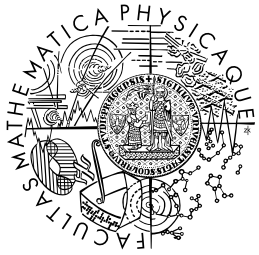

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Dráb

Variant calling using local reference-helped assemblies

Name of the department

Supervisor of the master thesis: Mgr. Petr Daněček, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Variant calling using local reference-helped assemblies

Author: Martin Dráb

Department: Name of the department

Supervisor: Mgr. Petr Daněček, Ph.D., Department of Software Engineering

Abstract: Abstract.

Keywords: high-throughput sequencing variant calling local assembly de Bruijn graphs

Dedication.

Contents

1. Introduction

Despite technological advances in the past fifteen years, genome sequencing (determining the DNA sequence of an organism) remains a challenging task. With the current technology it is not possible to "read" the DNA sequence in one piece from the beginning to the end. Instead, relatively short fragments must be sequenced individually and then painstakingly assembled into the complete sequence. This is a very laborous and expensive process, complicated by the fact that large proportion of most genomes consists of repetitive sequences. Therefore, instead of the pure de novo approach of assembling the complete sequence from scratch, most of the sequencing done today merely aims to determine differences between the sequenced organism and a complete, gold standard reference genome obtained previously by other means. This approach is known as *resequencing* and is, comparatively, a much simpler task.

The most widely used sequencing platform today is Illumina and the algorithms developed in this work assume Illumina data on input. The platform takes advantage of massive parallelism, where an ensemble of DNA molecules is fragmented into very short pieces (usually ~500 bp) and then sequenced in parallel from both ends. In one sequencing run, more than 10^9 of short sequences are obtained simultaneously. These sequences are called *reads*. Due to technical limitations, reads are only ~150 bp long, but when they are mapped to the reference genome, many will overlap and cover the whole genome, thus enabling to determine the differences between the sequenced organism, such as single nucleotide polymorphisms (SNPs) or short insertions and deletions (indels). Collectively, these differences are refered to as *variants* and the process of determining the variants is known as *variant calling*.

[show IGV snapshot - what alignment looks like, with a SNP and an indel]

In order to distinguish between random sequencing errors and real variants, we need sufficient coverage (the average number of reads mapped to a position of the genome). Thus for variant calling we need first to confidently place the reads to the correct location of the genome (the problem known as *mapping*), correctly determine the exact sequence of matches, mismatches, insertions and deletions at the location (*alignment*) and finally apply a statistical model to tell appart true variation from random sequencing errors, mapping errors and alignment errors (variant cal ling).

Single nucleotide variants are easier to call, indels are more problematic. This is partly because reads containing indels can be often aligned in multiple ways, leading to ambigous alignments and false calls, especially in difficult parts of the genome of low complexity or high repeat content.

1.1 Basic Terms

For the rest of this thesis, a genome is viewed as a string of character, each represents a nucleotide, at certain position. Since DNA molecules consist of four types of nucleotides, only four characters, `A`, `C`, `G` or `T`, appear in the string. The four nucleotides have similar chemical structure. They are composed of a sugar molecule and a posphate group, which are identical for all four of them, and a

unique nitrogenous base. Hence the term *base* is often used when referring to a nucleotide at a specific position. Each of the bases pairs with one other, `A` with `T` and `C` with `G`, therefore knowing the sequence of one strand of a DNA double helix determines the sequence of the complementary strand. Hence the terms *base pair* and *base* are used interchangeably.

The algorithm developed and discussed in this work expect a large bunch of short reads on input. It expects that the sequenced organism and the reference genome are similar enough to, so an assumption that reads are correctly mapped can be made. Its task is to resolve local differences arising from incorrect read alignments. This is a subproblem of the assembly one, called *microassembly*.

Some assembly algorithm transform each read into a sequence of *k-mers*, substrings of equal length, usually named k. The read sequence of length l is decomposed into $l - k + 1$ k-mers $k_0, ..., k_{l-k}$ of length k. If we denote bases in the read as $b_0, ..., b_{l-1}$, the k-mer k_i covers bases from b_i to b_{i+k-1}. Such definition implies that adjacent k-mers overlap by $k - 1$ bases.

An example of transforming e sequence `TACTGGCC` into k-mers of length 3 is ilustrated on Figure 1.1. The sequence has 8 bases in length and 6 k-mers are created from it. The read sequence, as in all other figures in this work, is marked red, contents of individual k-mers is in yellow.

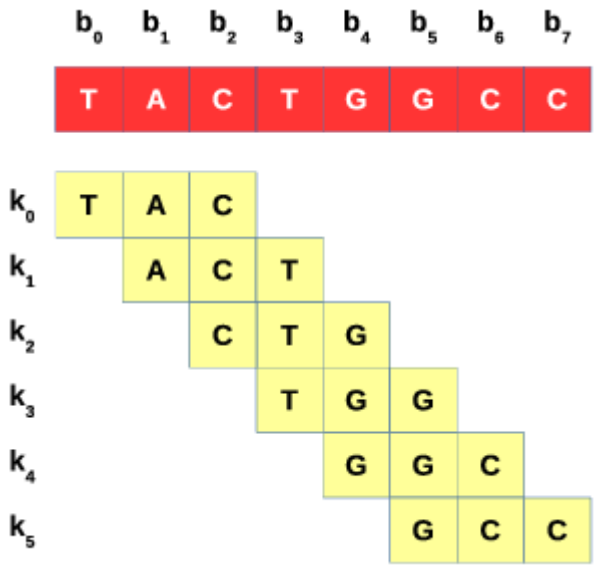

Figure 1.1: Sequence transformation into k-mers

1.2 The Model

The simplest mathematical assembly model, assumes that we are assembling genome string of length g, given a set of n reads of length l ($l << g$), each potentialy transformed into a sequence of $l - k + 1$ k-mers. The model assumes that reads are sampled from the genome string uniformly and randomly and that they are error-free. Since the probability of sampling a base at certain genome position is very low for single sampling event, and the number of sampling events is large, genome coverage (i.e. the number of reads covering a position) follows a

Poisson distribution. In other words, the probability that base at certain position is covered by k reads is

$$\frac{c^k}{k!} * e^{-c}$$

c is a base coverage depth, also known as sequencing depth, and can be computed simply as a total number of bases within the input read set divided by the length of the genome.

$$c = \frac{n * l}{g}$$

Very similar relations apply for k-mer coverage depth, only the formula for the k-mer coverage depth needs to be changed to $\frac{n*(l-k+1)}{g-k+1}$. These details brings answers to at least two important questions. Primo, how many reads need to be created to (statistically) cover the whole genome string, and, secundo, how to determine whether a given read set is free of sequencing errors?

The percentage of genome not covered by any read from the set is equal to $P(k = 0) = e^{-c}$. Multiplying it by the genome size g gives us the expected number of bases with zero coverage. So, to cover the whole genome, this number must be lower than 1 which places condition of $c > lng$. For example, to cover the whole human genome ($g = 3 * 10^9$), the read coverage depth must be at least 22.

Answer to the second question is implied by the facts that k-mer coverage of the genome follows the Poisson distribution. In an ideal case, the k-mer frequency distribution function of an error-free read set would follow the probability mass function of the Poisson distribution, meaning that frequencies of most of the k-mers are near the k-mer average depth. However, when we introduce possibility of sequencing errors, some k-mers would be sampled less often and some become even unique. The k-mer frequency distribution of such a read set does not follow the Poisson distribution. The aspect of sequencing errors and their means of their correction are described in great detail in Chapter 3.

1.3 The Real World

In contrast to the idealized model described in the previous section (1.2), read seuqncing data contain sequencing errors. In other words, some of the reads contain incorrectly interpreted bases. To help with identifying such bases, each base of a read has its *base quality*, a probability that the given base is incorrect. Bae quality (BQ) is usually represented as a Phred score Q defined as

$$P(base\ is\ wrong) = 10^{-\frac{Q}{10}}$$

Phred score has a natural interpretation, for example $Q = 40$ indicates one error in 10000 bases, $Q = 30$ one error in 1000 bases and so on (Figure 1.2 map the score values to probabilities). When stored in a text format, such as SAM or FASTQ, base qualities are encoded as ANSI characters. Because the first 32 ASCII characters are not printable, and the space character is coded by 32, the base quality values are incremented by 33. When loading the reads from such a text format, this needs to be taken in account.

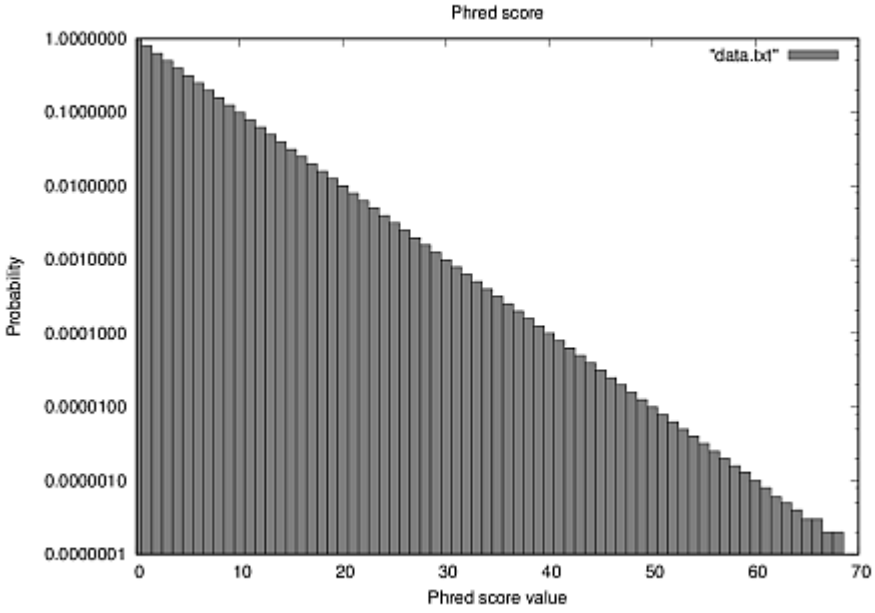

Figure 1.2: Mapping Phred score values to probabilities. Be aware of the logarithmic scale on the y axis

Mapped reads are accompanied by their position (POS) in the reference sequence, by additional alignment details and a mapping quality (MQ), which is a measure of mapping confidence expressed as a Phred score. Although the position information may be wrong, it may help us in cases when we are interested in correcting only a part of the genome and would like to filter out reads that do not map confidently.

The alignment details are given in form of a CIGAR string which describes how the individual bases of the read map to the reference. The string is formatted as a set of numbers defining the number of bases, and a character code describing the alignemnt alignment. The most common operations include:

- **Match (M)**. Alignment match. That can represent both sequence match and mismatch.
- **Mismatch (X)**. The sequence mismatch. In practice, this code is seldomly used since M may code both matches and mismatches
- **Insertion (I)**. The sequence is inserted to the reference at a given position. Then, the read continues to follow the reference at this position plus one unless alignment operation following the insertion tells otherwise.
- **Deletion (D)**. The read sequence skips the corresponding part of the reference.
- **Hard-clip (H)**. The original read was longer, but was trimmed in downstream analysis.

- **Soft-clip (S)**. This part of the read could not be aligned well and was excluded from the alignment.

For example, assume a reference sequence `CAGGTGTCTGA` and the read GGTGAATCTA aligned as follows:

```
          1 2 3 4 5 6     7 8 9 A B
Reference: C A G G T G     T C T G A
Read:        G G T G A A T C T   A
```

The read was mapped to the reference position 3 and the CIGAR string of the alignment is `4M2I3M1D1M`.

Genomes of diploid organisms, introduce another problem not covered by the simple mathematical model covered earlier. Since such organism owns two sets of chromosomes (one from each of its parents), both sets are present within the read set obtained by chopping the genome to short reads. That implies that two genome strings are present within, and need to be reconstructed, from the input read set.

1.4 Major Assembly Algorithm Classes

Currently, there are two widely used classes of algorithms: overlap-layout-consensus (OLC) and de-bruijn-graph (DBG) [8]. Algorithms belonging to the former base their idea on constructing a graph recording overlapping reads and extracting the alternate sequences from it. Main idea behind the DBG class is to chop all reads into k-mers that are then uses to construct a de Bruijn graph which is, after some optimizations, is subject to derivations of alternate sequences.

1.4.1 De Bruijn Graphs

De Bruijn graphs (DBG) were originally invented as an answer to the superstring problem (finding a shortest circular string containing all possible substrings of length k (k-mers) over a given alphabet) [7]. For a given k, de Bruijn graph is constructed by creating vertices representing all strings of length $k-1$, one per string. Two vertices are connected with a directed edge if there exist a k-mer such that the source and destination vertices represent its prefix and suffix of length $k-1$ respectively. The k-mer labels the edge.

Formally, let L_k be a language containing all words of length k over an alphabet σ, then de Bruijn graph $B(V, E)$ is defined as

$$V = \{v_w | w \in L_{k-1}\} \tag{1.1}$$

$$E = \{(v_u, v_w) | x \in L_k, uisitsprefixandwitssuffix\} \tag{1.2}$$

$$\tag{1.3}$$

The shortest superstring is found by concatenating all $k-1$-mers represented by vertices on any Eulerian cycle of the corresponding de Bruijn graph. Since a polynomial-time algorithm for finding an Eulerian cycle is known, the superstring problem can be effectively solved.

Application to Genome Assembly

SImilarly, de Bruijn graphs can be used for genome assembly, especially if the genome question is circular. Assume that, for a fixed k, all k-mers of the target genome are known. Then a DBG can be constructred in two different ways:

- **K-mers are represented by edges**. Each edge is labelled by a k-mer in the same way as in the graph used for solving the superstring problem. Each edge connects vertices representing (k-1)-mers forming prefix and suffix of its k-mer. The genome string can be reconstructed by choosing the right one from all possible Eulerian cycles. This approach is presented and discussed in [7].
- **K-mers are represented by vertices**. Each k-mer is represented as a single vertex. Edges reflect the k-mer order within reads. To recover the genome, one needs to find a Hamiltonian path of the graph, which is one of NP-complete. problems. Among others, HaploCall [6] used by GATK takes advantage of this way of de Bruijn graph usage.

HaploCall

Rather than assembling whole genome at once, HaploCall starts with detection of regions that are likely to contain variants. A score is computed for every genome position, reflecting the probability of variant occurrence. Regions are formed around positions classified as interesting. As subset of the input reads is assigned to each of the regions; reads are selected based on their mapping position. Each region is then processed separately and independently of others.

The active region processing phase starts by decomposing the reference sequence covering the region into k-mers and using them as vertices in a newly constructed de Bruijn graph. Edges connect vertices representing k-mers adjacent by their position within the reference. For each edge, a weight value is initialized to zero.

When the reference is transformed into the graph, a similar process happen with each of the input reads assigned to the active region. The read is decomposed into k-mers that are inserted into the graph in the same manner as the reference k-mers. Again, edges follow the order of k-mers within reads. New vertices are created only for k-mers not seen in the reference, similar case applies to new edges — if an edge denoting adjacent position of two k-mers already exists, its weight is increased by one. Otherwise, a new edge with weight initialized to 1 is inserted into the graph. The weight values actually count number of reads covering individual edges.

After inserting all the reads into the graph (the HaploCall documentation refers to the step as Threading reads through the graph), it is time to simplify the graph structure a little bit. The main concern here is to remove graph sections created due to sequencing errors present in the input reads. Such sections are identified by their low read coverage and removed. Other structure refinements are performed, including removal of paths not leading to the reference end.

In the next stage, most probable sequences are extracted from the resulting graph. Each sequence is represented by a path leading from the starting k-mer of the reference to ending one and its probability score is computed as the product

of so-called transition probabilities of the path edges. Transition probability of an edge is computed as its weight divided by the by the sum of weights of all edges sharing the same source vertex. By default, 128 most probable paths are selected.

The Smith-Waterman algorithm is used to align each of the selected sequences to the reference. The alignment is retrieved in form of a CIGAR string that indicates places of possible variants, such as SNPs and indels. Indel positions are normalized (left aligned).

The CIGAR strings actually contain a super set of the final set of called variants. To filter out false positives, other solutions are employed (for example, the HaploCall documentation mentions UnifiedGenotyper for GATK).

1.4.2 Overlap-Layout-Consensus

Algorithms taking advantage of this approach usually do not decompose reads into k-mer sequences. Reads are considered to be the smallset units. As the name suggests, the work is done in three steps.

Main purpose of the overlap phase is, not surprisingly, to compute overlaps between all possible pairs of the input reads. Since number of short reads within high coverage data sets may go to milions, the process of overlap computation may take a lot of processor and memory resources. Overlapping reads are connected into longer seuqnces called *contigs*. Actually, a graph is being constructed, its vertices represents individual reads (or contigs) and edges represent overlaps.

The definition of read overlapping is not strict. Two reads are considered overlapping (and thus, are connected by an edge) if they overlap at least by t bases, allowing several mismatches. The constant t is a parameter of the algorithm and is called a *cutoff*.

During the *layout* phase, a read-pairing information is used to put disconnected parts of the read overlapping graph together and resolving structures created by sequencing errors. The *consensus* phase is responsible for deriving candidate seuqnces for variant calling from the graph. In an ideal case, the cadidate sequences would be the Hamiltonian apths of the graph.

1.5 Goals of this Work

Main goal of this work is to develop an algorithm capable of variant calling on high-thorought data sets, comparable to, or more precies than methods currenlty employed. The approach used by HaplotypeCaller (HaploCall) should be used as an inspiration which implies the newly developed algorithm would take advantage of the reference sequence and would belong to the class of algorithms utilizing de Bruijn graphs.

The new algorithm should accept the input data set in widely used formats, such as SAM for the read set and FASTA for the reference sequence, and output the called variants in the VCF format. Its results need to be compared to at least two other assembly and variant calling tools. GATK (HaploCall) [6], as a representative of the DBG class and reference-aided methods, and FermiKit [9], utilizing the OLC concept.

2. The Algorithm

Our algorithm takes a reference sequence and a set of reads as inputs, and outputs a VCF file containing all detected variants. Both inputs are read into memory during the initialization phase, there are no memory-saving optimizations employed. In other words, no index files are used for the input data.

The variant calling is done on region basis. The reference sequence is divided into regions of constant length (2000 bases by default), sometimes also referred as *active regions* and with 25 % overlap. Reads are assigned to individual regions according to their mapping position. All regions are called independently and in parallel. To detect the variants, the following steps are performed:

- the reference sequence covering the active region is transformed into a De Bruijn-like graph,
- reads assigned to the active region are integrated into the graph,
- the graph structure is pruned and optimized,
- variants are extracted,
- genotypes and phasing are determined,

2.1 Input, Output and Preprocessing

The reference sequence is expected to be in the FASTA format and starting at position 1. In a typical scenario, the whole chromosome is provided as an input. The FASTA file may contain multiple distinct reference sequences (covering mutliple chromosomes). Reference sequences are processed one at a time which means that at most one sequence is present in memory at any moment. Only characters `A`, `C`, `G` and `T` are considered valid nucleotides. Other characters, such as `N` used to mask low-complexity regions, are treated as invalid and reference regions filled with them are not subject to variant calling.

The read set needs to be stored in a text file reflecting the SAM format. Presence of header lines is not required, all information is deduced from the reads. Since the algorithm does not perform any error correction on its own, the input reads must be corrected beforehand. The error correction is supported as a separate command of the tool implementing our algorithm. The correction method was adopted from the `fermi-lite`[11] project and Chapter 3 describes it in detail.

After the whole SAM file is read into memory and parsed, reads considered useless for the purpose of variant calling are removed from the set. Contents of the `FLAGS` column of the SAM file serves as a main filter, since it is used to detect the following types of undesirable reads:

- **unmapped**, recognized by the bit 2 set to 1,
- **secondary alignments**, detected by the bit 8 set to 1,
- **duplicates**, bit 10 is set to 1,

- **supplementary**, bit 11 is 1.

Also, reads with the mapping position (`POS`) set to zero and with mapping quality (`MAPQ`) lower then certain threshold, defined to 10 by default, are removed from the set. Read's mapping quality is also used to update individual base qualities. Each base quality q is updated according to the formula

$$q = min(q, MAPQ)$$

Several other SAM fields play a role in read preprocessing. The `CIGAR` string is used to detect and remove soft-clipped bases. The `QNAME` values are used to detect paired end reads, `RNAME` gives the name of the input reference sequence (chromosome).

The SAM file format is described in great detail in [10].

When all the reads are preprocessed this way, they are assigned to that respective active regions based on the mapping position. Then, the main algorithm comes into play.

2.2 K-mer Representation

Our algorithm works with k-mers only through an interface consisting from several functions and macros. The k-mer implementation is used as a blackbox which allows us to make several implementations and then choose the best fitting one without any consequences on the rest of the algorithm, exceptthose implied by the interface.

We currently use two implementations representing k-mers in different ways, shown on Figure 2.1. For performance reasons, the implementations may be switched only in compile time.

The upper part of the figure shows the representation of the *debuging* k-mer. The main idea behind this type of k-mer is to make the content easily human-readable, which is excellent for debugging purposes. The k-mer sequence is stored in a character array, each element represents one base. Except its context number, the purpose of which is explained in the next section, each debug k-mer also remembers its size. That value is used for debugging purposes only, mainly to recognize attempts to pass wrong k-mer size arguemnt to one of the k-mer interface routines. Debug k-mers also have an advantage of potentialy unlimited maximum size. It is clear that debug k-mers are not a good option in terms of performance, especially for bigger k.

Compact, k-mers in contrast, are more optimized for performance and quick append and prepend operations (moving the k-mer sequence forward or backward). Their design is greatly inspired by the k-mer implementation of the `fermi-lite` project. The k-mer substring is stored in three 64-bit integers. Each base is represented by three bits, each stored separately in the three integer fields. To read a i^{th} base (starting from zero), one needs to combine $(k - i - 1)^{th}$ bits of the integers and then translate the result by rules described in Table 2.1. The table also describes the meaning of the eight possible values.

Structure of a compact k-mer is displayed in lower part of Figure 2.1. Its advantages are a fixed size of 28 bytes regardless of k and performance (on the other hand, space occupied by a debugging k-mer depends on used k value and

is not known at the compilation time). However, limiting the maximum k-mer size to 63 bases this way may impose a problem in case of assembling repetitive regions.

Table 2.1: Base representations in debugging and compact k-mers

Compact value	Debugging value	Meaning
0	`A`	The base `A`
1	`C`	The base `C`
2	`G`	The base `G`
3	`T`	The base `T`
4	`B`	Denotes beginning and and of the active region.
5	`H`	Used to represent k-mers of helper vertices.
6	`D`	Used to represent k-mers of helper vertices.
7	`N`	-

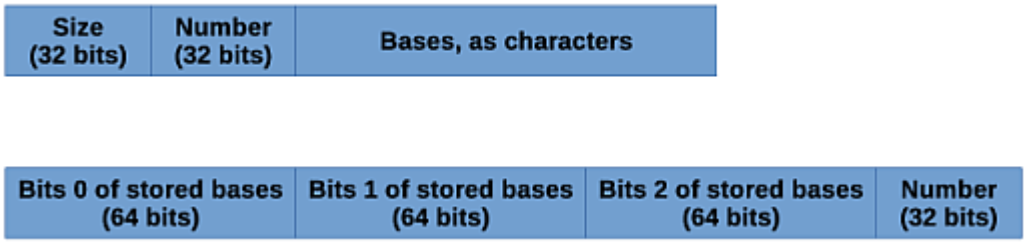

Figure 2.1: Two possible k-mer representations used by our algorithm; debugging (the upper part) and compact (the lower part).

All k-mer implementations usable by our algorithm must reserve some space for storing a *k-mer context number*. The number is used to differentiate even between k-mers that contain the same substring of length k. That implies that a k-mer, according to this new definition, is unique only if it differs in both the substring and the context number from all other k-mers. K-mers that differ only in the context number are sometimes referred as *equal by (sub)string* or *equal by sequence*. The context number proves to be useful when fighting repeats within the reference sequence (described in Section 2.3).

The whole chapter uses the term k-mer in this sense described above, unless explicity specified otherwise.

The `fermi-lite` project uses two bits to represent each base in the k-mer. Our algorithm can be modified to do the same, since the helper vertices actually do not need k-mers and exist mostly for the sake of code simplicity (no special handling regarding helper vertices is required). Similarly, there is no real reason for marking beginning and end of the active region by a special character, this was only useful for debugging.

2.3 Reference Transformation

The first step of the algorithm lies in transforming a reference sequence covering the selected active region into a de Bruijn-like graph. The idea behind this step is very similar to other assembly algorithms based on these graph types.

The reference is decomposed into k-mers, each overlaps with the adjacent ones by $k-1$ bases. K-mers representing the same sequence are differentiated by their context number, so each k-mer derived from the reference is unique. Two extra k-mers, denoting the beginning and the end of the active region are added to the set. Then, each k-mer is represented by a single vertex in the graph, and edges are defined by the order of the k-mers within the reference.

Formaly speaking, with the active region of length l represented as a string $b_1 \dots b_l$, k-mers $k_0, ..., k_{l-k+2}$ are derived from the region as follows:

- $k_0 = (Bb_1 \dots b_{k-1}, 0)$
- $k_1 = (b_1 \dots b_k, 0)$
- . . .
- $k_i = (b_i b_{i+k-1}, c_i), 2 \leq i \leq l-k+1$
- . . .
- $k_{l-k+2} = (b_{l-k+2} \dots b_l B)0)$

c_i represents the context number of the k-mer k_i. The number is set to zero for k-mers unique within the active region. On the other hand, let's assume that k-mers $k_{i_1}, ..., k_{i_n}, i_0 < ... < i_n$, contain the same string. Their context numbers are defined as

$$c_{i_j} = j,$$

k_0 and k_{l-k+2} are special k-mers added to the set in order to show the start and end of the active region within the graph. `B` is a virtual base that ensures these k-mers are unique. The bases must not appear anywhere else within the active region. All k-mers created in this step and all vertices created from them are also called as *reference k-mers* and *reference vertices*. Similarly, k-mers and vertices created during the read integration phase, are sometimes referred as *read k-mers* and *read vertices*.

Each k-mer k_i is transformed into a single vertex v_i. Edges follow the order of the k-mers in the active region. In other words, the edge set of the graph is

$$E = \{(v_i, v_{i+1})\} \quad 0 \leq i \leq l-k+1$$

Figure 2.2 displays a graph created by transforming the active region `ATCTGTATATATG`

with k-mer size of 5. The algorithm creates the following k-mers:

$$k_0 = (BATCT, 0) \tag{2.1}$$
$$k_1 = (ATCTG, 0) \tag{2.2}$$
$$k_2 = (TCTGT, 0) \tag{2.3}$$
$$k_3 = (CTGTA, 0) \tag{2.4}$$
$$k_4 = (TGTAT, 0) \tag{2.5}$$
$$k_5 = (GTATA, 0) \tag{2.6}$$
$$k_6 = (TATAT, 0) \tag{2.7}$$
$$k_7 = (ATATA, 0) \tag{2.8}$$
$$k_8 = (TATAT, 1) \tag{2.9}$$
$$k_9 = (ATATG, 0) \tag{2.10}$$
$$k_{10} = (TATGB, 0) \tag{2.11}$$

As can bee seen, there are two k-mers representing sequence `TATAT`, namely k_6 and k_8. Because of their distinct context numbers, they are represented as separate vertices. Introduction of the context numbers removed a loop from the graph. The loop can be observed on Figure 2.3 that shows a standard de Bruijn graph constructed from the same active region. K-mers k_6 and k_8 are represented by the same vertex. In order to recover the sequence, it is required to know how many times the loop was actually used during the transformation step.

Although this solves the problem of cycles for now, at least for now, things become more complicated in the next step of the algorithm which inserts individual reads into the graph

2.4 Adding Reads

The basic idea behind this stage is farily simple and similar to the approach used in the previous case. The read being added is decomposed into k-mers. If the k-mer is not present in the graph already, a new vertex is created. Again, vertices representing adjacent k-mers in the read are connected by edges.

Figure 2.4 shows a graph created by transforming a region of `ACCGTGGTAAT` and adding the read `ACCGTAGTAAT` to the resulting graph. K-mer size is set to 5. The read is divided into the following k-mers:

$$k_0 = (ACCGT, 0) \tag{2.12}$$
$$k_1 = (CCGTA, 0) \tag{2.13}$$
$$k_2 = (CGTAG, 0) \tag{2.14}$$
$$k_3 = (GTAGT, 0) \tag{2.15}$$
$$k_4 = (TAGTA, 0) \tag{2.16}$$
$$k_5 = (AGTAA, 0) \tag{2.17}$$
$$k_6 = (GTAAT, 0) \tag{2.18}$$

In this example, the k-mers k_0 and k_6 were already present in the reference graph but new vertices has to be created for the rest.

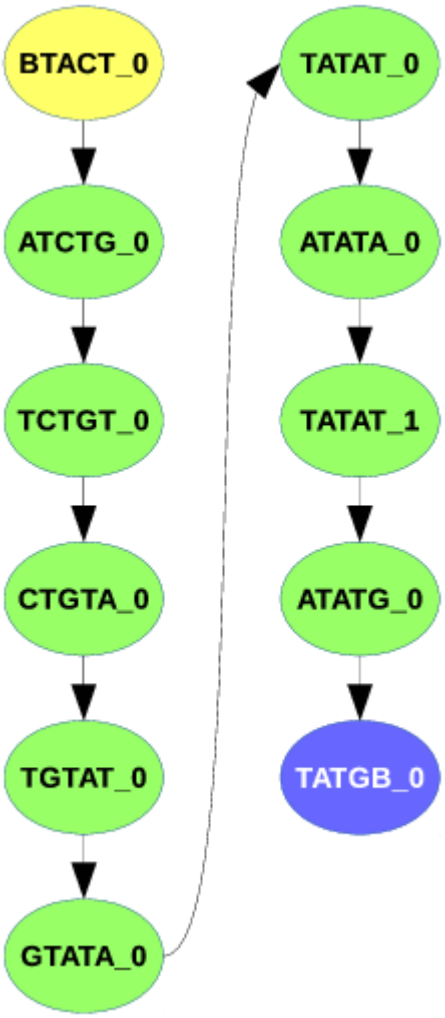

Figure 2.2: Graph resulting from the transformation of `ATCTGTATATATG` sequence

Finally, edges are added (if necessary) to show the k-mer order within the read. When talking about classical de Bruijn graphs, edges do not need to be expressed explicitly since they always connect adjacent k-mers. In case of our modification to de Bruijn graph, we chose to make the k-mer connection explicit. Two reasons led us to such decision:

- our definition adds context number to k-mers and we wish to connect only certain k-mers adjacent by their strings,
- we need to make connections between non-adjacent k-mers, e.g. shortcuts representing paths with no branches, or fullfilling purposes discussed mainly in Section 2.5.

Keeping graph edges explicitly has also its drawbacks, especially those related to performance. Classical de Bruijn graphs can be represented only by a set of their vertices, since the edges can be deduced on demand.

The figure also suggests how to retrieve the alternate sequence introduced by the read — by going through edges supported by the read and concatenating the last bases from the k-mers present on the path. K-mers covering start and end

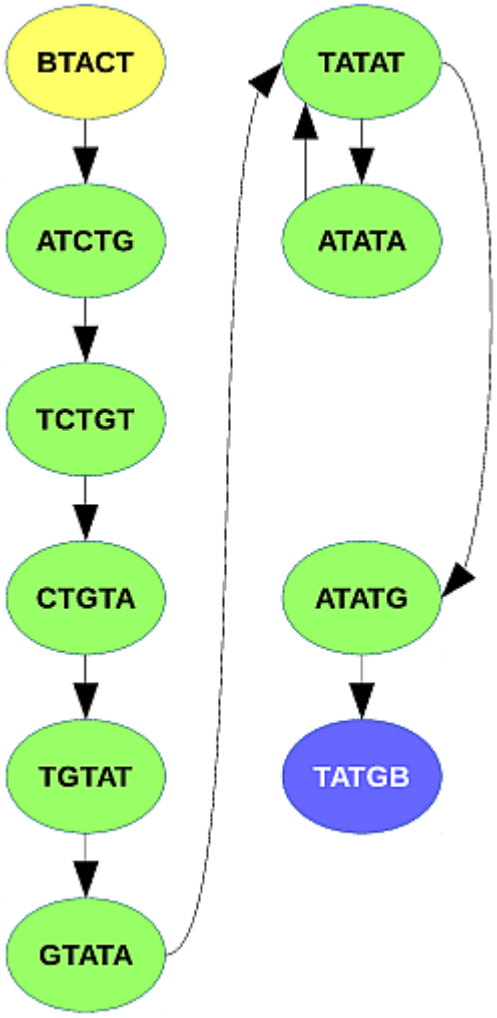

Figure 2.3: Transformation of the `ATCTGTATATATG` squence to a standard de Bruijn graph (with no k-mer context numbers)

of the reference region are the only exceptions; all except the `B` is used from the former, nothing from the latter.

Figure 2.4 reflects an ideal state, meaning that all places, where the read differs from the reference, are covered by distinct k-mers, and the distance between each two of them is greater than k. If these conditions are met, each single n-base long difference (SNP, insertion or deletion) adds at most $n + k - 1$ new vertices to the graph. Each difference then creates only two linear paths, one covering the reference, the other for the alternate allele. Such structures are called *bubbles* and are easy to work with.

However, it may happen that some of the k-mers covering a difference colide by sequence with either k-mers of the reference, or k-mers of other reads covering a totally different place of the active region. To minimize such problematic cases, additional graph transformations need to be made after all the reads are itegrated into the graph.

Unfortunately, the basic idea does not work in our case. Introduction of the k-mer context numbers prevented loops in the graph derived purely from the

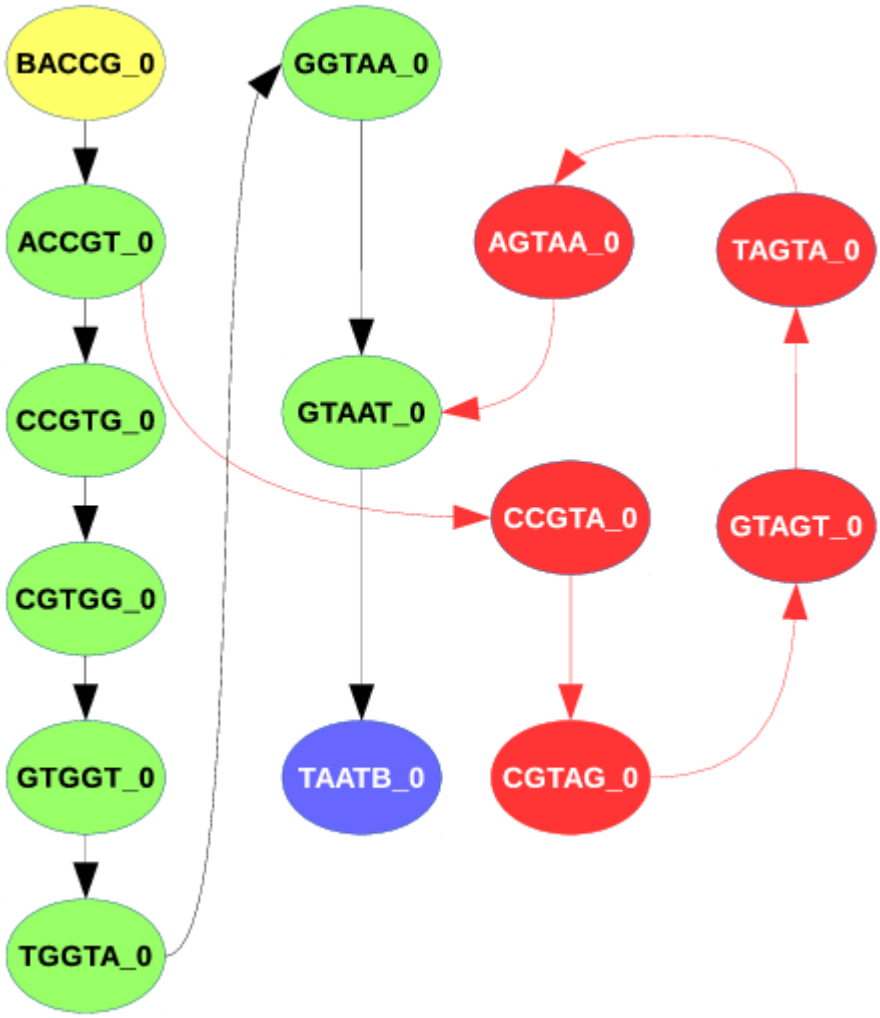

Figure 2.4: Basic idea behind adding a read into a de Bruijn graph

reference. But since multiple k-mers representing the same sequence may exist, it is not always possible to easily determine which of the vertices should be assigned to individual k-mers of the read. For example, if looking at the graph from Figure 2.2, it is not clear which vertex (or vertices) should be assigned to a k-mer with `TATAT` sequence.

We decided to solve the issue by transforming the basic idea into the following steps:

- decompose the read into sequence of k-mers (a so-called *short variant optimization*, described later, may be applied),
- to each k-mer assign a set of vertices with the same k-mer sequence (differences in context numbers are permitted),
- from each set, select one vertex to represent the k-mer of the read (the selection process is described later in this section),
- connect all read vertices in a way that respects the order of the k-mers in the read.

2.4.1 Transforming the Read into K-mers

Let's define a read of length n as a sequence $r_1 \dots r_n$ of bases. If the short variant optimization, described in the next paragraph, is not applied, the read is decomposed into individual k-mers $k_1, \dots, k_{n-k+1}$ in the same way as for the reference case, except that no extra k-mers to denote read start and end are created. The k-mers look as follows:

$$k_1 = (r_1 \dots r_k, 0) \tag{2.19}$$
$$\dots \tag{2.20}$$
$$k_{n-k+1} = (r_{n_k+1} \dots r_n, 0) \tag{2.21}$$

Then, the step described in Subsection 2.4.2 is applied.

As described in **??**, in an ideal case, a n-base long difference from the reference produces $n + k - 1$ k-mers different from all reference k-mers. To reduce the probablity that some of the new k-mers actually colide with either the reference, or another read, the *short variant optimization* may be applied. The optimization reduces the number of k-mers representing a n-base long difference to:

- n for an insertion,
- zero for a deletion,
- 1 for a SNP.

The optimization assumes that when recovering a sequence from the graph, only the last base of each k-mer, wth the exception of the starting one, is used. So, only k-mers covering the difference by their last base need to be added; reference k-mers may be used for the rest in case the difference is followed by a reasonable number of bases equal to the reference.

To make the k-mers covering a n base long difference by their last bases really unique, the first n bases of the first k-mer is changed to D, a virtual base used for helper purposes only. Rest of the k-mers is obtained by appending the bases of the difference to the first one. Keep in mind that n is always smaller than the k-mer size, since we are talking about the *short* variant optimization.

Figure 2.5 shows how the graph is optimized for a read containing SNP. The reference and read sequences are taken from Figure **??**. Since the difference has 1 base in length, only one k-mer (`DCTGA_0`) is used to represent it. The k-mer is followed by reference k-mers. As can be seen, their last base are equal to one of k-mers from Figure **??**.

The short variant optimization is applied for k-mer k_i if the following holds:

- There is only one reference vertex with a k-mer equal to k_{i-1} by sequence. Let's assume this is vertex v_{j-1}
- There is at most one vertex for k_i that either is a result of a read addition, or is a reference one but do not immediately follows the vertex v_{j-1} in the reference.

Such conditions are met when the read differs from the reference at base r_{i+k-1}. To determine whether the difference is only a short one, the Smith-Watterman algorithm is applied. If this is the case, the action depends on the difference type:

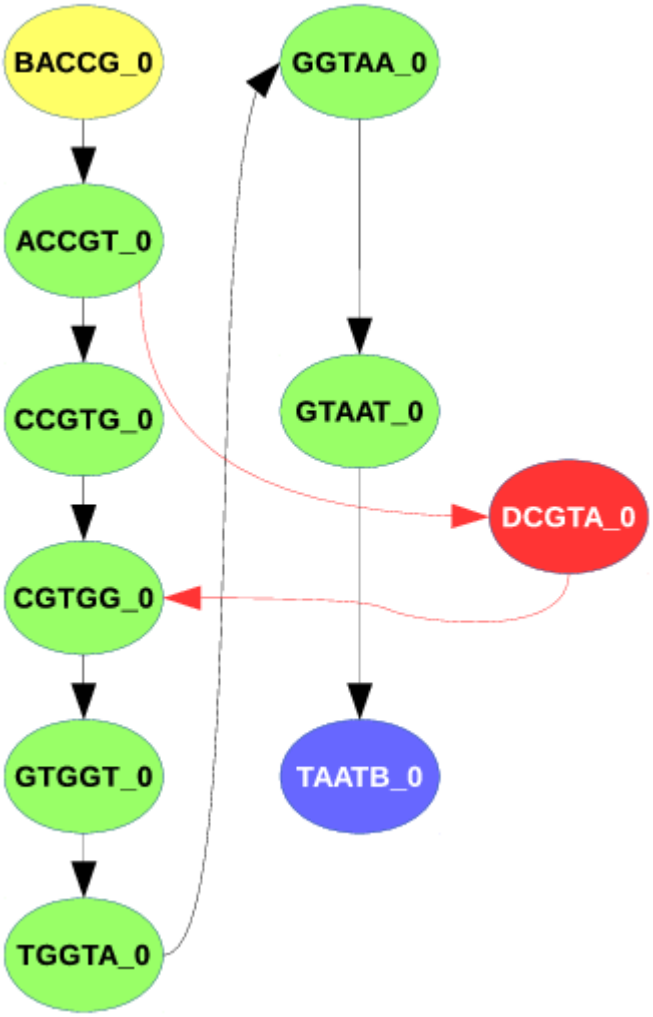

Figure 2.5: Short Variant optimization

- for n-base long deletion, k_i is defined as v_{j+n-1},
- for insertion of length n, k_i, ..., k_{i+n-1} are defined as with no short variant optimization and k_{i+n} is set to v_j,
- for SNP, k_i is left as such and k_{i+1} is defined as v_{j+1}

2.4.2 Assigning Sets of Vertices to K-mers

The process of assigning vertex sets to individual k-mers derived from the read in the previous step is quite straightforward — a set assigned to a certain k-mer x contains exactly the vertices sharing the same k-mer string. The k-mer context number is not taken into account. If a k-mer is not represented by any vertex of the current graph (thus, the k-mer would receive an empty set), a new vertex is created to represent it. With a classical de Bruijn graph, all sets would contain exactly one vertex. With k-mer context numbers, there can be multiple vertices per k-mer. This complicates the task of integrating reads into the graph because the graph may contain multiple paths representing a single read (by using differenct vertices with k-mers equal by sequence).

Previous steps of the algorithm, described above, impose the following conditions on the assigned vertex sets:

- each set contains either read, or reference vertices, but not both,
- if a set contains read vertices, its size is always one,
- sets containing reference vertices do not have such restriction,
- each two sets are either distinct (their intersection is an empty set) or equal.

The second and third condition holds because k-mer context numbers are used to differentiate reference k-mers but not the read ones. The fourth one is implied by the fact that each set contains all existing vertices with k-mers equal by sequence to the read k-mer being processed. So, if two sets have a non-empty intersection, they actually refer to the same read k-mer (k-mer context numbers are not used when processing read k-mers).

In formal terms, a set M_i is assigned to a k-mer k_i where

$$M_i = \{v_{i_j} | v_{i_j} \in V(G), kmer(v_{i_j}) \; equals \; to \; k_i \; by \; sequence\}$$

When a set is assigned to each k-mer, it is time to integrate the read into the graph in the form of a path, starting in the vertex representing k_1 and ending in the vertex covering k_{n-k+1}. SInce M_i sets may contain more than one vertex, it is necessary to select vertices to form a path best fitting to the read. To derive a good path, we decided to assume the following:

- they should follow the reference sequence in the forward direction,
- The probability of skipping large number of reference vertices (long deletions) is low,
- multiple reads cover one place, sharing appropriate parts of their paths.

These requirements cannot be enforced too strictly as de Bruijn graphs are not very suitable for coping with repeats of length k or more. The case of a difference containing a copy of reference at leat k baes in length might be enough to break the first assumption. The second assumption permits exceptions by definition. The third forms a base for most of the genome assembly algorithms.

In order to choose the correct vertices from the M_i sets, we decided to reduce it to a shortest-path problem on a helper graph the structure of which is defined by the sets and their contents as explained in 2.4.3.

2.4.3 The Helper Graph

The helper graph is an oriented layered graph. Each layer consists of all vertices contained in one reference M_i set. The order of the layers respect the order of M_i sets. Sets consisting of read vertices are not part of the helper graph. Only adjacent layers are connected by edges, their orientation reflects the order of the sets. Each subgraph consisting of two adjacent layers is a full bipartite graph. The structure of the helper graph does not take equality of M_i into account. In other words, when $M_i = M_j$ for $i \neq j$, both sets are represented within the helper graph as individual layers, even if they refer to the same vertices of the (main, non-helper) graph.

Formaly speaking, let $M_i = \{v_i^1, ..., v_i^{n_i}\}$ and let i index the reference sets only. Then the helper graph G_h can be defined as follows:

$$G_h = (V_h, E_h) \tag{2.22}$$

$$V_h = \cup_i M_i \tag{2.23}$$

$$E_h = \{(u, v) | u \in M_i, v \in M_{i+1}\} \tag{2.24}$$

By finding the shortest path leading from a vertex in the first layer to one in the last layer, we perfrom the process of selection of vertices representing the read in the main graph. The shortest path depends on weights of edges connecting the adjacent layers. In general, the weighting function follows these rules:

- the weight is increased by a *missing edge penalty* if there is a missing edge on the path from u to v in the main graph,
- the weight is increased by a *reference backward penalty* if reference position of u is greater or equal to the reference position of v,
- the weight is increased by a *reference forward penalty* if reference psotiion of u is far less than reference position of v.

The rules actually indicate why M_i sets covering read vertices are not parts of the helper graph – since their vertices maintain no reference position, only the missing edge penalties would apply and that can be included within missing edge penalties of the reference vertices only.

For an example of a helper graph, let's have a reference sequence `ACTATACTA` and a read `ACTAGACTA`. The left part of Figure 2.6 shows the main graph just after adding vertices for the reference and the read k-mers with short variant optimization applied. The resulting helper graph is shown on the right part of the figure.

Six k-mers are derived from the read which means that vertex sets M_0, ..., M_5 are assigned to them. Since the second k-mer is represented by a read vertex, the M_1 set is not included as a layer of the helper graph. Other sets contains reference vertices, so they form individual layers. Adjacent layers are then connected. Edges with applied penalties (only the reference backward penalty in this case) are depicted red. The black edges show the shortest path.

The shortest path select the first vertex from M_0 and the second one from M_5 to represent the read within the main graph (since other sets contain only one vertex, the selection process is trivial there). The resulting path in the graph can be used to correctly recover the sequence covered by the read.

As Figure 2.7 indicates, both graphs look a little bit differently when the short variant optimization is not applied. The main graph contains more read vertices which reduces the number of layers in the helper graph. Although the graphs are different, the sequence covered by the rad is the same and can be correctly recovered again.

2.5 Graph Structure Optimization

When all reads are integrated into the De Bruin-like graph, it is time to optimize its structure in order to get rid of unpopulated paths, usually created by read errors, and resolve some other issues caused mostly by repetitive regions inside either the reference or the reads.

2.5.1 Connecting Bubbles

Figure 2.8 demonstrates one class of the structural problems. A subset of reference sequence was transformed into vertices 1, 2, 3, 4, 5 and 6. A set of reads is represented by a "path" 2, 3, 4, 5, R1, R2, 1, 2, 3, R3, 6. The left part of the figure shows how such a graph woud look like without any optimizations. It is clear that recovering the correct sequence would not be trivial.

However, since we know that the path leads from R2 to R3 (through 1, 2, 3), we can theoretically replace edges (R2, 1) and (3, R3) by a special edge (R2, R3), as the right part of the figure suggests. An information about the sequence covered by vertices 1, 2 and 3 needs to be recorded within the new edge. Recovering the correct sequence from the right part of the figure does not impose a problem since it is just a simple bubble.

A more general, and a very typical, situation is shown on Figure 2.9. The subgraph contains a subset of the reference (vertices 1, ..., $n+1$) and two bubbles; one ending by $R1$ and connected to 2, another starting at $R2$ and leading from n. If edges I_1 (the input edge) and O_1 (the output edge) share reasonable amount of reads ($|reads(I_1) \cap reads(O_1)| > thershold$), the subgraph may also be interpreted as that the reads contain a sequence of length $n-1$ that is also present in the reference. In that case, it is wise to connect the vertices $R1$ and $R2$ directly the same way as on Figure 2.8, bypassing the reference part. The edge maintaining the direct link is marked as C_1 (the connecting edge). Reads shared by the input and the output edges are moved to the connecting one.

If C_1 is created, the read set intersection is also used to decide whether the edges I_1 and O_1 should be removed. The input edge is deleted if does not share

enough reads with the next reference edge (meaning there are no valid sequences leading through both these edges). Similarly, the output edge is deleted in case it does not share enough reads with the last reference edge (no valid paths goes through the edges).

Figure 2.10 depicts probably the most general case; multiple reads share the same sequence of n bases $(R_1, \ldots, R_n)$. There is k input and l output edges. To determine the association between individual input and output edges, the intersection of covering reads is used again and connecting edges are created if necessary. More precisely, the following rules apply:

- If i^{th} input and j^{th} output edges share reasonable amount of reads ($|reads(I_i) \cap reads(O_j)| > threshold$), a connecting edge $C_{i,j}$ is created and the shared reads are moved to it. The new edge starts in $source(I_i)$ and ends in $dest(O_j)$.
- I_i and O_i are removed in case their read coverage drops below threshold as a result of moving it to the newly created C_i edges.

2.5.2 Helper Vertices

The bubble connection optimization works well when the bubbles being connected contain at least one read vertex. If this is not the case, however, the effect of replacing input and output edges with connecting edges leads to a destruction of the sequences recorded within the graph.

As an example, consider a case ilustrated by the left part of Figure 2.11. The relevant part of the reference runs from vertex 1 to 7 and the read coverage supports a path of $3 \to 4 \to 5 \to 6 \to 1 \to 2 \to 4 \to 5 \to 7$. Applying steps described in this section results in creation of connecting edges $6 \to 4$ and $2 \to 7$ and removal of the edges $6 \to 1$, $2 \to 4$ and $5 \to 7$. Such a graph cannot be used to recover the alternate sequence.

As this problem arises only when the bubbles are formed entirely by edges connecting reference vertices (the short variant optimization produces such cases for deletions), the countermeasure is quite straightforward; the solution is to insert special placeholder vertices with an empty sequence. We use a conservative approach for their insertion which means they divide the following types of edges:

- output degree of their source vertex is greater than one,
- input degree of their destination is greater than one.

The right part of Figure 2.11 indicates where the helper vertices would be inserted. They divide edges $6 \to 1$, $2 \to 4$ and $5 \to 7$. If the bubble connection is applied now, it leads to creation of edges $H1 \to H2$ and $H2 \to H3$ and deletion of $H1 \to 2$, $3 \to H2$, $H2 \to 4$ and $5 \to H3$. Thanks to the optimization, the alternate sequence can be now recovered ($3 \to 4 \to 5 \to 6 \to H1 \to H2 \to H3 \to 7$).

2.6 Variant Calling

The graph structure optimization phase ends by removing nodes and edges with insufficient read coverage. Then, alternate sequences covered by the reads are

extracted. Because the intra species variability of biological sequences is relatively small, it is practical to work with a list of differences (*variants*), rather than whole sequences. The algorithm identifies parts of the sequence shared with the reference genome and the alternate alleles, which roughly correspond to one line of a VCF file.

Variants are extracted by detecting certain subgraphs. When such a subgraph is discovered, the variant sequence is determined and integrated back into the graph by replacing its reference edges by a signle *variant edge*. Read edges unique to the variant are also removed from the subgraph which simplifies its structure. The process of variant detection stops when no suitable subgraphs are found.

Figure 2.12 shows four types of subgraphs used for variant extraction and demonstrates how they are modified in the process. The reference part of the graph must always start in vertex 1, end in n and all inner nodes must have one input and one output edge Only edges of reference or variant type may be present in the reference part. In all cases, this path is replaced with a variant edge, shown as blue, connecting directly 1 and n. All edges that are removed as a result of the extraction are plotted as discontinuous lines.

Simple Bubble

Read edges and vertices form a linear path leading from 1 to n. Since the edges are not part of any other variant, they all are removed and the whole subgraph degenerates only to two reference vertices connected by a variant edge (Figure 2.12c).

Bubble with Inputs

Figure 2.12b shows a subgraph where the read node has more than one input edge. In such case, only the sequence of read edges leading from the 1 vertex to the first vertex with more inputs than one is removed. Other read edges may take part also in other variants.

Bubble with Outputs

This subgraph may be viewed as an opposite to the bubble with inputs. All read vertices in the subgraph are allowed to have more outputs than one, but only one input. Only the last sequence of edges, starting in the last read vertex with output greater than one and ending in the n vertex is removed, since it may participate only in one variant (Figure 2.12b).

Diamond

The most complicated case is shown under in Figure 2.12d. The sequence of read vertices may contain one with output degree greater than one followed (not necessarily directly) by one with unresticted input degree. Only the edges covering one part of the supposed diamond are present in one variant only and hence are removed.

2.7 Variant Graph and Variant Filtering

When variants are extracted from the de Bruijn-like graph, they need to be filtered and their genotype and phasing computed. For this, a *variant graph* is built and the task is transformed into a graph colouring problem.

The variant graph represent each variant by two vertices; one for its reference and one for its alternate sequence. The final task is to colour each vertex by one of these colors:

- **Blue**. The variant part is used by the first sequence.
- **Red**. The variant part is used by the second sequence.
- **Purple**. Both sequences go through the variant part.

If a vetex is coloured purple, the vertex representing the other part of the variant is not required (no sequence goes through it) and is removed from the graph. The deletion usually happens only to the vertices representing the reference pats, since removing a vertex of the alternate path means that the variant was filtered out.

Before colouring, graph vertices are connected by several types of bidirectional edges that place vairous conditions on the colour of their sources and destinations.

- **Variant edges** connect vertices representing parts of one variant. Their source and destination must be coloured differently.
- **Read edges** connect variant parts covered by the same subset of reads with size greater than a threshold, such vertices need to be coloured by the same colour, with exception of purple. If one of the vertices is purple, an arbitrary color may be assigned to the other.
- **Pair edge** put together variant parts that are covered by paired-end reads. Since paired-end reads indicate that the variants covered by them belong to the same alternate sequence, the coloring restrictions are the same as for the read edges. Two vertices are connected by a paired edge if the share a sufficient number of paired-end reads.

The graph usually has a form of many components. Variants in one component share the same phasing.

When the graph is coloured, the genotype and phasing information are known. The variants are written to the resulting VCF file.

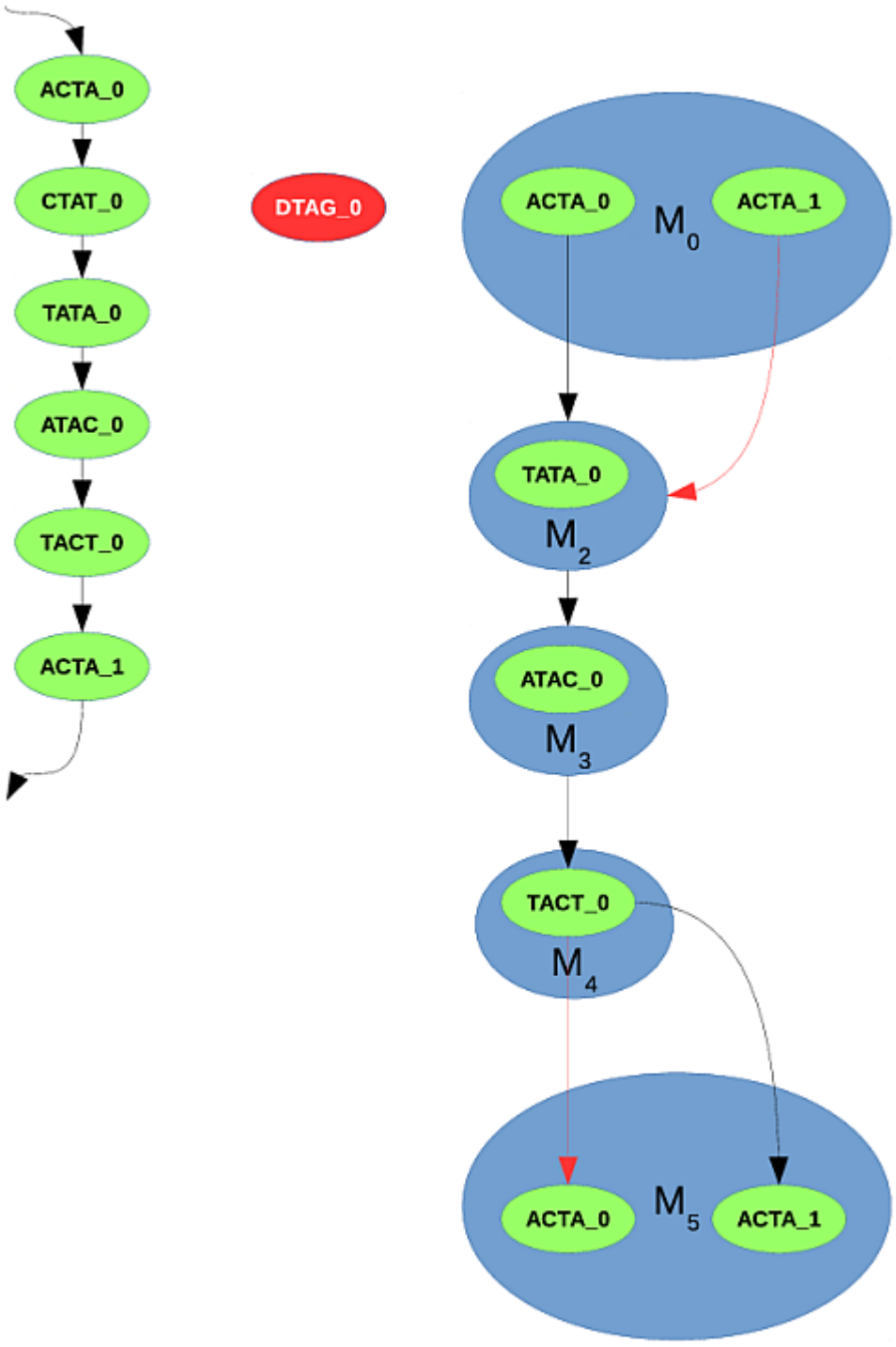

Figure 2.6: Helper graph creation with short variant optimization

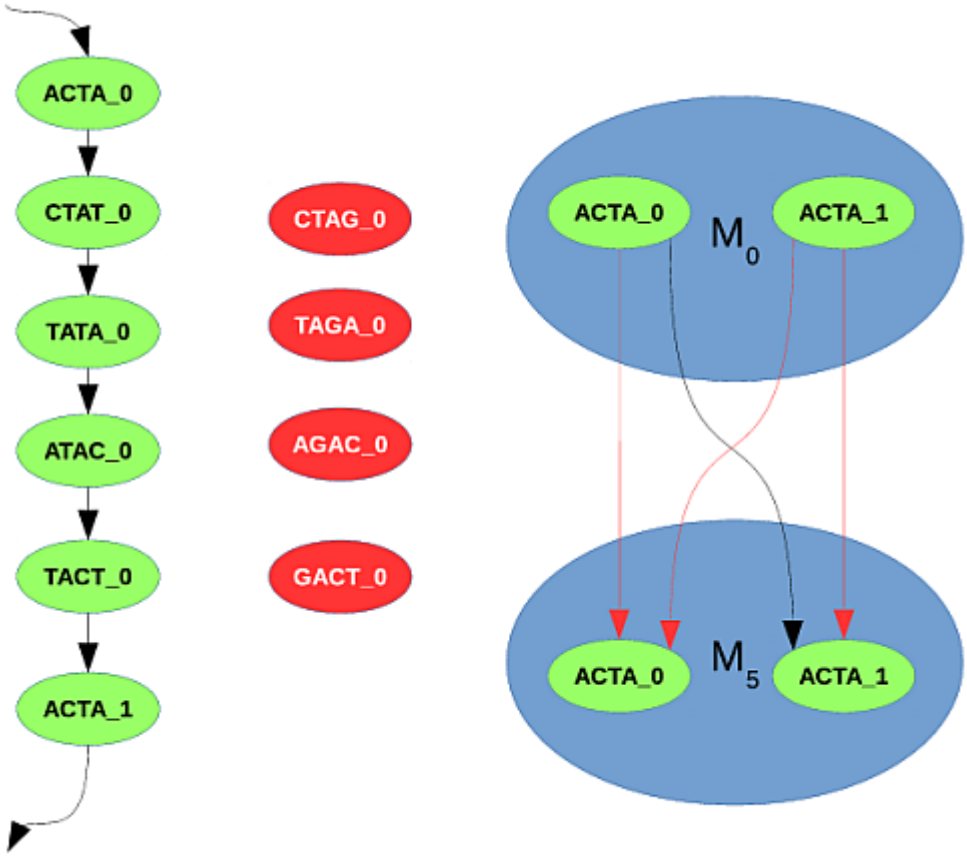

Figure 2.7: Helper graph creation without short variant optimization

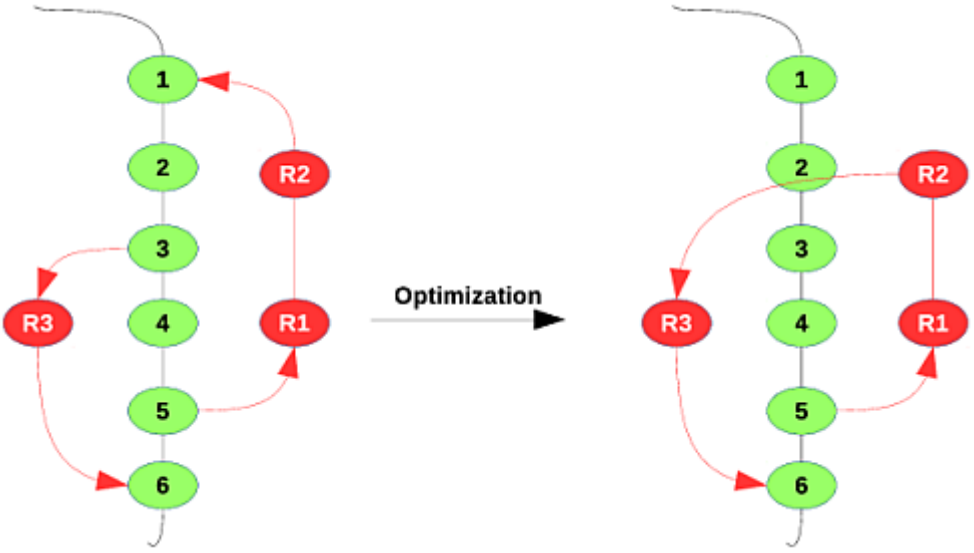

Figure 2.8: Benefits of connecting bubbles. Green color marks vertices added during the reference transformation, red is used for read vertices

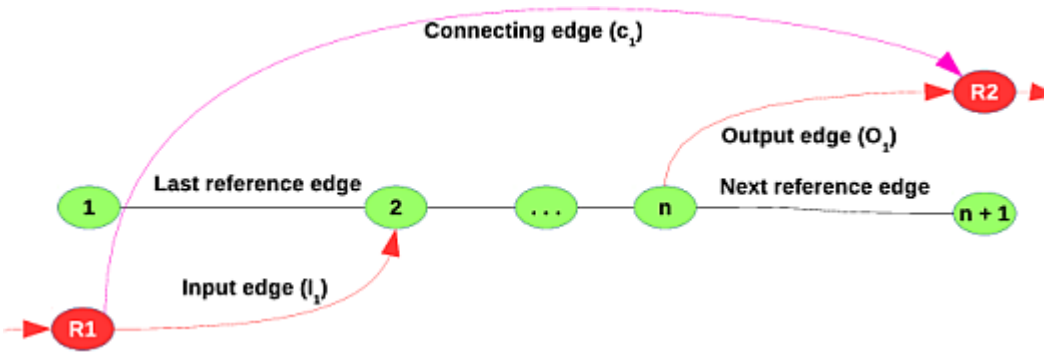

Figure 2.9: A subgraph required for connection, colors have the same meaning as in Figure 2.8

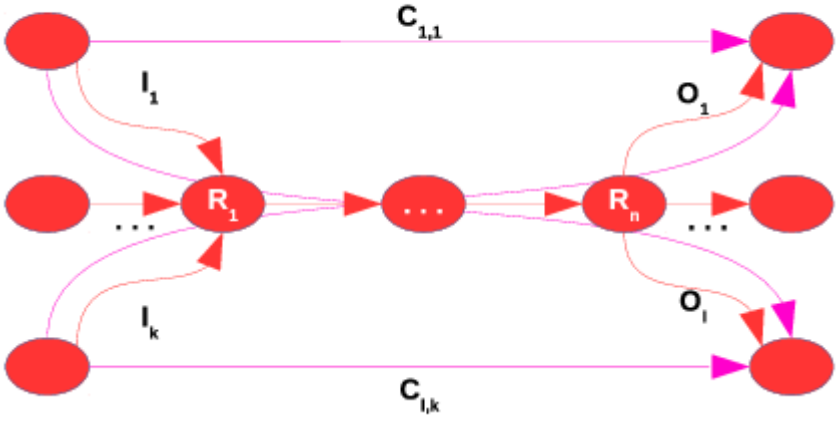

Figure 2.10: A general form of the subgraph

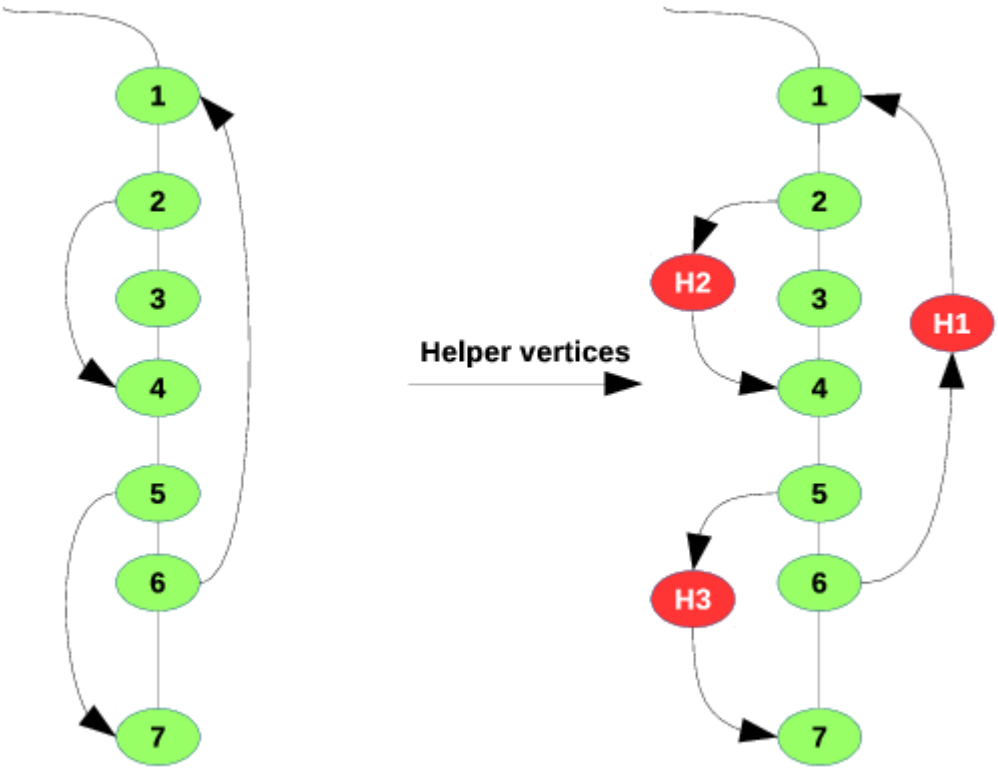

Figure 2.11: Use case for helper vertices

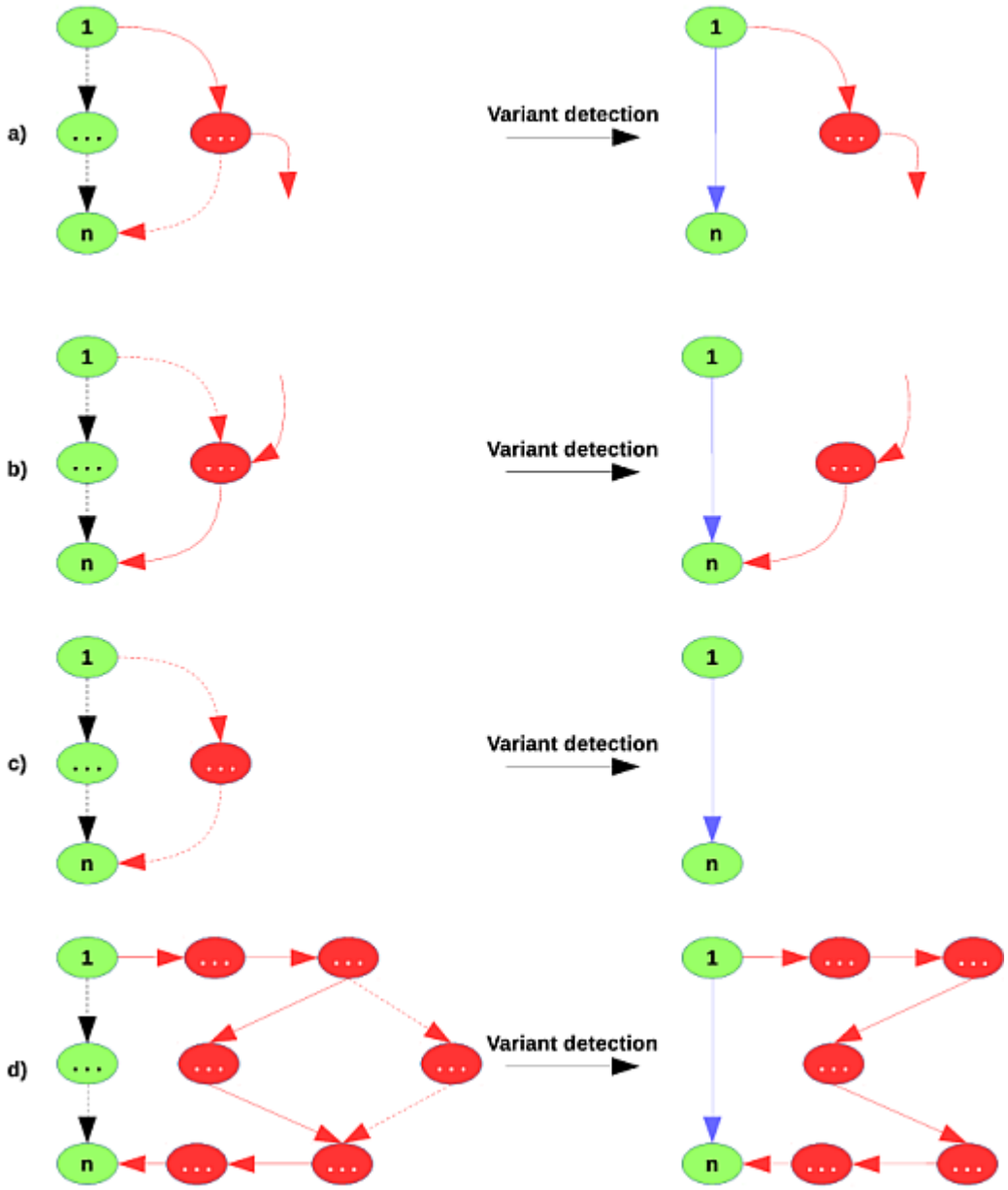

Figure 2.12: Variant detection cases

3. Read Error Correction

Read data used as an input to many assembly algorithms contain plenty of errors, such as wrongly read bases. To make the data usable for assembly, an error correction step is required. However, it does not remove all the errors and assembly algorithms must cope with that fact, especially when dealing with read ends.

Currently, two different approaches are used to correct read errors, and both are based on transforming individual reads into series of k-mers. One is based on detecting errors as low covered edges (or paths) in a de Bruijn graph, the other works with a k-mer frequency distribution. During development of our algorithm presented in this thesis, we made several attempts to implement an error correction algorithm based on de Bruijn graphs. Since we use these graphs also during assembly performing error corrections on them seemed to be a natural choice. Although they definitely helped to improve the quality of input reads, all our attempts performed worse compared to the k-mer frequency distribution approach.

In the end, we decided to adopt the error correction algorithm used by the `fermi-lite` library [11] and based on k-mer frequency disbribution. This chapter describes both approaches.

3.1 De Bruijn Graphs

This method transforms the input set of reads into a de Bruijn graph in a way very similar to one used by our assembly algorithm. Although implementation details may differ, the basic idea is the same: each read mapped to certain active region is divided into a sequence of k-mers, each k-mer serves as a vertex and the edges follow the k-mer order within the sequence. Reference sequence, covering the active region, may also be included in the graph.

The basic assumption is that errors produce unique k-mers and thus also nodes and edges with low coverage. Low-covered edges with source vertices that have output degree greater than one are especially interesting. A change of even a single base can divert a read path through edges with higher read coverage. The locality of the change depends on the k-mer size used.

A simple example demonstrating the main idea behind the method is displayed in Figure 3.1. Many reads share a sequence of `TTGCGCTAA`. However, there is also a single read that contains one error — `TTGCACTAA`. The de Bruijn graph shown on the figure uses k-mers 4 bases long. Combination of both sequences produces a standard bubble.

If the bubble was supported by reasonable amount of reads, it would be treated as a SNP. However, because there is only one supporting read, it may be reasonable to consider its divergence from other reads as an error, and to correct it, so the read path would follow more populated edges. When the correction is done, the resulting graph becomes linear, as shown on the right side in Figure 3.1.

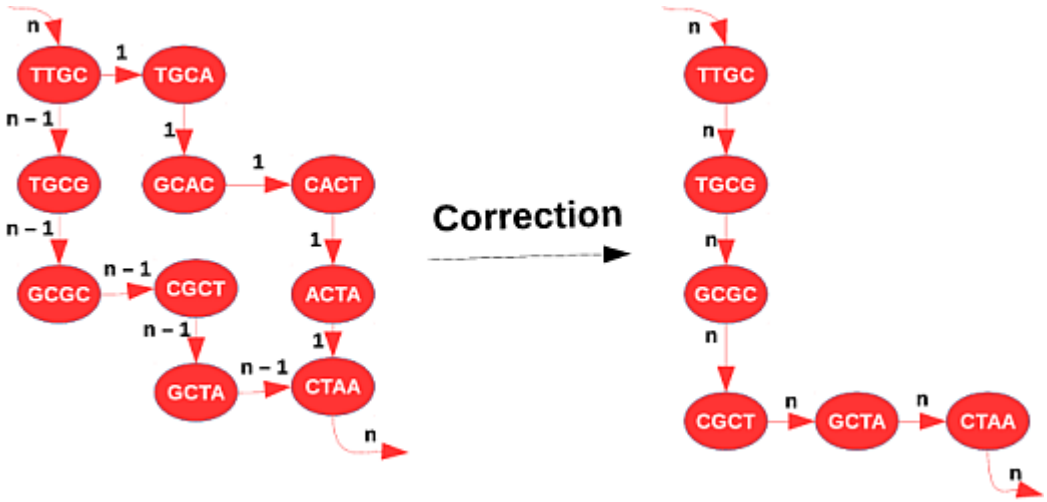

Figure 3.1: Simple examle of a read error detection by utilizing De Bruing graphs

3.2 K-mer Frequency Distribution

The method is based on the assumption that k-mer frequency distribution of an error-free read set has certain statistical properties. Specifically,it follows the Poisson distribution centered around the average k-mer coverage depth. Figure 3.2 shows the frequency distribution for an error-free read set and for a read set with error rate 1%.

As can be seen in the figure, errors lead to enrichment of unique and rare k-mers. The idea behind the correction algorithms based on this method is to transform the low-frequency k-mers into those with frequencies in the right place.

This approach is also used by the `fermi-lite`[11] software and is covered in the next section.

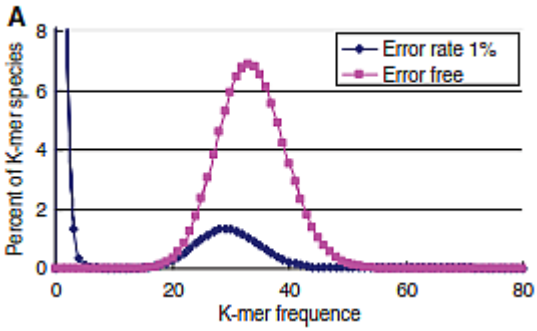

Figure 3.2: K-mer frequency distribution for errornous and error-free read sets

3.3 The Fermi-lite Approach

`Fermi-lite` is a standalone C library as well as a command-line tool for assembling Illumina short reads in regions from 100 bp to 10 million bp in size. It is largely a light-weight in-memory version of `fermikit`[9] without generating any

intermediate files[11]. Results of the assembly are not produced in the VCF format, but as a graph. Read error corrections are not the main goal of the project, although this step is definitely required for a successful assembly.

We have successfully extracted the error correction algorithm from the project. The implementation should work well on multiprocessor systems and trades performance over memory consumption. The algorithm proceeds in the following steps:

- **Preprocessing**. The input read sequences are divided into k-mers, k-mer frequencies are calculated.
- **Error correction**. The problem is reduced into a shortest path graph problem and is solved by a modified version of Dijkstra's algorithm.
- **Unique k-mer filtering**. Unique k-mers introduced during the error correction phase are removed from the read sequences.

3.3.1 Data Preprocessing

The main goal of the preprocessing phase is to compute frequencies for all k-mers found in the input read set. The frequencies are computed by inserting the k-mers into a k-mer table. During this phase, several terms related to k-mers and their occurrences are introduced:

- A k-mer occurrence is defined as *high quality* if quality of all bases covered by it is greater than certain threshold (set to 20 by default). If not all bases satisfy this condition, the occurrence is considered as *low quality*.
- A k-mer is considered *solid* if its frequency is greater than certain threshold.
- A k-mer is considered *unique* if it its frequency is zero.
- A k-mer is referred as *absent* if it its frequency is is below certain threshold.

K-mers are implemented as four 64-bit integers (Figure 3.3). Each base is represented by two bits. This limits the maximum k-mer size to 64 but allows effective implemtations of standard k-mer operations. The first 64-bit integer stores least signifficant bits of the k-mer bases, the second contains their most signifficant bits. The bases are stored in an opposite order — the first base resides in bits $k-1$, the second to $k-2$ and so on. The second two integers store the same content, but with different order — the first base is stored in bits 0, the second in bits 1 etc. Such a k-mer form vastes memory; only two 64-bit integers are required to hold all the necessary information. The error correction algorithm often takes advantage of the two-integer representation.

Such a represetantion allows quick appends or individual base changes. To append a base into the first two integers, they just need to be shifted by one to the left, ORed with the new base, and ANDed with $2^k - 1$ to set the unused bits to zero.

The k-mer table is actually a set of $2^{l_{pre}}$ khash tables. When a k-mer is being inserted or looked up, l_{pre} bits of its data are used to select the table and the rest serves as an input to the hash function. $l_{pre} = 20$ by default. This representation

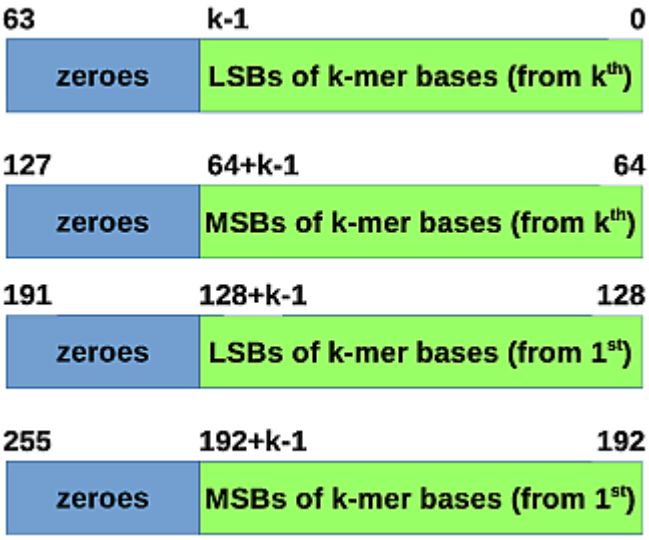

Figure 3.3: K-mer representation used by the `fermi-lite` project

of the k-mer table increases overall memory consumption, but has great impact on its performance in parallel environemnts.

The table uses 14 bits to track occurrences of each k-mer. Lower 8 bits count low quality occurrences, higher 6 bits are used by high quality ones. The counting stops on values of 255 and 63, no integer overflow happens. The table actually stores their hashes and occurrences, rather than full k-mers. Hence, distinct k-mers may be treated as one, since the table may use up to l_{pre} (20 by default) of k-mer content to choose the right khash table, and up to 50 bits may be stored as a key. That means, key colisions may be avoided if the k-mer size drops below 35 (the table actually has a different hash function for k-mers size below 33).

After the insertion stage, a k-mer frequency distribution is computed individually for low quality and high quality occurrences. The most common frequency is named *mode* and is used during the second and third pahse of the algorithm.

All phases of the algorithm are partialy driven by its parameters. Table 3.1 provides a short description of them.

3.3.2 Error Correction

The error correction is performed separately for each read sequence. The correction problem is transformed into a shortest-path search in a layered oriented graph. Vertices represent individual k-mers, edges connect adjacent ones and their weights reflect the cost of transforming one k-mer to another. The weight function is defined by formula

$$w(k_i) = w_{absent\_high} * nhq(k_i) + w_{absent} * nlq(k_i) + w_{ec} * ec(k_i) + w_{ec\_high} * bq(k_i)$$

$$nhq(k_i) = \begin{cases} 0, & k_i \text{ is a high quality k-mer} \\ 1, & \text{otherwise} \end{cases}$$

$$nlq(k_i) = \begin{cases} 0, & k_i \text{ is a low quality k-mer} \\ 1, & \text{otherwise} \end{cases}$$

$$ec(k_i) = \begin{cases} 0, & \text{the edge ending in } k_i \text{ does not introduce an error correction} \\ 1, & \text{otherwise} \end{cases}$$

$$bq(k_i) = \begin{cases} 0, & \text{the last base of } k_i \text{ has quality above } \texttt{q} \\ 1, & \text{otherwise} \end{cases}$$

Dijkstra's search algorithm is used and its main loop can be decomposed into the following steps:

- Retrieve the vertex with lowest price, and its k-mer from the heap.
- by separately appending `A`, `C`, `G`, and `T` to the k-mer, touch the adjacent vertices on the next layer and compute the cost of their connections.
- Insert the newly created vertices into the heap.

Each path from the starting vertex to a vertex in the last layer represents one possible corrected part of the read sequence. Four such paths are computed. The path computation stops if a gap greater than `max_path_diff` is detected in their costs.

3.3.3 Unique k-mer Filtering

The error correction phase may produce unique k-mers which, as Figure 3.2 indicates, are not desirable. The `fermi-lite` library attempts to get rid of such k-mers. Each corrected read sequence is processed separately (and in parallel with others).

At first, the longest k-mer sequence covered by non-unique k-mers is found. Denote its length, in k-mers, as n and the read sequence length as l. Then, the read sequence between the read start and the first base covered by the found k-mer sequence is removed from the read. If the read is covered only by non-unique k-mers, nothing is removed, since the k-mer sequence covers the whole read. However, if the following condition holds:

$$\frac{n + k - 1}{l} < \text{min\_trim\_frac}$$

the read is removed from the read set. This case includes also zero-length k-mer sequences that appear when the read contains unique k-mers only.

Table 3.1: Parameters of the `fermi-lite` algorithm

Parameter	Default value	Description
k	-	K-mer size. By default, this value is set to the base-two logarithm of the total number of bases.
q	20	Base quality threshold used to recongize high quality content from low quality one. $P(error) = 10^{-\frac{q}{10}}$
min_cov	4	Mimimum frequency for solid k-mers.
win_multi_ec	10	
l_{pre}	20	Number of khash tables in the k-mer table, defined as 2^{l_pre}
min_trim_frac	0.8	Defines how long the sequence of solid k-mers must be in order not to remove the read during unique k-mer filtering.
w_{ec}	1	Participates in the weighing function used in the error correction step.
w_{ec_high}	7	Participates in the weighing function used in the error correction step.
w_{absent}	3	Participates in the weighing function used in the error correction step.
w_{absent_high}	1	Participates in the weighing function used in the error correction step.
max_path_diff	15	Participates in the weighing function used in the error correction step.

4. Results

When developing a new algorithm, an important part is testing and evaluation against existing solutions. This chapter describes this stage, informing about a data set used to debug and improve our solution, and the method of comparison with other solutions, such as fermikit and GATK. The last part of the chapter covers certain variants that proved to be interesting when examined by our algorithm.

4.1 Test Data Set

chr1:1-40,000,000 region of the high-coverage data from the 1000 Genome Project. In addition to the input reads [1], also variants called by fermikit and GATK are available in form of VCF files [3]. The VCF files were used to compare the accuracy of the algorithm. In order to be able to make meaningful comparisons and to avoid remapping of all reads, we used the same version of the reference genome which was used also by the 1000 Genome Project (GRCh37 [2]).

The test read set consists of 12,475,011 reads with lengh of 151 bases. Figure 4.1 shows k-mer frequency distribution of the set with k-mer size of 21 bases. The shape of the graph, when compared to Figure 3.2 suggests that the set indeed contains read errors, therefore the error correction step was applied.

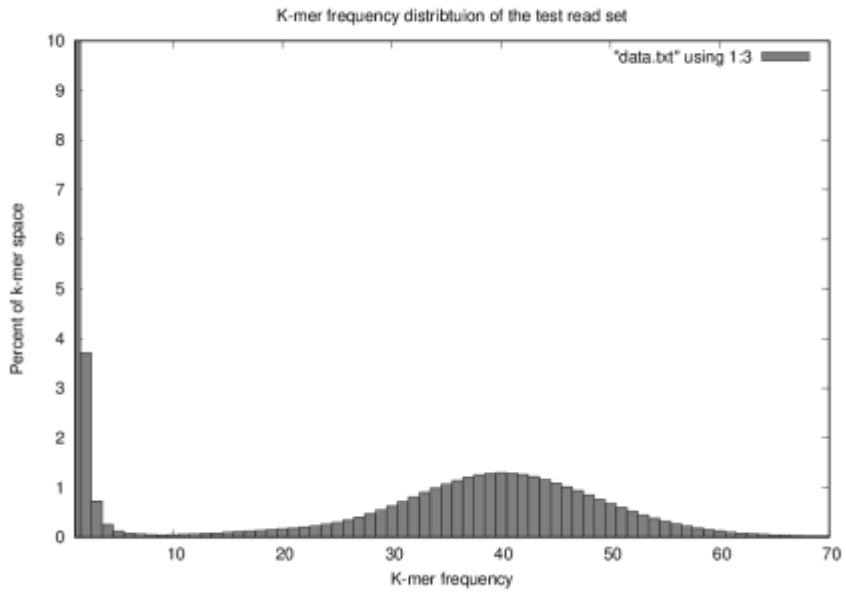

Figure 4.1: K-mer frequency distrubtion of the raw input read set

As Table 4.1 indicates, the error correction process removed and shortened a signifficant proportion of the reads. Approximately 21% of the input reads was subject to repairs. Figure 4.2 shows a distribution of the number of repaired bases per read, not including effects of read trimming.

Table 4.1: Statistics related to error correction of the test data set

Category	Value	Percentage
Total reads	12,475,011	-
Removed	64,653	0.52% of all reads.
Shortened	944	0.0076% of all reads
Total bases	1,880,123,991	-
Bases repaired	5,098,764	0.27% of all bases

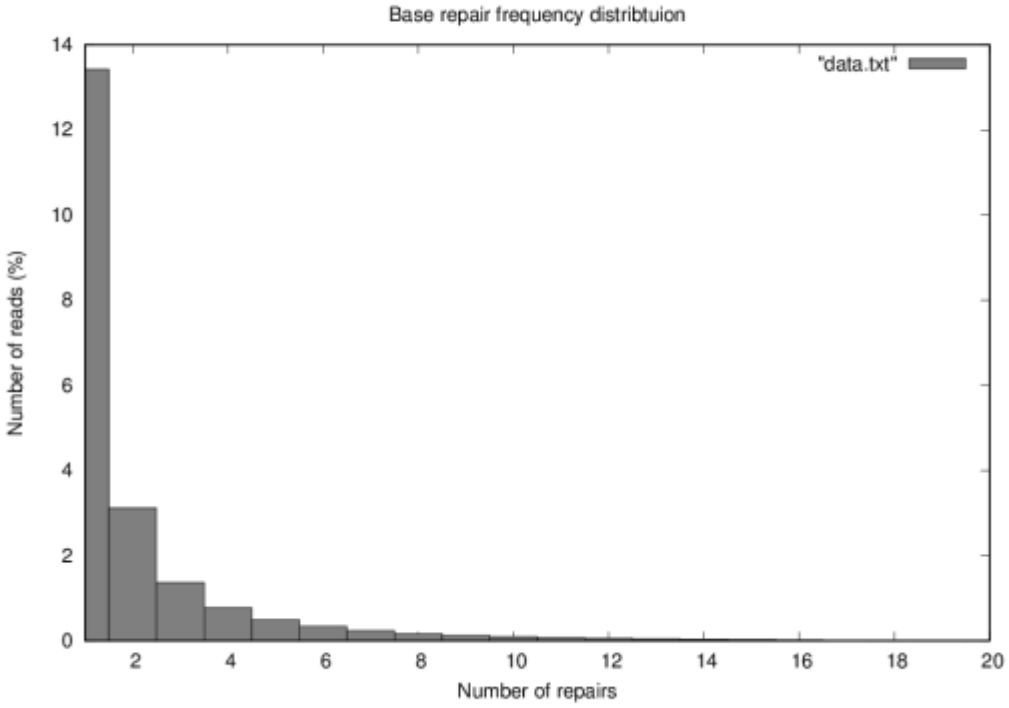

Figure 4.2: Distribution of a number of repaired bases in a single read

Figure 4.3 shows the k-mer frequency distribution of the corrected read set. Although still not perfect, the distribution of corrected reads resembles the theoretical distribution of an ideal error-free set more than the distribution of raw reads.

As described in Section 2.1, not all input reads, even from the corrected set, can be processed by our algorithm. Table 4.2 summarizes numbers of reads discarded for various reasons. The preprocessing phase removed nearly one fifth of the corrected data set (18.99%). Most of the reads were removed due to being possible duplicates (87.65%). Quite a large portion of reads were not accepted because of their low mapping quality (12.97%). Also, about 3% of all the reads were shortened in order to remove soft-clipped regions.

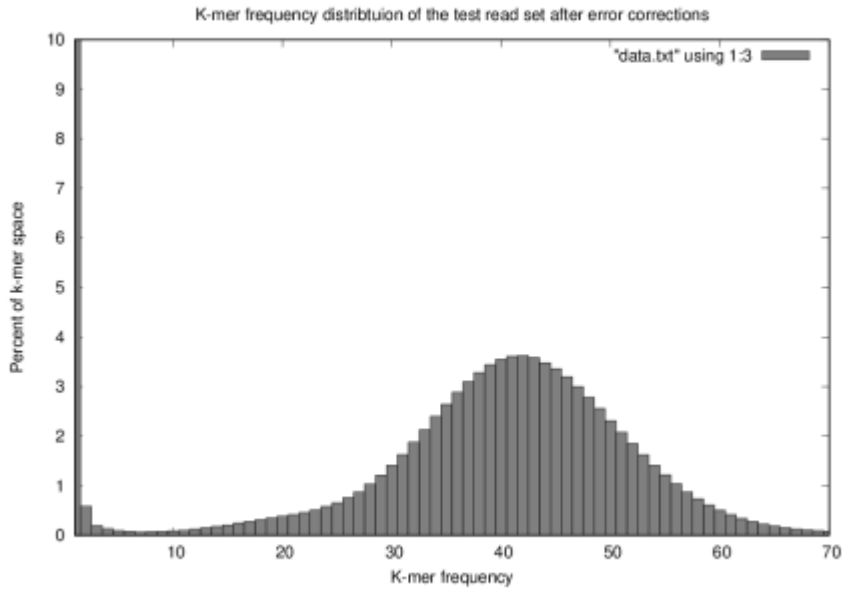

Figure 4.3: K-mer frequency distrubtion of the corrected read set

Table 4.2: Categories of reads present within the corrected test data set

Name	Value	Percentage
Total reads	12,410,475	-
Bad reads	2,357,002	18.99% of all reads
Low MAPQ	305,588	12.97% of bad reads
Unmapped	5,020	0.21% of bad reads
Supplementary	33,621	1.43% of bad reads
Duplicate	2,065,795	87.65% of bad reads
Soft-clipped	305,209	3.04% of accempted reads

4.2 Quality Evaluation

In order to evaluate the algorithm, the generated VCF files were compared to those generated by the following variant calling toolchains:

- **GATK**. These VCFs were taken as reference points, since the method used by GATK should be very similar to our algorithm. Hence, results of all algorithms were compared against them.
- **Fermikit**. Unlike GATK and our algorithm, Fermikit's assembly algorithm is based on the OLC concept.
- **Fermikit on regions**. This is a combination of the OLC approach used by Fermikit and the short region one adopted by our algorithm (and also by GATK). The fermikit assembly algorithm was run on exactly the same regions as our algorithm. The aim of this test case was to force the Fermikit to minimize differences of outputs generated by DBG and OLC algorithms.

- **samtools mpileup, bcftools call**. A traditional variant caller implemented by SAMtools and BCFtools packages [5].

Results generated by these algorithms were compared to those of GATK using the tool **rtgeval**. Rtleval is a wrapper for RTG's vcfeval, a sophisticated open source variant comparison tool developed by Realtime Genomics. It simplifies the use of vcfeval and potentially helps to get consistent results given VCFs produced by different variant callers [4].

Rtgeval accepts two VCF files on input: a test set and a truth set. The truth set is a VCF file generated by a reference algorithm, in our case GATK. The test set VCF is produced by the algorithm to be tested. The evaluation is done separately for SNPs and indels and each variant is sorted into one of three categories:

- **True positive (TP)**. The variant is present in both sets.
- **False negative (FN)**. It is present in the truth set only.
- **False positive (FP)**. It can be found only in the test set.

Rtgeval can compare VCF files in three different modes: positional, allelic and genotypic. The positional mode is intuitive; the tool is determining whether the same varaints are present at approximately the same position inside both test and truth set. If the positions difference does not exceed 10 bases, the variants are considered true positives. Otherwise, either false negative, or fale positive is reported. In allelic mode, the tool focuses on biallelic variants and evaluates whether they are correctly detected by both tested algorithms. The genotyping mode compares genotype and phasing information of the variants.

4.2.1 Positional

Table 4.3: Positional comparison of results generated by our algorithm

/	Fermikit	Fermikit (regions)	mpileup	Our algorithm
SNP TP	45,241 (89.3%)	47,650 (94%)	49,201 (97.1%)	48,117 (94.96%)
SNP FN	5,432 (10.7%)	3,043 (6%)	1,472 (2.9%)	2,556 (5.04%)
SNP FP	385 (0.76%)	1,441 (2.84%)	2,090 (4.12%)	803 (1.58%)
INDEL TP	7,853 (71.6%)	9,802 (89.4%)	8,707 (79.39%)	9,468 (86.43%)
INDEL FN	3,114 (28.4%)	1,365 (10.6%)	2,260 (20.61%)	1,499 (13.57%)
INDEL FP	250 (2.28%)	835 (7.61%)	1,412 (12.95%)	1,242 (11.32%)

- describe algorithms and software to evaluate the results (fermikit, fermikit run at individual regions, GATK, mpileup of samtools, rtgeval for evaluation),
- Show the position-based and genotype results of rtgeval.
- describe interesting variants (variants that are not found by other software, explain some false negatives, demonstrate that graph optimizations actually revealed some variants...).

References

[1] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/pilot2_high_cov_GRCh37_bams/data/

[2] $http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/human\_g1k\_v37.fast$

[3] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/

[4] https://github.com/lh3/rtgeval

[5] http://www.htslib.org/

[6] https://gatkforums.broadinstitute.org/gatk/discussion/4146/hc-step-2-local-re-assembly-and-haplotype-determination

[7] Compeau, Phillip C., Pevzner, Pavel A., Tesler, Glenn: How to apply de Bruijn graphs to genome assembly Nature Biotechnology, Volume 29, Number 11, November 2011

[8] Li, Zhenyu, Chen, Yanxiang, Mu, Desheng, Yuan, Jianying, Shi, Yujian, Zhang, Hao, Gan, Jun, Li, Nan, Hu, Xuesong, Binghang Liu, Yang, Bicheng, Fan Wu: Comparison of two major classes of assembly algorithms: overlap-major-consensus and de-bruijn graph Advance Access, 19 December 2011

[9] Li, Heng: FermiKit: assembly-based variant calling for Illumina resequencing data Cornell University Library, 24. 4. 2015 https://arxiv.org/abs/1504.06574

[10] Sequence Alignment / Map Format Specification The SAM/BAM Format Specification Working Group 28 Aug 2015 https://github.com/samtools/hts-specs

[11] https://github.com/lh3/fermi-lite

List of Figures

List of Tables