and each element in the stack still requires $\log_2(d - 1)$ bits of space. We can use *Elias gamma coding* [Eli75] to store the variable length differences as a sequence of bits. Gamma coding is done by taking the N-digit binary representation of an integer and adding $N - 1$ zeroes to the beginning. For example, $\gamma(5) = 00101$, since $5_2 = 101$.

Any gamma coded integer can then be decoded by first counting the

number of zeroes in the beginning, and taking the corresponding number of bits as the binary representation of the integer. Though, this also means that we are adding some additional overhead to the path tracing portion of the algorithm.

Now, even in the worst case, where the graph is a complete graph and thus every vertex is reached at every step, we will only need at most $|\gamma(d)| = 2\log_2(d) + 1$ bits to store each row in the matrix. The total space required by this increasing stack structure is

$$\begin{aligned}
&\sum_{v \in V} \sum_{i=1}^{|M[v]|} |\gamma\left(M[v][i] - M[v][i-1]\right)| \\
\leq &\sum_{v \in V} \sum_{i=1}^{|M[v]|} 2\log\left(M[v][i] - M[v][i-1]\right) \\
&\leq \sum_{v \in V} 2\log d + d \\
&\leq 2n\log d + dn.
\end{aligned}$$

Although this already brings the memory cost quite low, we still need to deal with the relatively high running time, especially with long paths and large graphs. There is not much that can be done about the long paths but the size of the de Bruijn graph can be dealt with using filtering.

5 Read filtering for gap filling

When solving the genome assembly problem by finding paths in de Bruijn graphs, we end up with long contigs that are connected by the unresolved gaps we were attempting to find with gap filling. The gaps are there because of uneven representation of k-mers in the graph, which is a consequence of either missing vertices (and edges) in the graph or repeats in the genome that are longer than the read length.

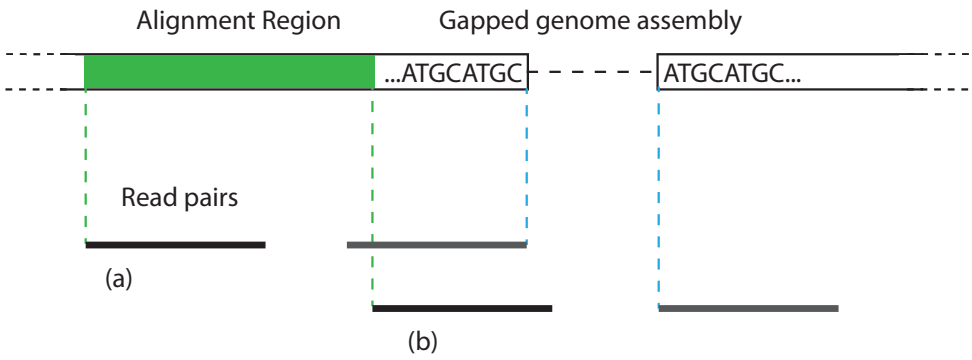


Figure 5: The alignment region defined by the minimum and maximum starting positions for gap-covering read alignments.

For filling a gap, we would intuitively want to only use the subset of reads that cover the given gap. By restricting the set of reads to those that cover the gap, we can give stricter assumptions about the distribution of the k-mers. To find all the reads that cover a region of the scaffold, we are going to use *read alignment*.

In this section we will first go over the read alignment problem and then present a method using the read alignments to find all gap-covering reads.

5.1 Read alignment

Read alignment is a well studied problem in bioinformatics [LD09]. The basic idea is to find, for a set of reads, the positions where they could have been sequenced from in a reference genome. This is greatly complicated by the fact that not only do the reads contain sequencing errors, they also were not sequenced from the reference genome. We can generally solve the problem by allowing each read alignment to have a certain *edit distance*.

The edit distance of two strings $ed(S, T)$ is defined as the minimum number of edit operations to transform S into T. One of the most common set of edit operations are the Levenshtein edit operations: insertions, deletions

and substitutions. For example, the Levenshtein distance of the strings ACTGACTG and CTCGACTGC is 3, as shown by the alignment

A	C	T	G	A	C	T	G	
-	C	T	C	A	C	T	G	C.

Aligning a single read to a reference genome can then be done by finding the substring of the genome that minimizes the edit distance. Similarly aligning a set of reads is done by minimizing all edit distances between the reads and substrings of the genome.

Definition 4. *Read Alignment problem.* Given reads R, and a reference genome $G = g_1 g_2 \cdots g_m$, find all alignment positions $\mathcal{A}(r) = (x, y)$ for reads $r \in R$ such that $\sum_{r \in R} ed(r, G[x..y])$ is minimized.

The problem has been solved in multiple ways in the past [LD09, MS13]. One way is to construct a Burrows-Wheeler transform of the reference genome and use the backward search algorithm to find exact matches for parts of the reads and attempt then extend them such that the edit distances are kept minimal. This is essentially what the commonly used Burrows-Wheeler Aligner [LD09] does.

Read pairs could also be taken into consideration here; for some estimated insert size between two reads the possible positions for alignments that minimize the edit distance to a reference genome are greatly reduced.

The reverse application for read pairs is also useful: insert sizes for read pairs can be observed by aligning all reads and calculating the distances between read pairs. In fact, the entire distribution of insert sizes for a sequencing read library can be inferred from an alignment, though some care needs to be taken when doing so.

5.2 Insert size distribution inference

The insert sizes of a read library can mostly be assumed to be distributed according to the normal distribution $\mathcal{N}(\mu, \sigma)$ [MCC+12]. The mean μ and

deviation σ that define the distribution can be inferred from the mapped alignments.

In the case of the fragmented assemblies considered here, there is a bias that has to be taken into account; the reads with longer insert sizes can be expected to span across contigs [SFA15].

By inferring the parameters for the entire distribution of insert sizes, we can estimate a lot of the properties of the read library and its insert sizes. For example, we can give more accurate estimates for insert sizes to refine the read alignment of read pairs. Or, in fact as we will see in the next subsection, we can also use the parameters to estimate the minimum and maximum insert sizes for a given percentage of the distribution.

5.3 Problem formulation

The read filtering problem can be defined formally as,

Definition 5. *Read Filtering problem.* Given an interval of a gap $[s..e]$ and read alignments $A(R)$ for reads R, find the reads $r \in R$ such that $A(r)$ overlaps with $[s..e]$.

Finding all the reads whose mate would be in a region $R = [s..e]$ can be seen as finding all the reads that map to a region that is defined symmetrically on both sides of R by the first and last possible positions a read could start from to have a mate belong to R. The regions R_{left} and R_{right} can be defined using the parameters of the original region and the combined length of both the read length ℓ and the expected insert size. The regions can be defined as,

$$R_{left} = [s - (\max + 2\ell)..e - (\min + \ell)],$$
$$R_{right} = [s + (\min + \ell)..e + (\max + \ell)],$$

where max and min are the maximum and minimum insert sizes respectively.

The values for max and min can computed from the distribution of insert sizes. For example, choosing the insert sizes to be within the 95% confidence interval of the distribution, i.e. $P(|X| \geq I) \leq 0.05$, gives us a maximum and minimum insert sizes of $\mu \pm 1.96\sigma$.

Though the use of the traditional 95% confidence interval is prevalent in statistics, in a filtering context, we want to allow the minimum and maximum insert sizes be as far apart as possible. The tradeoff here is between letting too many incorrect reads through the filter and leaving the correct reads out.

We will use a default of $\mu \pm 4\sigma$ to compute max and min. It should also be noted that the insert sizes are not necessarily even symmetrically distributed, so using a different multiplier for max and min could be useful in some cases.

Now finding all the possible reads to contribute to the gap can seen as finding all the mapped reads that overlap with R and all the mates of reads that map with either R_{left} or R_{right}.

As the sequenced reads can be assumed to be read from random positions, we can further assume that all positions should be sequenced with equal coverage. Thus we can calculate the expected coverage $C = \frac{|reads|}{|genome|}$ and expect the set of filtered reads to have the same coverage $\frac{|filteredreads|}{|region|} \approx C$. Now if the read filtering gives a coverage that is significantly smaller than expected, we are likely to be missing reads that are unmapped. All the unmapped reads can then be added to the set of filtered reads.

5.4 Implementation

Since the alignments can technically be given in any order, there are some issues with finding all the alignments that overlap a given region. With an unordered set of alignments finding all the ones that overlap with a region requires going through all the alignments and checking if they do overlap. To efficiently iterate through all the alignments, they are often first sorted by

their mapped position and then indexed into bins. This allows us to find the bins that overlap the region in time that is linear in the number of bins and then find only the reads that overlap with the region in time that is linear in the number of reads in the bins.

Finding mates of reads poses another problem because the reads are now ordered by their position rather than by their name. Assuming that there is going to be more reads to find by position, this is the better way to order the input. We can find the mates by using a pair of hash functions, the first computes the hash of a read's name and whether the read is first or second in the pair, the second one uses the opposite of the read's order in the pair. We can then iterate through the reads and find all the mates in $O(kmn)$ where k is the length of the read identifiers, m is the number of alignments and n is the number of reads that overlapped with either R_{left} or R_{right}. Since we can expect m to dominate here, the time can further be optimized by only iterating through all the reads that did not map.

As we are simply filtering out reads that would not contribute to the gap, we need not to worry about possible false positives. So a Bloom filter can be used to more efficiently find all the mates of the reads that are mapped to the regions. As the filter has a lookup time that is independent of the number of alignments are added to the set. For matching to the corresponding mates of the found reads, we can achieve a time complexity of $O(kmh)$ where k and m are as previously set and h is the number of hashing functions.

6 Structural variations

Structural variations are variations between genomes that are large enough to affect the structure of the genome. This distinction exists only to differentiate the problems of finding and genotyping the structural variations and the much easier problem of finding and genotyping single nucleotide variations.

Structural variations exist in many different, similar to the usual edit

operations insertions, we have the insertions and deletions. Due to biological reactions, there are three additional variations, inversions, which inverts a region of the genome, and translocations, which swaps two regions of the genome, and copy-number variations, which duplicates a region of the genome multiple times.

The hardest of the variations to genotype is the insertion variants. Genotyping an insertion in a donor genome can be defined similarly to the gap filling problem [MBCT15]: finding the starting and ending positions of an insert and filling the gap with something that is approximately the expected length of the insert.

There are four basic approaches to finding structural variations.

1. Insert size based
2. Split-read alignment based
3. Coverage based
4. *De novo* assembly

Insert size-based approaches [MCC$^+$12, CWM$^+$09] attempt to take the short paired-ended reads of a genome and align them to the reference and find any abnormalities in the observed and expected insert sizes of the reads. The main problem here is how to find all the alignments that support each abnormality.

Split-read alignment-based approaches [YSL$^+$09, MS13, ESW$^+$12] attempt to align reads across the insertions and deletions. This approach gives an accurate prediction of breakpoints but mostly for insertions and deletions up to around 30bp.

Coverage-based approaches aim at finding deletions and duplications by finding areas that get abnormal amount of reads mapped to them. They only work for very large deletions and duplications.

De novo assembly-based approaches to finding structural variations are barely relevant as the quality of de novo assembly is not currently sufficient for finding variations between individuals.

The basic approaches can also be combined to create *hybrid* approaches [MHS13, RZS+12, JWB12]. The power of combining the basic approaches can be seen from how the different approaches can find variations with different lengths. The optimal hybrid approach would thus be to use a split-read aligner to find short variations, insert size-based approach to find medium length variations and a coverage-based approach for long variations.

In the following subsections we describe in more detail the state-of-the-art for the two approaches to finding structural variations that are suitable for insertion calling, insert size and split-read alignment.

6.1 Maximal clique enumeration

Finding structural variations using paired-end reads and the insert sizes of the reads is a fairly standard method. Basically it entails looking for read pairs that have an insert size that deviates too much from the mean of the distribution. The main problem arises from trying to figure out which of these pairs are just noise.

This approach is usually accompanied with the assumption that the paired-end reads that do map to the reference genome represent the entirety of the distribution of insert sizes. Sahlin et al. [SFA15] showed that this assumption does not in fact hold and introduces bias. An improved null hypothesis showed promising results with simulated data.

CLEVER [MCC+12] (CLique-Enumerating Variant findER) creates a graph of read alignments and finds maximal cliques in the read alignment graph. If these cliques are large enough, there could then be a deletion or insertion. This approach is easiest to grasp by splitting it into two subproblems that the authors solved in their paper. First is the construction

of the read alignment graph and the second is the enumeration of maximal cliques in the graph.

Read alignment graphs are graphs with alignments as vertices and the edges represent two alignments that stem from the same allele. The edges are computed with rigorous statistical testing based on the expected insert sizes of the paired-ended reads. A maximal clique in a read alignment graph then means that the group of read pairs all vote for the same observed insert size.

The problem of enumerating all maximal cliques cannot be solved in polynomial time on arbitrary graphs, since there can be an exponential number of maximal cliques. Thus using a specifically engineered method for read alignment graphs is necessary for efficient computation.

6.1.1 Edge computation

The first subproblem in the clique enumeration approach is the construction of the read alignment graphs that are used later to find maximal cliques.

Definition 6. *Read Alignment Graph Construction problem.* Given alignments $\mathcal{A}(r)$ for reads $r \in \mathcal{R}$. Compute the most likely edges E between alignments in order to construct the read alignment graph $G = (A, E)$.

Let $A = (x_A, y_A)$ and $B = (x_B, y_B)$ be two alignments that have the first read end at position x_A and x_B and the second read start at position y_A and y_B respectively. The empirical insert sizes of the reads based on the alignments are defined as $I(A) = y_A - x_A$ and $I(B) = y_B - x_B$ respectively. The amount of overlap between the two alignments is defined as $O(A, B) = \min(y_A, y_B) - \max(x_A, x_B) - 1$. The mean interval length is $I(A, B) = \frac{1}{2}(I(A) + I(B))$ and difference of the mean interval length and overlap is defined as $U(A, B) = I(A, B) - O(A, B)$.

Now the statistical testing for edges can be defined. Let variable X be $\mathcal{N}(0, 1)$-distributed and μ and σ the parameters of the insert size distribution.

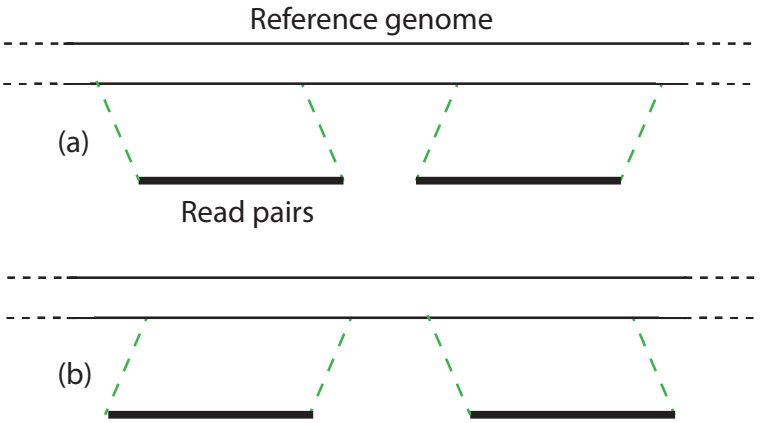


Figure 6: The two possible variants found with CLEVER. The first case is where the observed insert size is significantly shorter than expected. This indicates an insertion in the donor genome. The second symmetrically indicates a deletion in the donor genome. Figure adapted from [MCC+12].

An edge between A and B is added iff,

1. $A \neq B$,
2. $O(A, B) \geq 0$,
3. $P(|X| \geq \frac{1}{\sqrt{2}} \frac{|I(A)-I(B)|}{\sigma}) \leq 0.05$, and
4. $P(X \geq \sqrt{2} \frac{U(A,B)-\mu}{\sigma}) \leq 0.05$.

6.1.2 Enumerating maximal cliques

The second subproblem is the enumeration of maximal cliques in the now constructed read alignment graph. Cliques are sets of vertices that are adjacent to all other vertices in the clique. Maximal cliques are cliques that cannot be extended with any more vertices in the graph.

Definition 7. *Maximal Clique Enumeration problem.* Given a read alignment graph $G = (V, E)$ and the endpoints x_A, y_A of alignment intervals $A \in \mathcal{A}$. Find the maximal cliques in G.

First the endpoints of alignment intervals need to be sorted in increasing order. The endpoints are then scanned from left to right. We also need to keep a record of all the active cliques being considered at the moment, i.e. cliques that have not been extended enough to be maximal.

Assume l is the current endpoint being processed. The two possible cases for endpoints are whether they are are a left or a right endpoint of the alignment interval and they need to be handled differently.

If l is a left endpoint, the corresponding alignment A is added to the set of active cliques. If the alignment is not adjacent to any active clique, it is added as the start of a new active clique $\{A\}$. If the intersection of the open neighborhood of A and an active clique C is exactly C, i.e. if A is adjacent to at least all vertices in C, A is added to the active clique C. Otherwise if A is adjacent to some vertices V_C of a clique C, a new clique $\{A\} \cup V_C$ is added to the set of active cliques.

If instead l is a right endpoint of an alignment interval, the active cliques that the corresponding alignment A is a part of, are considered to be maximal. Now the maximal cliques can be outputted.

6.1.3 Runtime analysis

The problem of enumerating all maximal cliques in a general graph is not solvable in polynomial time on arbitrary graphs. Thus it remains to be shown that this specific method is in fact practical.

Computing the intersection of the neighborhood of the current vertex with all active cliques can be done by first iterating over all vertices in active cliques and intersecting the resulting neighborhood with each active clique by iterating over all vertices contained in the clique. This takes $O(kc)$ time,

where k is the upper bound on the local alignment coverage and c is the maximum number of active cliques.

Detecting all duplicates and cliques that are subsets of other cliques can be done by computing the intersections between all pairs of cliques that are modified according to the algorithm. All pairwise intersections can be computed in $O(kc^2)$ time.

Lastly, sorting the alignment intervals takes $O(m \log m)$ time, where m is the number of alignment intervals. So the total running time is $O(m(\log m + kc^2) + s)$, where s is the size of the output.

6.2 Split-read alignment

Another approach to finding structural variations is allowing the reads to be split when aligning them. A split-read can be thought of as being analogous to the previous approach, where we implicitly assumed the paired-end reads to split between the read-ends. This approach considers the case where reads can also be split within the reads themselves.

Definition 8. *Split-read Alignment problem.* Given a set of reads R, find reads that map to the reference with insertions or deletions added to the reads.

Allowing reads to split at any position and aligning the parts to the reference is clearly not feasible with any amount reads, as the number of possible alignments for reads grows at an exponential rate. LASER [MS13] gets around this by aligning anchors for possible split-read alignments.

LASER [MS13] (Long-indel-aware Alignment of SEquencing Reads) uses a split-read alignment approach to structural variation finding. The reads are allowed to be split in some position; if enough of the split reads align to non-contiguous positions, there could be an insertion or a deletion.

First we look for global anchors for alignments by first extracting M length prefixes and suffixes from the reads. The fragments are then mapped

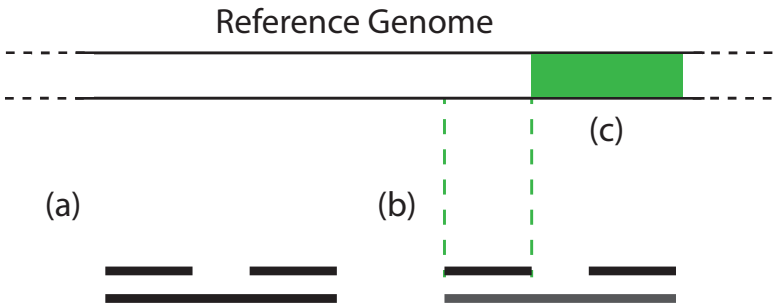


Figure 7: Overview of the method used in LASER. Figure adapted from [MS13]. First the anchors are extracted (a), then aligned to the reference (b) and finally extended to full split-read alignment (c).

to the reference sequence with any standard sequence aligner, for example BWA [LD09].

Fragments that align to more than 25 positions are considered to be as useless as ones that do not align to the reference at all. Thus we need to choose M to be large enough to not produce false positives.

Now we can look for local anchors in the areas next to the global anchors by using any standard exact pattern matching algorithm. If the global anchor is the prefix of a read, we look to the right of the anchor. If the global anchor is the suffix, we look to the left.

The anchors are then extended by adding characters from the reads until a threshold for edit distance is reached. After that, anchors that are close enough are joined to create split-read alignments.

We can further limit the number of required alignments by defining

regions of interest around the deletions we found earlier with CLEVER. LASER also estimates the probabilities $P(A)$ of alignment $A \in \mathcal{A}(r)$ being the correct one among the alignments of a read r. This can be combined with information obtained by CLEVER to further adjust the probability that the variation is true.

Now, with the accurate insertions sites, we can can finally combine the read filtering and gap filling methods to genotype the insertions.

7 Insertion genotyping

On an abstract level, genotyping an insertion variant can be seen as reconstructing the missing sequence inside the donor genome.

Definition 9. *Insertion Genotyping problem.* Given a reference genome, position p in the reference genome and length l of insertion, reconstruct the sequence $D[p..p+l]$ in the donor genome D from the reads R.

This definition is clearly compatible with the definition of the gap filling problem. The main difference is that there can be no guarantees about the difficulty of the filling process. Gaps in the assemblies were due to repeats that are difficult to reconstruct, whereas insertions can simply be random sequences added to the genome. Though they can also be repeated sequences from the genome in which case the difficulty is similar to gap filling.

An example of insertion genotyping with a de Bruijn graph is given in Figure 8.

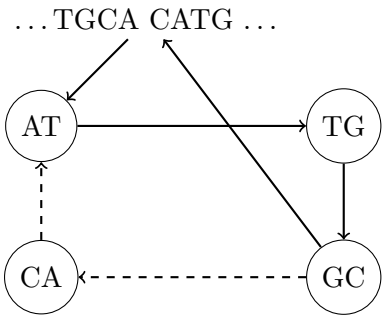


Figure 8: An example of insertion genotyping using a path through a de Bruijn graph. Note the similarity to the gap filling example in Figure 4.

Similar to gap filling in Gap2Seq, we only need the right and left flanking sequences and the length of the insertion. We can thus simulate a gap by taking the flanking sequences $D[p-k..p], D[p..p+k]$ from the reference genome.

Although overlapping variations should be taken into consideration when extracting the the sequences, doing so would require accurate knowledge of all the surrounding variations. As we are interested in specifically assembling insertion variants, we will assume no other variations overlap with the flanking sequences. Note that Gap2Seq does attempt to overcome incorrect flanks by letting the gap filling start or end on a different k-mer than the first possible, which will mitigate the problem somewhat.

8 Results

All of the following experiments follow roughly the same ideas. For both read filtering and insertion genotyping simulations, the reference genome used is chromosome 17 from the latest version (GRCH38) of the human reference genome. The Ns (ambiguous bases) in the sequence are replaced by random nucleotides, as Ns are also used to mark gaps in the sequences.

Paired-end reads are generated from the reference using dwgsim [1] with a read length of 100bp down to a coverage of 30x. All de Bruijn graphs are built with a fixed k of 31. The core gap filling software used is the reference implementation of Gap2Seq, which uses an implementation of the exact Bloom filter-based de Bruijn graphs.

The data sets and experiment set up used in the biological data experiments for gap filling with read filtering are the same as in [SSMT15]. The different genome assemblies used are taken from GAGE [SPZ+11].

The reference genome in the biological data experiments for insertion genotyping is the WS210 version of the *C. elegans* genome. The donor genome used is the *C. elegans* Hawaiian strain CB4856, more specifically the recent Illumina sequencing SRX523826.

The methodology and results are explained in more detail in the respective subsections below.

8.1 Read filtering

The filtering is evaluated experimentally by generating assemblies from the reference with gaps of varying lengths added. The reads are then mapped to the assemblies and filtered based on the alignments. The filtering is compared to a known truth by mapping the same reads to the reference genome without the gaps and taking all the reads that overlap with a given gap.

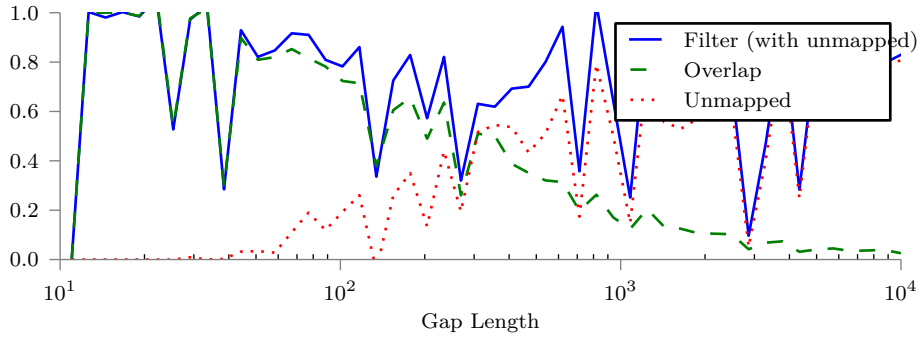
The read filtering results can be partitioned into four groups: true positives, reads correctly filtered in; false positives, reads incorrectly filtered in; true negatives, reads correctly filtered out; and false negatives, reads incorrectly filtered out. We then use the metrics *precision* and *recall* to evaluate the filtering scheme,

[1] `https://github.com/nh13/DWGSIM`

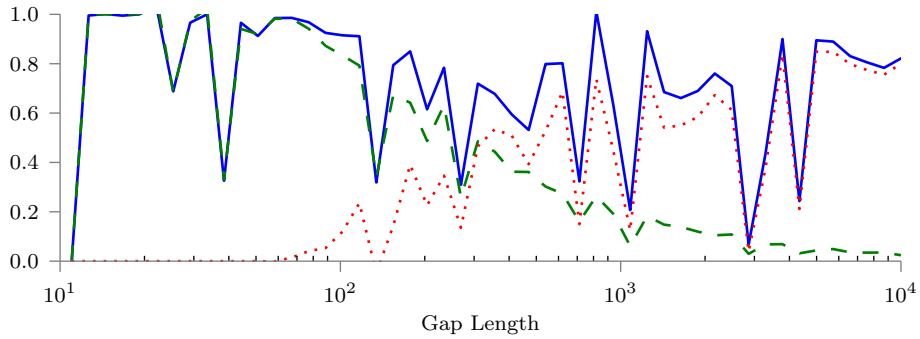
$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}},$$
$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}.$$

A fully lossless filter would have a recall of 1 since it would not be missing any of the reads. As our approach relies on read alignment, which in practice cannot be guaranteed to align all reads correctly, we also cannot guarantee a a lossless filter. Though it does give a good goal for the filter.

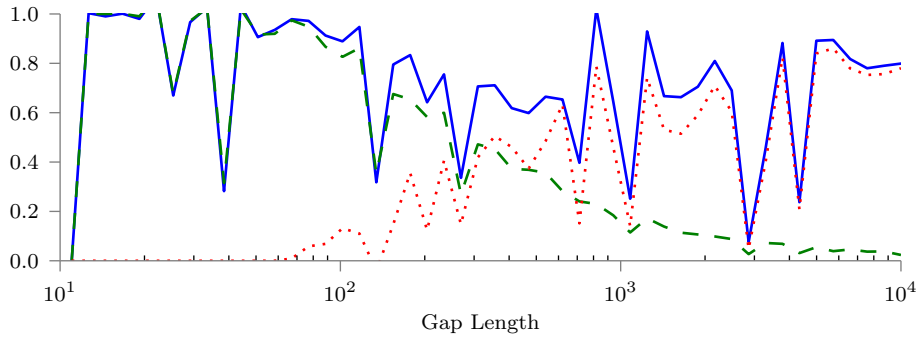
The different filtering schemes we are comparing are the simple read pileup on the gap, here called overlap; using all the reads that are unmapped, called unmapped; and filter is the use of pileups on the gap and the two regions next to the gap to get the possible overlapping mates.



(a) $\mu = 150, \sigma = 15$



(b) $\mu = 1500, \sigma = 150$



(c) $\mu = 3000, \sigma = 300$

Figure 9: Recall scores of the different read filtering schemes. The threshold for using unmapped reads in the filtering is set to 25. The parameters for the insertion size distribution simulations are given for each case.

The recall scores (see Figure 9) show that finding reads that map to a gap in the genome assembly by simply taking all the reads the read aligner placed close or over the gap works fine with small gaps. As the gaps get longer the read aligner is no longer able to align reads to the gap.

Estimating read alignments by paired-end read pairs is then useful up to a point. This method fails to find reads from a growing section in the middle of the gap when the gap length exceeds the insert size and no reads can be estimated to cover the middle of the gap.

The recall scores for different insert sizes shows that larger insert sizes generally give better filtering results than smaller insert sizes. However, increasing the mean insert size in practice also increase the standard deviation of the insert size distribution. Thus it also affects the quality of the read pair alignment position estimation.

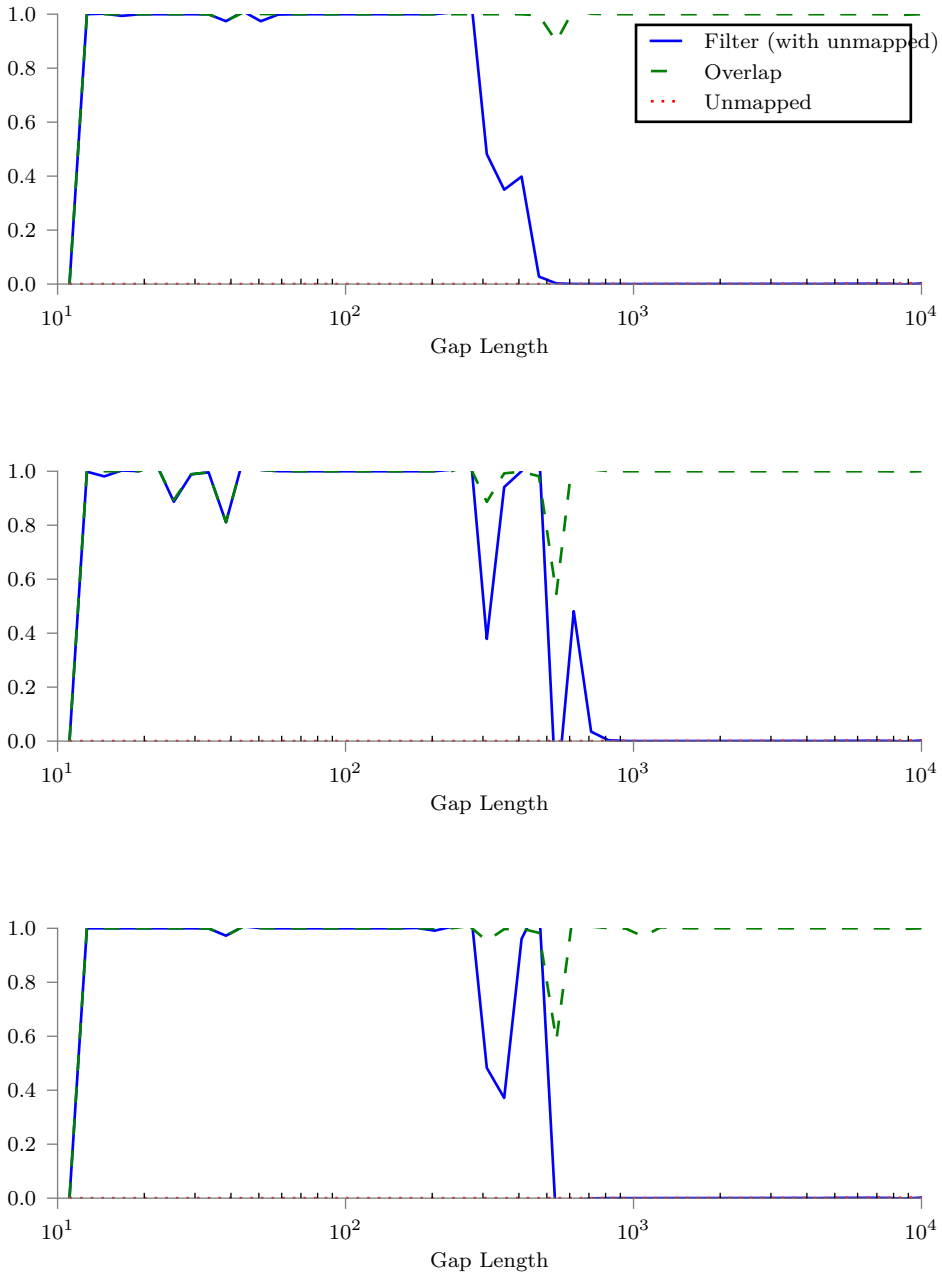


Figure 10: Precision scores of the different read filtering schemes. The formatting is identical to the recall score figures (Figure 9). Note that the precision score for unmapped is always 0.

Finally, combining the filtered reads with all unmapped reads when the number of filtered reads goes under a given threshold boosts the recall score, but at the same time it destroys the precision score for the filtering (see Figure 10). This is due to unmapped reads always giving a precision score of 0.

Finding a good threshold for using the unmapped reads means finding a balance between either having too few reads to find a useful path over the gap or having too many reads to find paths in the graph.

8.1.1 Effect on bacterial genomes

Practically all gap filling tools solve the problem of gap filling small bacterial genomes with ease. The effect of read filtering on bacterial genomes should thus be minimal.

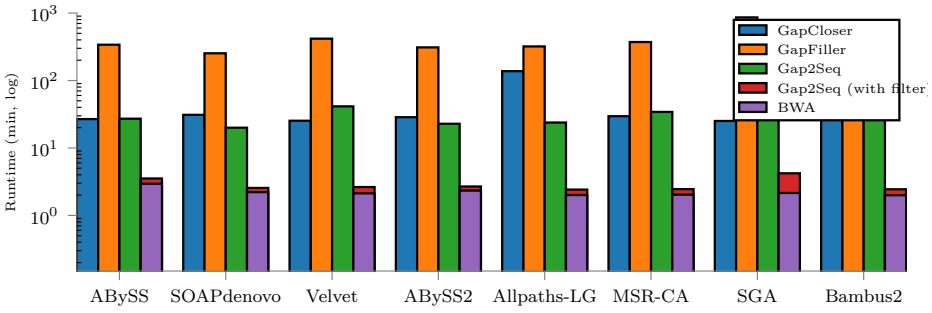


Figure 11: Running times for the different gap filling tools on different assemblies of *S. Aureus*. Time required to align the reads for filtering is labeled as BWA.

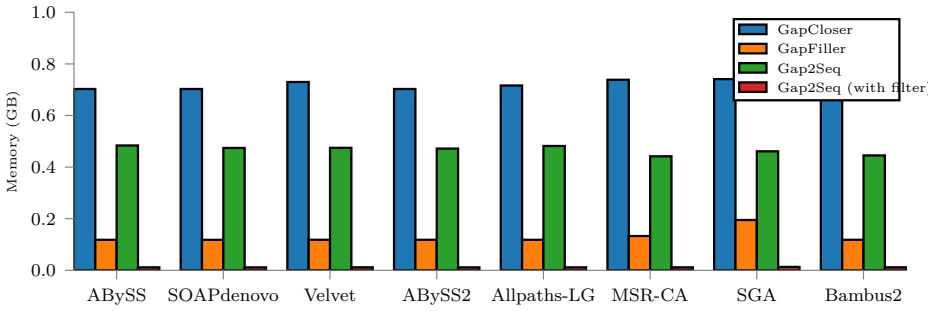


Figure 12: Maximum memory usage for the different gap filling tools on different assemblies of *S. Aureus*.

Figures 11 and 12 show how much faster and memory efficient the gap filling is when combined with read filtering. Even with the small bacterial genomes, the time and space required for filling the gaps is brought down to a surprisingly small amount.

Though, the speedup does come at a price. Table 3 shows how much the gap filling can produce erroneous sequences at worst. Tables 1 and 1 give show how well the method works at its best. However, the data set is still too small to fully benefit from the read filtering, and in fact mostly suffers from it.

The gap filling quality tables (Tables 1–6) are all split into three categories based how well Gap2Seq with read filtering performs on the assemblies. The methodology and formatting is similar to the one used in [SSMT15]; QUAST [GSVT13] is used evaluate alignments between the filled scaffolds and reference genomes. The evaluation metrics used by QUAST are as follows:

1. Misassemblies: The number of misassembled sequences in a scaffold that are larger than 4000 bp, differentiating them from simply erroneous sequences.

2. Erroneous length: Total length of all mismatches, indels and local

misassemblies.

3. Unaligned length: The total length of the unaligned sequence in an assembly.

4. NGA50: The size of the longest scaffold such that the sum of the lengths of all scaffolds longer than it is at least half of the reference genome size, after scaffolds have been broken at every misassembly position.

5. Number of gaps: The number of sites with one or more unknown position.

6. Total gap length: The total length of the gaps.

		Original	**GapCloser**	**GapFiller**	**Gap2Seq**	**Gap2Seq (with filter)**
SOAPdenovo	Misassemblies	2	+0%	+0%	+0%	+0%
	Erroneous length	35433	-1.3%	+0.7%	+1.5%	+3.1%
	Unaligned length	4055	-100%	-100%	-100%	-100%
	NGA50	69834	+0%	+0%	+0%	+0%
	Number of gaps	9	-22.2%	-33.3%	-55.6%	-100%
	Total gap length	4857	-61.4%	-34.6%	-94.2%	-100%
MSR-CA	Misassemblies	10	-30.0%	-30.0%	-20.0%	-30.0%
	Erroneous length	17276	-2.7%	+1.2%	-4.5%	+46.7%
	Unaligned length	0	+0%	+0%	+0%	+0%
	NGA50	64114	+50.3%	+20.4%	+50.3%	-7.7%
	Number of gaps	81	-51.9%	-30.9%	-56.8%	-100%
	Total gap length	10353	-75.6%	-39.4%	-70.5%	-100%

Table 1: Quality of the best filled gaps on *S. Aureus* data set. The results are shown in relative differences to the original assemblies. The best and worst scores are in highlighted by green and red text, respectively.

		Original	GapCloser	GapFiller	Gap2Seq	Gap2Seq (with filter)
Velvet	Misassemblies	25	+8.0%	+4.0%	+8.0%	+0%
	Erroneous length	24160	-32.1%	-19.4%	-36.0%	+83.1%
	Unaligned length	1270	-49.4%	-21.3%	-49.4%	-49.4%
	NGA50	46087	+19.1%	+49.0%	+73.3%	-6.6%
	Number of gaps	128	-46.9%	-40.6%	-68.8%	-98.4%
	Total gap length	17688	-59.6%	-47.9%	-81.2%	-99.9%
ABySS2	Misassemblies	5	+20.0%	+20.0%	+40.0%	+0%
	Erroneous length	10312	-4.6%	+5.0%	-27.2%	+87.2%
	Unaligned length	0	+0%	+0%	+0%	+0%
	NGA50	106796	+15.1%	+0%	+29.0%	+0%
	Number of gaps	35	-31.4%	-37.1%	-80.0%	-100%
	Total gap length	9393	-63.3%	-60.5%	-94.5%	-100%
Bambus2	Misassemblies	0	0.0%	+0%	+0%	+0%
	Erroneous length	24570	-23.0%	+13.9%	-0.7%	+98.5%
	Unaligned length	0	+0%	+0%	+0%	+0%
	NGA50	40233	+39.0%	+7.3%	+17.2%	-16.0%
	Number of gaps	99	-68.7%	-18.2%	-69.7%	-100%
	Total gap length	29205	-77.1%	-36.4%	-84.1%	-100%

Table 2: Quality of mediocre filled gaps on *S. Aureus* data set. Formatting is identical to Table 1.

		Original	GapCloser	GapFiller	Gap2Seq	Gap2Seq (with filter)
Allpaths-LG	Misassemblies	0	+0%	0.0%	+0%	+0%
	Erroneous length	5991	+1.1%	-0.3%	+10.9%	+183.0%
	Unaligned length	0	+0%	+0%	+0%	+0%
	NGA50	110168	-8.2%	+69.6%	+35.9%	-34.0%
	Number of gaps	48	-100%	-41.7%	-70.8%	-100%
	Total gap length	9900	-100%	-40.8%	-94.7%	-100%
ABySS	Misassemblies	5	+0%	+0%	+60.0%	+60.0%
	Erroneous length	10587	+27.7%	+40.5%	+72.0%	+227.7%
	Unaligned length	7935	-19.8%	-11.0%	-43.0%	-43.0%
	NGA50	31079	+0%	+0.3%	+0.3%	-5.0%
	Number of gaps	69	-17.4%	-31.9%	-87.0%	-98.6%
	Total gap length	55885	-25.2%	-25.2%	-94.5%	-94.5%
SGA	Misassemblies	2	+0%	+0%	-50.0%	+50.0%
	Erroneous length	13811	-42.7%	-31.0%	-22.2%	+2423.4%
	Unaligned length	0	+0%	+0%	+0%	+0%
	NGA50	9541	+148.1%	+9.7%	+214.6%	-52.8%
	Number of gaps	654	-74.8%	-37.5%	-80.1%	-100%
	Total gap length	300607	-53.8%	-10.3%	-72.1%	-100%

Table 3: Quality of the worst filled gaps on *S. Aureus* data set. Formatting is identical to Table 1.

8.1.2 Effect on eukaryotic genomes

The effects of read filtering should get more pronounced with a larger set of read. This is evident in the results for gap filling the much larger human14 assemblies from GAGE. Tables 4– 6 show how applying the read filtering affects the gap filling when working with large data sets.

Gap2Seq combined with read filtering is fills more gaps than regular Gap2Seq in all cases, and in some cases is actually the best available tool (the cases in Table 4). Though in most situations the filled sequence is erroneous (especially the assemblies in Table 6). It is difficult to tell whether the erroneous sequences come from the read filtering or if the read filtering only allows for the gap more efficiently and it would be filled with the wrong sequence anyway.

		Original	**GapCloser**	**GapFiller**	**Gap2Seq**	**Gap2Seq (with filter)**
Bambus2	Misassemblies	1584	+3.1%	+4.4%	+2.1%	-0.8%
	Erroneous length	11114542	-9.6%	+0.6%	-0.6%	-64.1%
	Unaligned length	161358	-42.7%	-37.8%	+2.5%	-65.6%
	NGA50	3045	+34.8%	+16.8%	+1.8%	+76.1%
	Number of gaps	11809	-16.4%	-2.4%	-6.6%	-100%
	Total gap length	10370362	-45.6%	-27.1%	-4.6%	-100%
SOAPdenovo	Misassemblies	1250	+17.1%	+6.1%	+11.7%	+5.4%
	Erroneous length	8449941	-1.3%	+3.1%	-0.5%	-7.5%
	Unaligned length	1306173	-28.8%	-27.2%	-14.1%	-90.1%
	NGA50	6592	+17.4%	+4.1%	+4.0%	+12.9%
	Number of gaps	8544	-25.2%	-5.2%	-18.8%	-99.9%
	Total gap length	10255930	-21.3%	-15.9%	-6.3%	-99.9%
Velvet	Misassemblies	9308	+26.3%	+31.1%	+13.3%	+24.9%
	Erroneous length	12531431	-10.4%	+41.1%	-0.5%	+14.0%
	Unaligned length	23484076	-58.4%	-64.5%	-20.7%	-98.4%
	NGA50	1793	+104.4%	+62.2%	+27.0%	-20.9%
	Number of gaps	51567	-43.4%	-26.0%	-26.6%	-94.5%
	Total gap length	63559964	-22.8%	-19.2%	-4.2%	-94.4%

Table 4: Quality of the best filled gaps on Human14 data set. Formatting is identical to Table 1.

		Original	GapCloser	GapFiller	Gap2Seq	Gap2Seq (with filter)
CABOG	Misassemblies	91	+16.5%	+5.5%	+7.7%	-4.4%
	Erroneous length	615239	+19.0%	-0.2%	-3.5%	+22.1%
	Unaligned length	2506	+0%	+0%	+0%	+0%
	NGA50	46665	+16.2%	+64.7%	+8.9%	+12.8%
	Number of gaps	3043	-18.9%	-51.9%	-13.8%	-99.7%
	Total gap length	231078	-50.3%	-42.0%	-22.6%	-99.9%
MSR-CA	Misassemblies	1110	+14.5%	+17.6%	+6.8%	-1.0%
	Erroneous length	5412965	+2.8%	+9.6%	-6.1%	+81.9%
	Unaligned length	318421	-30.5%	-30.6%	-13.1%	-66.4%
	NGA50	5704	+73.3%	+78.2%	+32.9%	-34.6%
	Number of gaps	30622	-34.9%	-47.8%	-27.8%	-99.8%
	Total gap length	6097928	-49.3%	-45.4%	-18.1%	-99.7%
ABySS2	Misassemblies	99	+18.2%	+3.0%	+5.1%	+23.2%
	Erroneous length	555099	+15.9%	+3.3%	+3.5%	+109.8%
	Unaligned length	157759	-21.4%	-36.4%	-15.4%	-89.2%
	NGA50	11869	+4.1%	+3.1%	+2.4%	-12.0%
	Number of gaps	2820	-14.5%	-38.5%	-23.9%	-100%
	Total gap length	949137	-34.9%	-25.3%	-36.8%	-100%

Table 5: Quality of mediocre filled gaps on Human14 data set. Formatting is identical to Table 1.

		Original	GapCloser	GapFiller	Gap2Seq	Gap2Seq (with filter)
ABySS	Misassemblies	3	+133.3%	+0%	+100%	+100.0%
	Erroneous length	190458	+18.2%	+6.1%	-9.4%	+112.1%
	Unaligned length	262068	-16.6%	-34.2%	-8.4%	-50.6%
	NGA50	1320	+1.0%	+0.7%	+1.3%	-0.8%
	Number of gaps	1061	-5.9%	-32.5%	-33.4%	-86.1%
	Total gap length	585628	-24.5%	-27.6%	-25.5%	-79.9%
Allpaths-LG	Misassemblies	95	-6.3%	+10.5%	+14.7%	+136.8%
	Erroneous length	667229	+34.5%	+6.6%	-3.0%	+273.0%
	Unaligned length	36941	-14.3%	-11.5%	+26.8%	-99.8%
	NGA50	34534	+48.3%	+22.5%	+23.3%	-14.9%
	Number of gaps	4307	-35.1%	-20.6%	-29.8%	-100%
	Total gap length	3227193	-37.9%	-17.3%	-16.0%	-100%
SGA	Misassemblies	8	+375%	+112.5%	+287.5%	+450.0%
	Erroneous length	1580489	+21.1%	-14.6%	-24.6%	+783.8%
	Unaligned length	1160159	-83.9%	-86.5%	-38.6%	-98.2%
	NGA50	2644	+244.2%	+238.0%	+149.1%	-18.4%
	Number of gaps	21459	-56.7%	-49.9%	-51.5%	-100%
	Total gap length	12840408	-53.5%	-55.4%	-30.2%	-100%

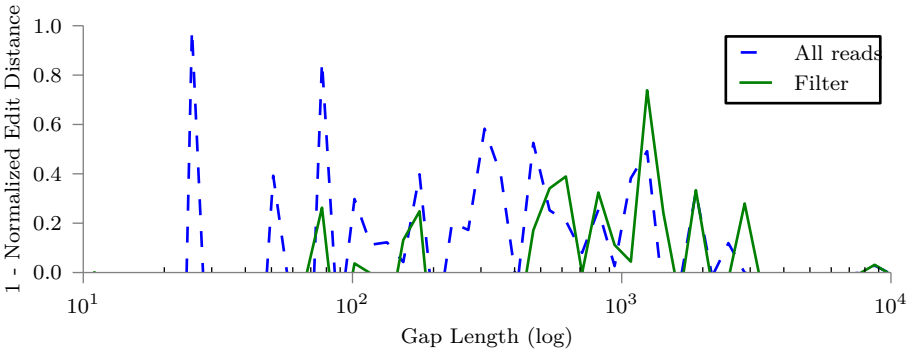
Table 6: Quality of the worst filled gaps on Human14 data set. Formatting is identical to Table 1.

8.2 Insertion genotyping

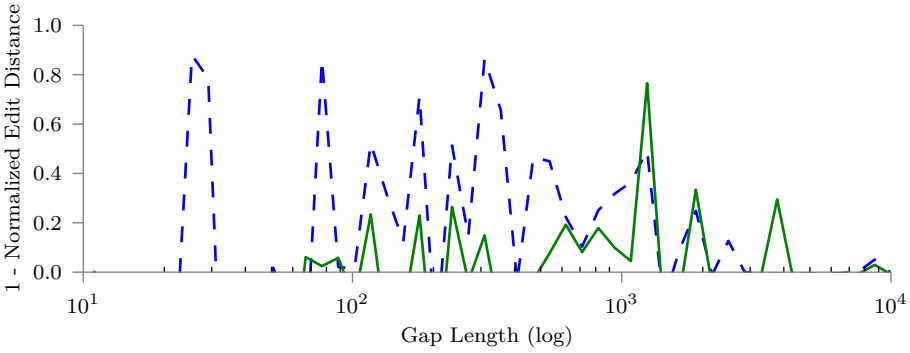
Genotyping efficacy was evaluated using both simulations and biological data.

8.2.1 Simulated data

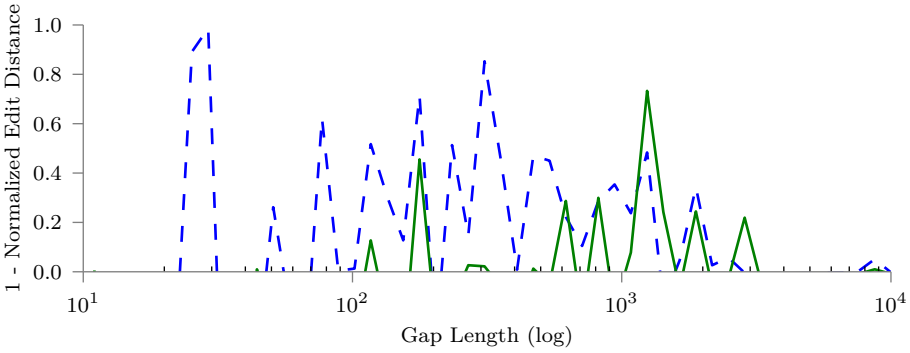
Evaluation with simulated data was done by simulating insertions of different lengths and counting the number of insertions that were filled.



(a) $\mu = 150, \sigma = 15$



(b) $\mu = 1500, \sigma = 150$



(c) $\mu = 3000, \sigma = 300$

Figure 13: 1-Normalized edit distances for different simulated gap lengths, i.e. higher score is better.

Figure 13 shows how the read filtering affects the quality of the gap filling.

The read filtering should generally be expected to perform better with read libraries that have a long insertions size. Though, the effect is not very pronounced. This could be due to the fact that the unmapped reads are added when the coverage of the reads is low.

Using all available reads is clearly better at accurately constructing smaller insertions. However, with long insertions read filtering becomes a requirement for the gap filling. Though even then, the gap filling is not perfect.

Table 7 shows Gap2Seq with and without filtering compared against MindTheGap [RGCL14], a tool for finding and constructing insertions. The positions of the simulated insertion sites are also given as input for MindTheGap, but it does not take length as input, rather it attempts to find the most likely insertion regardless of length.

Length	**MindTheGap**	**Gap2Seq**	**Gap2Seq (filter)**
0 – 100	0.585	0.586	0.483
100 – 300	0.563	0.459	0.504
300 – 500	0.642	0.522	0.637
500 – 1000	0.361	0.237	0.276
1000 – 10000	0.151	0.086	0.11
Overall	0.414	0.354	0.348

Table 7: Average 1-normalized edit distances over different length ranges for insertions constructed by different tools against the simulated gaps. The best and worst scores are in highlighted by green and red text, respectively.

While Gap2Seq is the best of the three at filling small gaps, using

read filtering improves the results for all other length ranges. However, MindTheGap still beats both tools by a slim margin.

8.2.2 Biological data

The insertions were evaluated using two complementary strategies; using experimentally validated insertions [VTGF+14], and comparing the lengths of the genotyped insertions against the lengths estimated by the insertion callers.

Using insertion sites found by Pindel [YSL+09], CLEVER [MCC+12], and LASER [MS13]. Insertion breakpoints found by MindTheGap were not used, as they do not have the corresponding length information for the insertions. However, the combined insertion sites from the aforementioned tools were separately given as input to MindTheGap.

Length	**Pindel**	**MindTheGap**	**Gap2Seq**	**Gap2Seq (filter)**
0 – 100	0.579	-0.968	0.224	0.109
100 – 300	0.018	0.321	0.31	0.376
Overall	0.361	-0.467	0.257	0.213

Table 8: 1-Normalized edit distances for insertions constructed by different tools against validated insertions. The negative scores mean the tool produced a result that was not only wrong, but also longer than the correct one. The best and worst scores are in highlighted by green and red text, respectively.

Table 8 shows how on the validated insertions, Gap2Seq performs overall better than MindTheGap, but slightly worse on insertions over 100bp long. MindTheGap discards length information when constructing the insertions and fails to get the lengths right when constructing the smaller insertions. Both Pindel and Gap2Seq utilize the length information and show much better results on the smaller insertions.

Pindel is the overall best tool in the data set, although this is mainly due to the insertions being short enough; it is only able genotype insertions up to around 200 bp.

With the read filtering scheme, Gap2Seq is the best performer at longer lengths. Though that comes with the cost of slightly worse results at short insertion lengths. Overall, at lengths only up to 300 basepairs long, it seems not worth it to run Gap2Seq with filtering.

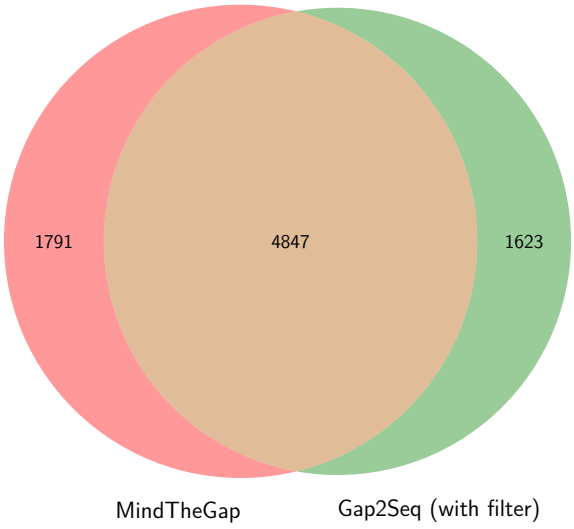


Figure 14: Venn diagram of the set of insertions genotyped by MindTheGap and Gap2Seq with filtering from the set of insertions found by CLEVER, LASER, and Pindel.

Figure 14 shows how much of the insertions genotyped are unique to MindTheGap and Gap2Seq. Both tools are basically equals in terms of the number of insertions genotyped. Pindel is not shown here, as it does not take the insertions found by CLEVER and LASER as input and it would

thus be an unfair comparison. Gap2Seq without filtering takes too much time to complete within any reasonable time and is not included.

Though Gap2Seq and MindTheGap are able to construct the same number of insertions, the quality of the genotyped insertions are hard to quantify. Table 9 shows how far the lengths of genotyped insertions are in average from the estimated insertion lengths.

Insertions	MindTheGap	Gap2Seq (with filtering)
CLEVER and LASER	1281.436	-7.738
Pindel	114.296	-46.244

Table 9: The average differences in length between the insertions genotyped by MindTheGap and Gap2Seq and the lengths estimated for the insertions by CLEVER, LASER, and Pindel.

MindTheGap tends to go very long with its genotyping, while Gap2Seq tends to go short. Gap2Seq is biased towards a smaller average difference, as Gap2Seq is allowed to construct insertions with a negative lengths, i.e. make them deletions. This happens when the best path through the de Bruijn graph is one that skips parts of the flanks.

9 Conclusions

We have proposed a method for filtering reads for gap filling and shown it to yield an unparalleled efficiency in gap filling. However, the quality of the gap filling suffers, especially with short gaps and with small sets of reads.

More importantly, we have shown that fully utilizing the solution space from gap filling gives promising results for insertion genotyping. The only other method of using gap filling techniques in insertion genotyping, MindTheGap, does not utilize the length information for the insertions and often

produces results with incorrect lengths.

Combining Gap2Seq with read filtering fills more gaps in all tested assemblies, but as the method can produce erroneous sequences, filling more gaps will inevitably lead to more erroneous filled sequences. Thus, being more conservative with gap filling is for the best. Finding ways to accurately estimate the probability of error for a filled sequence would thus be an important step towards the future.

With the read filtering unable to find reads to construct small insertions hints at a more hybrid method of using all reads for the small insertions, where the path finding will not branch as much. Such a method would require some balancing between the quality of using all reads and the performance of filtering reads, which while sounds relatively minor, is out of the scope of this thesis.

Of the space saving methods proposed in this thesis, the only one to be implemented was the read filtering method. An interesting possibility for future work would then be to experimentally validate the theoretical savings provided by using the stack-like dynamic programming matrix and the frequency-oblivious de Bruijn graph.

Using read alignment to find reads that cover a region of the reference genome is not strictly required here. We could also attempt to find approximate matches between the substring defined by a region and the parts of the reads. While this would theoretically be more efficient, we could not then also exploit the fact that when a gap is long enough the gap-covering reads are all unmapped.

References

[B+13] Keith R. Bradnam et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):1–31, 2013.

[BBG+15] Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Data Compression Conference (DCC), 2015*, pages 383–392. IEEE, 2015.

[BOSS12] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Algorithms in Bioinformatics*, pages 225–235. Springer, 2012.

[BP12] Marten Boetzer and Walter Pirovano. Toward almost closed genomes with GapFiller. *Genome Biology*, 13(6):1–9, 2012.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[CR13] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):1–9, 2013.

[CWM+09] Ken Chen, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, Devin P Locke, et al. BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods*, 6(9):677–681, 2009.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar 1975.

[ESW+12] Anne-Katrin Emde, Marcel H Schulz, David Weese, Ruping Sun, Martin Vingron, Vera M Kalscheuer, Stefan A Haas, and Knut Reinert. Detecting genomic indel variants with exact breakpoints in single-and paired-end sequencing data using SplazerS. *Bioinformatics*, 28(5):619–627, 2012.

[FMP+14] Laurent C Francioli, Androniki Menelaou, Sara L Pulit, Freerk van Dijk, et al. Whole-genome sequence variation, population structure and demographic history of the Dutch population. *Nature Genetics*, 46:818–825, 2014.

[GMP+11] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, et al. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[GSVT13] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072, 2013.

[HAES09] Fereydoun Hormozdiari, Can Alkan, Evan E Eichler, and S Cenk Sahinalp. Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes. *Genome research*, 19(7):1270–1278, 2009.

[IW95] Ramana M Idury and Michael S Waterman. A new algorithm for DNA sequence assembly. *Journal of computational biology*, 2(2):291–306, 1995.

[JWB12] Yue Jiang, Yadong Wang, and Michael Brudno. PRISM: Pair read informed split read mapping for base-pair level detection of insertion, deletion and structural variants. *Bioinformatics*, 2012.

[KM06] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[LHAB09] Seunghak Lee, Fereydoun Hormozdiari, Can Alkan, and Michael Brudno. MoDIL: detecting small indels from clone-end sequencing with mixtures of distributions. *Nature methods*, 6(7):473–474, 2009.

[LLX+12] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, et al. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 1(1):1, 2012.

[LP14] Yu Lin and Pavel A Pevzner. Manifold de Bruijn graphs. In *Algorithms in Bioinformatics*, pages 296–310. Springer, 2014.

[MBCT15] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.

[MCC+12] Tobias Marschall, Ivan G Costa, Stefan Canzar, Markus Bauer, Gunnar W Klau, Alexander Schliep, and Alexander Schönhuth. CLEVER: clique-enumerating variant finder. *Bioinformatics*, 28(22):2875–2882, 2012.

[MHS13] Tobias Marschall, Iman Hajirasouliha, and Alexander Schönhuth. MATE-CLEVER: Mendelian-inheritance-aware discovery and genotyping of midsize and long indels. *Bioinformatics*, 29:3143–3150, 2013.

[MS13] Tobias Marschall and Alexander Schönhuth. LASER: Sensitive long-indel-aware alignment of sequencing reads. *arXiv*, (1303.3520), 2013.

[PHCK+12] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[PLYC10] Yu Peng, Henry C. M. Leung, S. M. Yiu, and Francis Y. L. Chin. IDBA – a practical iterative de Bruijn graph de novo assembler. In *Research in Computational Molecular Biology*, volume 6044, pages 426–440. Springer, 2010.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[PWV+15] Daniel Paulino, René L Warren, Benjamin P Vandervalk, Anthony Raymond, Shaun D Jackman, and Inanç Birol. Sealer: a scalable gap-closing application for finishing draft genomes. *BMC Bioinformatics*, 16(1):230, 2015.

[RGCL14] Guillaume Rizk, Anaïs Gouin, Rayan Chikhi, and Claire Lemaitre. MindTheGap: integrated detection and assembly of short and long insertions. *Bioinformatics*, 30(24):3451–3457, 2014.

[RZS+12] Tobias Rausch, Thomas Zichner, Andreas Schlattl, Adrian M. Stütz, Vladimir Benes, and Jan O. Korbel. DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339, 2012.

[SFA15] Kristoffer Sahlin, Mattias Frånberg, and Lars Arvestad. Correcting bias from stochastic insert size in read pair data — applications to structural variation detection and genome assembly. *bioRxiv*, 2015.

[SPZ+11] Steven L. Salzberg, Adam M. Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J. Treangen, Michael C. Schatz, Arthur L. Delcher, Michael Roberts, Guillaume Marçais, Mihai Pop, and James A. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 2011.

[SSMT15] Leena Salmela, Kristoffer Sahlin, Veli Mäkinen, and Alexandru I. Tomescu. Gap filling as exact path length problem. In *Research in Computational Molecular Biology*, volume 9029, pages 281–292. 2015.

[SWJ+09] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[VR13] Niko Välimäki and Eric Rivals. Scalable and versatile k-mer indexing for high-throughput sequencing data. In *Bioinformatics Research and Applications*, pages 237–248. Springer, 2013.

[VTGF+14] Ismael A. Vergara, Maja Tarailo-Graovac, Christian Frech, Jun Wang, Zhaozhao Qin, Ting Zhang, Rong She, Jeffrey SC Chu, Ke Wang, and Nansheng Chen. Genome-wide variations in a natural isolate of the nematode Caenorhabditis elegans. *BMC Genomics*, 15(1):255, 2014.

[YSL+09] Kai Ye, Marcel H Schulz, Quan Long, Rolf Apweiler, and Zemin Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, 2009.