View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Helsingin yliopiston digitaalinen arkisto

Identification of variant compositions in related strains without reference

Mikko Rautiainen

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, December 28, 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Mikko Rautiainen

Työn nimi — Arbetets titel — Title

Identification of variant compositions in related strains without reference

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	December 28, 2015	28

Tiivistelmä — Referat — Abstract

The genomes of all animals, plants and fungi are organized into *chromosomes*, which contain a sequence of the four nucleotides A, T, C and G. Chromosomes are further arranged into *homologous groups*, where two or more chromosomes are almost exact copies of each others. Species whose homologous groups contain pairs of chromosomes, such as humans, are called *diploid*. Species with more than two chromosomes in a homologous group are called *polyploid*.

DNA sequencing technologies do not read an entire chromosome from end to end. Instead, the results of DNA sequencing are small sequences called *reads* or *fragments*. Due to the difficulty of assembling the full genome from reads, a reference genome is not always available for a species. For this reason, *reference-free* algorithms which do not use a reference genome are useful for poorly understood genomes.

A common variation between the chromosomes in a homologous group is the *single nucleotide polymorhpism* (SNP), where the sequences differ by exactly one nucleotide at a location. Genomes are sometimes represented as a consensus sequence and a list of SNPs, without information about which variants of a SNP belong in which chromosome. This discards useful information about the genome.

Identification of *variant compositions* aims to correct this. A variant composition is an assignment of the variants in a SNP to the chromosomes. Identification of variant compositions is closely related to *haplotype assembly*, which aims to solve the sequences of an organism's chromosomes, and *variant detection*, which aims to solve the sequences of a population of bacterial strains and their frequencies in the population.

This thesis extends an existing exact algorithm for haplotype assembly of diploid species (Patterson et al, 2014) to the reference-free, polyploid case. Since haplotype assembly is NP-hard, the algorithm's time complexity is exponential to the maximum *coverage* of the input. Coverage means the number of reads which cover a position in the genome. Lowering the coverage of the input is necessary. Since the algorithm does not use a reference genome, the reads must be ordered in some other way. Ordering reads is an NP-hard problem and the technique of *matrix banding* (Junttila, PhD thesis, 2011) is used to approxiately order the reads to lower coverage. Some heuristics are also presented for merging reads. Experiments with simulated data show that the algorithm's accuracy is promising. The source code of the implementation and scripts for running the experiments are available online at `https://github.com/maickrau/haplotyper`.

ACM Computing Classification System (CCS):
Computational genomics, Bioinformatics

Avainsanat — Nyckelord — Keywords
Variant compositions, Haplotype assembly, Variant detection, Exponential time algorithms

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

The DNA of all living beings is composed of the nucleotides A, T, C and G. An organism's *genome* is a sequence or sequences of these four letters. In eukaryotic species, which includes all animals, plants and fungi, the genome is composed of *chromosomes*, each of which is a sequence of nucleotides. The chromosomes are further arranged into groups of *homologous* chromosomes, where two or more chromosomes are almost exact copies of each others. For example, humans' genomes are arranged into pairs of homologous chromosomes, one of which is inherited from a person's mother and the other from the father. Species with pairs of homologous chromosomes are called *diploid*. Species with groups of more than two homologous chromosomes, for example some potato species which have groups of 3 or 4 chromosomes [Cor62], are called *polyploid*. The sequence of a chromosome is called a *haplotype sequence*, and a chromosome in a homologous group is sometimes called a *haplotype*.

Homologous chromosomes are typically very similar to each others with minor variation. A common variation is the *single nucleotide polymorphism* (SNP), where the haplotype sequences vary by exactly one nucleotide at some location, either as a substitution, having a different letter in the chromosomes, or indels (insertion-deletion), where a letter has been inserted or deleted from one of the chromosomes. SNPs can be either *heterozygous*, where all of the haplotypes have a different variant (nucleotide), or *homozygous*, where some haplotypes have the same variant. Figure 1 shows an example of SNPs. Two chromosomes from an organism are compared to a *reference genome*, which is the sequnce of one of the chromosomes from some other individual of the same species. The sequences have three SNPs. At the start there is a heterozygous SNP with a substitution, at the middle there is a homozygous SNP with substitutions, and at the end there is a heterozygous SNP with an indel and a substitution.

haplotype 1	ACTAACGCTGAATGAGACTAGT
haplotype 2	ACTCACGCTGAATGAGAC-AGT
reference	ACTAACGCAGAATGAGACGAGT

Figure 1: Heterozygous and homozygous SNPs

Assembling an organism's full genome is usually an arduous task. Current state-of-the-art sequencing technologies, called *next-generation sequencing*, read many small fragments of an organism's genome. The results of next-generation sequencing are *reads*, also called *fragments*, short sequences of DNA from anywhere in the organism's genome. Reads only contain the DNA that was sequenced, in particular a read does not contain the information about where it was sequenced from. The length and accuracy of reads varies based on the specific technology used; reads with a length of a few dozen base

pairs are cheap, common and accurate. Recently technologies for cheaply sequencing longer reads of thousands of base pairs have been published [SJ08] [EFG+09] [FWL+10] but these technologies have higher error rates. Even at these lengths, reads are tiny compared to their genomes; for comparison, the human genome is billions of base pairs long.

Due to the difficulty of assembling a genome from sequenced reads, most species lack a reference genome. A reference genome may also be incomplete or otherwise unusable for various reasons. Polyploid genomes are in particular more difficult to sequence than diploid sequences. Reference genomes also may not contain the organism's entire genome for homologous chromosomes. Another similar case is a population of closely related bacteria. A reference genome may not be available given the sheer amount of bacterial species. For this reason, *reference-free* algorithms which do not require a reference genome are useful for poorly understood genomes.

Since the chromosomes or strains are very similar, they can be represented as one sequence and a list of positions where SNPs are found. Figure 2 shows an example of this. Two sequences have a common consensus with two SNPs. However, the consensus does not tell whether the two A's belong in the same sequence or not. SNPs have different effects on an organism's phenotype depending on which other variants are found in the same chromosome [SST+01] [TBT+11], thus having just a list of SNPs misses *phase* information, which describes which variants occur in the same chromosomes or strains.

$$\mathrm{A\,C\,T}\,\begin{matrix}\mathrm{A}\\ \mathrm{T}\end{matrix}\,\mathrm{A\,C\,G\,A\,T\,G\,A\,G\,A\,C}\,\begin{matrix}\mathrm{A}\\ \mathrm{T}\end{matrix}\,\mathrm{A\,G\,T}$$

Figure 2: A consensus sequence with SNPs

Identification of *variant compositions* aims to correct this. A variant composition is an assignment of the variants to the chromosomes or strains, with the goal of identifying which variants belong in which chromosomes or strains. A closely related problem is *haplotype assembly*, which seeks to solve an organism's complete genome for each chromosome. The connection is that both problems seek to conclusively assign reads into the chromosomes. Another closely related problem is *variant detection*, which aims to solve the sequences of related strains and their frequencies in a population. The connection to variant detection is that the variant composition can be used in solving the sequences.

The process of assigning the variants of a SNP to the chromosomes is called *phasing*. This thesis generalizes an existing exact phasing algorithm for diploid genomes [PMP+14] to the reference-free, polyploid case. The phasing algorithm's time and space complexities are both exponential to the maximum *coverage* of the input. Coverage means the number of reads that

cover a position. Figure 3 shows an example. The long, solid line represents the actual genome, which is unknown, and the short solid lines represent the reads and their location in the genome. The position at the first dashed line has a coverage of three, since there are three reads that cover that position. Similarly, the second dashed line has a coverage of five.

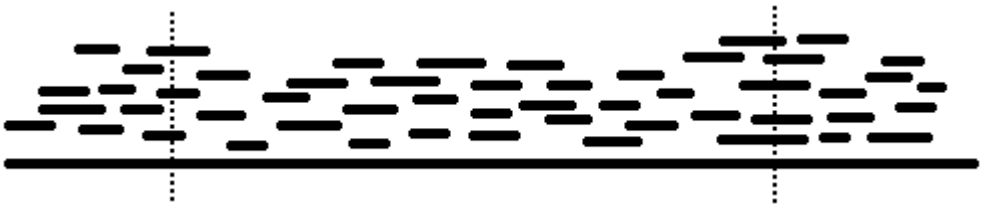

Figure 3: Coverage

Because the algorithm is exponential to coverage, reducing the coverage is necessary. Since a reference genome is not used, the reads cannot be ordered by aligning them to a reference genome. However, having the reads out of order raises the coverage, so ordering them is necessary. The technique of *matrix banding* [Jun11] is used to order the reads. After banding, some heuristics are used to merge the reads to lower coverage. Typical values for coverage after reduction are under 15, and the experiments use an input coverage of 80 before reduction.

2 Previous work

Identification of variant compositions is closely related to the *haplotype assembly problem*. Haplotype assembly seeks to solve the sequences of an organism's haplotype. The connection is that the variant composition is a list of variants for each haplotype at the SNPs, which can be used to calculate the haplotype sequences. Haplotype assembly problem has multiple similar formulations [LSLI02]. These formulations are based on building a *haplotyping matrix*, where the reads are rows and SNPs are columns, and finding a way to partition the rows into chromosomes such that all reads in a partition agree on the variant at each SNP. A haplotyping matrix does not always have such a partition. In that case, the problem is finding a way to modify the matrix the least to make such a partition possible. Lippert et al. [LSLI02] suggest three formulations: the *minimum fragment removal* aims to remove the least amount of reads, the *minimum SNP removal* aims to remove the least amount of SNPs, and the *minimum error correction* aims to flip the least amount of cells in the matrix. Greenberg et al. [GHL04] further suggested the *weighted minimum error correction*, which is the minimum error correction formulation extended such that flipping a cell can have a

variable cost. Haplotype assembly is NP-hard both in general [LSLI02] and also in the case when there are no gaps between reads [CvIKT05].

Identification of variant compositions is also related to *variant detection*. Variant detection seeks to solve the *quasispecies spectrum reconstruction problem* [ATM+11], which is reconstructing the sequences of the strains in a population and their frequencies. The connection is that the variant compositions can be used for the first part, reconstructing the sequences. Algorithms for variant detection are available [PMAP13] [ATM+11] [JHJ08] [ZBEB11].

Haplotype assembly for diploid organisms, organisms only having two chromosomes, has many algorithms available, both exact [CDW13] [HCP+10] [DCW13] [PMP+14] and probabilistic [BAMR13] [Kul14]. Solving the haplotype assembly for polyploid organisms, organisms having more than two chromosomes, is a more recent field of research. Both exact [NGB+08] [DV15] and probabilistic algorithms [AI13] [BYPB14] [SWBC08] are available.

Deng et al. [DCW13] published in 2013 an exact algorithm for solving the diploid haplotype assembly problem with the minimum error correction formulation. The algorithm has a time complexity of $O(|S|2^C C)$ where $|S|$ is the number of SNPs and C is the maximum coverage. In 2014, Patterson et al. [PMP+14] extended the algorithm to the weighted minimum error correction formulation, and improved the time complexity to $O(|S|2^{C-1})$. This thesis further extends the algorithm to the polyploid case with a time complexity of $O(|S|C\frac{k^C}{k!})$, where k is the number of chromosomes. The algorithm uses only the reads and requires no reference genome.

For this implementation, the DiscoSNP program [URL+14] is used for detecting SNPs from the reads. DiscoSNP works by building a de Bruijn graph from the reads and finding the SNPs from it.

Since the algorithm does not use a reference genome, the locations of the SNPs and reads are unknown. *Matrix banding* [Jun11] is used to find an approximate ordering of the SNPs and reads.

3 Extending haplotyping to multiple strains

The pipeline is roughly divided into three stages. In the preprocessing stage, a *haplotyping matrix* is built from the reads. The second stage manipulates the haplotyping matrix to lower coverage. The third stage, the *phasing* stage, assigns the reads into strains.

3.1 Preprocessing

The purpose of this stage is to produce the matrix used in the later parts of the algorithm.

The first step for constructing the haplotyping matrix is to detect the SNPs in the strains. The DiscoSNP [URL+14] program is used for SNP

detection. DiscoSNP works by building a de Bruijn graph from the reads. SNPs are detected by finding parts of the graph where a path branches to two paths, which then join a short distance later. These paths are extended forwards and backwards until the next extending node is ambiguous. The output of DiscoSNP are pairs of sequences around the SNPs, which correspond to the two paths in the graph.

The SNP sequences discovered by DiscoSNP are then matched with the reads to produce a *haplotyping matrix*. The haplotyping matrix contains the SNPs as columns, reads as rows, and the value of a cell is either the read's nucleotide at the SNP or a marker − that the read does not cover that SNP. In the first case, the read is said to *support* a certain variant, that is the nucleotide, at the SNP. The haplotyping matrix is then said to have a variant at that cell.

The preprocessing also produces a *weight matrix*, which is similar to the haplotyping matrix except the value determines how certain the nucleotide is, with higher weights meaning more certain. The values are used in the later stages. The values are initialized with 1 where the haplotyping matrix has a variant and 0 otherwise.

Since the phasing algorithm's time and space complexity are exponential to coverage, it is necessary to reduce coverage before the algorithm can be used. Two kinds of coverage are recognized. *Essential coverage* is coverage where a read supports some variant for a SNP. *Accidental coverage* is coverage where a read does not directly support a SNP, but supports SNPs both before and after it. A read is said to either *essentially cover* or *accidentally cover* a SNP in these cases. Figure 4 shows these coverages. SNP number 2 has essential coverage 3, from reads 3, 5 and 6, and accidental coverage 2, from reads 1 and 4. Essential coverage cannot be reduced by permuting the SNPs, but accidental coverage can. Different strategies are used for the two kinds of coverage.

	1	2	3
1	A	-	T
2	-	-	G
3	-	A	T
4	A	-	G
5	-	C	G
6	T	C	-

Figure 4: Essential and accidental coverage

3.2 Reducing essential coverage

To reduce essential coverage, some supports must be removed. The program does this by merging similar reads together. Reads are merged by a greedy process. First, the SNP with the highest essential coverage is found. Then, candidate reads for merging are those rows which essentially cover the SNP. Two most similar candidate reads are merged to produce a merged read, and the two original reads are discarded. This is repeated until essential coverage is acceptably low. In the implementation, an essential coverage of 10 is considered acceptably low.

Distance between reads is given by the following equation where F is the haplotyping matrix and W is the weight matrix:

$$dist(r_i, r_j) = \sum_{x \in [1..|S|]} \begin{cases} 0 & \text{if } F(i,x) = F(j,x), \\ W(i,x) + W(j,x) & \text{if } F(i,x) = - \text{ xor } F(j,x) = -, \\ 2 * (W(i,x) + W(j,x)) & \text{if } F(i,x) \neq F(j,x) \end{cases}$$

In other words, SNP locations where the reads agree on a variant contribute a distance of 0. Locations where one read has a variant but the other doesn't, contribute the weight of the support that has a variant. Locations where the reads disagree contribute twice the weight of both supports.

Merging reads i and j produces a new read i' with

$$W(i', x) = W(i, x) + W(j, x)$$

$$F(i', x) = \begin{cases} F(i,x) & \text{when } W(i,x) > W(j,x) \\ F(j,x) & \text{otherwise} \end{cases}$$

The resulting read has a variant in every SNP where either of the merged reads had. The variant is taken from the read which has a higher weight at that SNP.

Usually, the variants are equal, however they may not always be. This is problematic since reads are merged two at a time, and the merging process discards information about the lower-weight reads variants. When merging multiple reads together, the final merged read's variants depend on the order in which the reads were merged.

The merging process's accuracy depends greatly on read length. This is elaborated further in the experiments section.

3.3 Reducing accidental coverage

The main approach to reducing accidental coverage is *matrix banding*. All of the techniques used here are based on Junttila's dissertation [Jun11]. The haplotyping matrix is treated as a binary matrix, where cells that have a

variant are 1, and cells without a variant are 0. The rows and columns are then permutated to bring the ones together in the matrix. The same permutation is also applied to the weight matrix. Figures 5 and 6 show an example of a matrix before and after banding. Cells with a variant are colored black, and cells without are white. The ideal output would be a solid diagonal band.

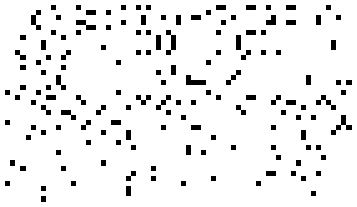

Figure 5: A binary matrix before banding

Figure 6: A binary matrix after banding

For evaluating a banding's score, the implementation uses a function based on the total coverage of each SNP:

$$score(H) = \sum_{i \in [1,|S|]} 2^{coverage(i)}$$

This score function is proportional to how much time and memory the phasing algorithm needs to run on the matrix H with two strains. A lower score corresponds to a shorter run time.

The first step of matrix banding finds an approximate *consecutive-ones property* (C1P) on the haplotyping matrix. The C1P means that there is a permutation of the matrix's columns such that in any row, the ones are in consecutive positions without zeroes between them. Depending on the matrix, C1P may not always exist, and even when it does finding it is NP-hard. The implementation uses *spectral sorting* to find an approximate C1P. Spectral sorting works by building a distance matrix between each row, finding the eigenvector associated with the second smallest eigenvalue of the matrix, and sorting the columns based on the values of the eigenvector. The solution depends on the row distance function used for building the

distance matrix. The dissertation defines three functions, the *dot product*, *correlation similarity* and *Jaccard similarity*. The implementation tries all three measures and selects the best scoring result.

The second step of matrix banding uses *barycentric sorting*. For each row, a centerpoint is calculated as the average position of the ones. The rows are then sorted based on the centerpoints. The same operation is then performed on the columns. This is repeated for a certain number of iterations and the best iteration's matrix is selected.

The last step in matrix banding uses *simulated annealing*. The implementation uses a different method for selecting a neighbor state than those described in the dissertation. Since the energy function does not depend on the order of the rows, only columns need to be considered in the neighbor state selection. The implementation picks a small number of columns to permutate. First, a starting column is picked either randomly or by selecting the SNP with the highest coverage. Then, more columns are picked randomly. After picking a column, there is a 20% chance of continuing, and an 80% chance of picking another column. These probabilities were chosen somewhat arbitrarily; probabilities of 10% and 30% continuation chance were tried but there was no significant difference in their performance. After the columns are selected, they are permutated randomly.

The second approach to reducing accidental coverage is removing outliers. This approach is only used if matrix banding does not reduce accidental coverage to an acceptably low level. To remove an outlier, the SNP with the highest accidental coverage is found. Each row that accidentally covers the SNP is a candidate for outlier removal. To make a read not accidentally cover the SNP, all of its supports must be removed from either the left side or right side of the SNP. The read with fewest supports in either side is picked, and then those supports are removed from the haplotyping and weight matrices. The implementation treats an accidental coverage of 10 as acceptably low.

Finally, after both essential and accidental coverage are reduced separately, the total coverage might still be too high. In this case, rows are merged from the SNP with the highest total coverage until total coverage is low enough. The heuristic for selecting rows to merge is the same as when reducing essential coverage, except all rows which cover a SNP are candidates for merging instead of just those which essentially cover it. For the total coverage, the implementation treats 14 as acceptably low.

3.4 Phasing

The phasing algorithm is a generalization of the algorithm by Patterson et al [PMP+14]. The difference is that this algorithm works with an arbitrary number of strains instead of two. The output of the phasing algorithm is an assignment of the reads into the strains.

A *partition* is an assignment of the reads, or a subset of reads, to the

strains. Partitions are stored as a *partition vector*, which describes for each read either the strain where it is assigned, or a marker that this read is not assigned to any strain. For example, $\{1, -, 2, 1, 3, -\}$ is a partition of four reads into three strains, where reads 1 and 4 are in the same strain and reads 2 and 6 are not assigned to any strain. The example partition is over reads 1, 3, 4, and 5. Representing the partitions this way simplifies some operations in the algorithm.

The overlap of two partitions is the indices where they both have a value. For example, the overlap of $\{1, 2, 2, -, -\}$ and $\{-, 1, 2, 3, 3\}$ is indices 2 and 3.

The *unpermutated form* of a partition P is marked as $u(P)$. The unpermutated form means that the strains are re-labeled according to the order in which they appear in the partition vector. For example, the partition vector $\{3, 1, 3, 2, 2, 1, 4\}$ has the unpermutated form $\{1, 2, 1, 3, 3, 2, 4\}$.

This operation of re-labeling the strains is called a *renumbering*. A renumbering is a bijection of strain numbers. A *renumbering vector* describes how the strains are re-labeled. Given an original partition P and a renumbering vector R, the renumbered partition P' is given by

$$P'_i = R_{P_i}$$

The example mentioned above has a renumbering vector of $\{2, 3, 1, 4\}$. For example, strain number 2 in the original partition is replaced with 3 in the renumbered partition. Changing a partition to its unpermutated form is one example of a renumbering but others are also used in the algorithm.

Two partitions are equivalent if and only if their unpermutated forms are equal. This means that both partitions assign the reads into same sets. Only partitions in the unpermutated form need to be considered in the algorithm since all other partitions are permutations of some unpermutated form.

The set of *active reads* for a SNP is the reads that cover it, either essentially or accidentally. The notation $\alpha(i)$ is used to represent active reads for SNP number i.

The set of all partitions over a set of reads r is marked as $Par(r)$. Note that this set does not actually have all partitions, only all unpermutated forms. This cuts the number of partitions from $k^{|r|}$ to $\frac{k^{|r|}}{k!}$.

A partition P_1 *extends* partition P_2 if and only if their unpermutated forms over the overlapping area are equal. The notation $Ext(P_1, P_2)$ is used to mark this. The set of partitions that extend a partition is said to be its *extensions*.

A partition is said to be *conflict-free* if all reads in a strain assign the same variant at each SNP. For example, the haplotyping matrix in figure 7 is conflict-free for the partition vector $\{1, 2, 1, 2\}$, but not for the partition vector $\{1, 2, 2, 2\}$. In the conflicting example, the cell at read 3 and SNP 2 is said to be in conflict since it is different from the consensus variant of strain

2 for SNP 2.

	1	2
1	C	T
2	C	C
3	C	T
4	C	C

Figure 7: A haplotyping matrix

In practice, the haplotyping matrix rarely has a conflict-free partition. Instead, the algorithm finds the partition closest to a conflict-free partition, with the distance being the total weight of the conflicting cells in the haplotyping matrix. For a partition P, haplotyping matrix F, weight matrix W, strain s, variant $v \in \{A, T, C, G\}$, and a SNP number i, define a cost function

$$\delta(P, s, v, i) = \sum_{x|P_i=s \wedge F(x,i) \neq v} W(x, i)$$

The function describes the cost for assigning the strain s to have variant v at SNP i. Using this, define the *partition cost function*

$$\Delta(P, i) = \sum_{s \in [1,k]} \min_{v \in \{A,T,C,G\}} \delta(P, s, v, i)$$

as the cost of partition P at SNP i. This cost function means the minimum cost to make the partition conflict-free. These functions are generalizations of the cost functions described in [PMP+14], and are equal to them in the case of two strains.

The algorithm is a dynamic programming algorithm that goes through the SNPs one by one. The table C contains the best scores for a partition at any SNP; element $C(i, x)$ is the best score at SNP i for partition x. At the first SNP, the table is initialized with

$$C(1, x) = \Delta(x, 1), \forall x \in Par(\alpha(1))$$

At every SNP other than the first, the table is calculated by

$$C(i, x) = \Delta(x, i) + \min_{y \in Par(\alpha(i-1)), Ext(x,y)} C(i - 1, y)$$

This means that the value of the current partition is the cost for the partition, plus the cost of the best-scoring partition of the previous SNP that the current partition extends.

To get the final result, the table C can be backtraced and the partitions at each SNP must be merged. However, the implementation does not actually

get the result by backtracing; instead the optimal partition over all reads so far is stored when going through the SNPs, and when moving to the next SNP, the partitions at the current SNP are merged with the partitions over all the reads so far.

When two partitions are merged, one of them must usually be renumbered. For example, the partition $\{-,-,1,2,1,1,3\}$ extends the partition $\{1,2,3,4,3,-,-\}$, since the unpermutated form of the overlapping part is $\{-,-,1,2,1,-,-\}$ for both partitions. The algorithm renumbers the rightmost partition to correspond to the leftmost partition's strain numbers. The renumbering vector is constructed by first assigning the overlapping part. Having a left partition P, a right partition Q and a set of indices for the overlapping part I, the renumbering vector R is given by the equation

$$R_{P_x} = Q_x, x \in I$$

This equation only assigns the renumbering vector for the strains which appear in the overlapping part. The equation works only when the two partitions actually extend each others, otherwise it would produce more than one value for some indices. Some indices may be left unassigned by this equation. The remaining indices must be arbitrarily chosen to make the renumbering vector a bijection. The implementation simply assigns the remaining values in order. In the example above, the renumbering vector would be $\{3,4,1,2\}$. The values of the first two indices are determined by the overlapping part, and the last two arbitrarily. The final merged partition is then $\{1,2,3,4,3,3,1\}$. After the algorithm has passed through all SNPs, the solution is the merged partition with the lowest score.

When some values of the renumbering vector are chosen arbitrarily, the merged partition's early reads and late reads are assigned essentially randomly. The merged partition above could as well have been $\{1,2,3,4,3,3,2\}$ and the merging would still be consistent. This leads to the possibility that some strains are swapped in the middle of the partition. The experiments section measures how often this happens in practice.

Calculating the partition cost function δ directly from its equation would take $O(c)$ time, where c is the coverage at the current SNP, and Δ would take $O(kc)$ time. However, two optimizations make it possible to do it faster. The first optimization is to iterate through the partition only once, and calculate δ for all strains simultaneously. This is done by keeping a two-dimensional array x with size $k * 4$ that keeps track of the cost for assigning a strain to a nucleotide. At each read in the partition, the cost for the strain that the read is assigned to is increased at the nucleotide the read has at that position. For example, if read 5 is assigned to strain 2, and read 5 has the nucleotide A at the current SNP, then the value at $x_{2,A}$ is increased by the weight of the cell when the iteration handles read 5. The final cost is then calculated with

$$\Delta(P, i) = \sum_{s \in [1..k]} \max(x_{s,A}; x_{s,T}; x_{s,C}; x_{s,G}) - (x_{s,A} + x_{s,T} + x_{s,C} + x_{s,G})$$

This improves the cost for calculating Δ to $O(c + k)$.

The second optimization is to use a *Gray code* to order the partitions. A Gray code is an ordering of vectors, in this case the partition vectors, where two consecutive vectors differ by exactly one element. Then, calculating the next x from the previous x can be done in constant time. At each SNP, the first x must be calculated as usual. Then, using a notation $x_{i,s,a}$ to mark x for the ith partition, $x_{i+1,s,a}$ is equal to $x_{i,s,a}$ except for the element that changed between the two partitions. For example, if the read 5 with nucleotide A was assigned to strain 3 at the previous partition, and to strain 4 at the current partition, then the next x is calculated with

$$\begin{cases} x_{i+1,3,A} = x_{i,3,A} - W(5, i) \\ x_{i+1,4,A} = x_{i,4,A} + W(5, i) \\ x_{i+1,s,a} = x_{i,s,a} & \text{elsewhere} \end{cases}$$

This removes the need to iterate through each partition and Δ can be calculated in amortized constant time. The Gray code optimization was originally described by Patterson et al. [PMP+14]. However, extending the optimization to multiple strains requires a special Gray code that only outputs the unpermutated forms of the partitions.

To use this optimization, the unpermutated forms of the partition must be ordered according to a Gray code. Due to lack of time, the Gray code optimization was not actually implemented, but the following describes how to do it. To form the Gray code ordering, consider a graph of the partitions, where nodes correspond to the partitions, and the edges connect partitions which differ by exactly one element. Then, a Gray code over the partitions is possible if the graph contains a Hamiltonian path, or a path that visits each node exactly once. For a partition over one read, this is trivially true. Then, a path for the graph with $c + 1$ reads can be constructed from a path in the graph with c reads. Each node in the graph with c reads is divided into the partitions where the first c elements are equal, and the final element varies. Figures 8 and 9 show an example of extending a graph of 3 reads to a graph of 4 reads. The node {1, 2, 1} is divided into the child nodes {1, 2, 1, 1}, {1, 2, 1, 2} and {1, 2, 1, 3}.

There are two key features of the graph that make a Hamiltonian path possible. First, a node's child nodes form a clique, so they can be visited in any order. Second, if two nodes were connected in the previous graph, their child cliques will have at least two connections between the cliques: the nodes that end with 1 are connected, and the nodes that end with 2 are connected. These nodes exist in all cliques. Therefore, to build a Hamiltonian path for

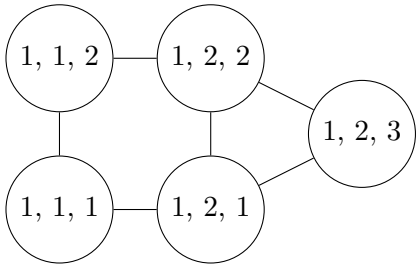

Figure 8: A graph of partitions for 3 reads

the $c+1$ graph from the c graph, mark every other node in the path with a 1, and every other with a 2. Then, divide each node in the path such that for nodes marked with a 1, the sub-path starts at the child clique's node that ends in 1, visits all nodes that end in 3 or higher, and ends at the child node that ends with 2. Correspondingly, for nodes marked with a 2, the sub-path starts at the child clique's node that ends in 2, visits all nodes ending with 3 or higher, and ends at the node ending with 1. Figures 10 and 11 show an example of extending a path in the graph of 3 reads to the graph of 4 reads. Algorithm 1 shows the pseudocode for the algorithm. The graph does not need to be explicitly created, and only the nodes in the path are processed. Since extending the graph by one read will at least double the number of nodes, the total number of nodes processed is at most twice the number of nodes in the final graph, so ordering the partitions with the Gray code can be done in linear time to the number of partitions.

In theory, the algorithm could be made with an asymptotic running time of $O(|S|C\frac{k^C}{k!})$, where C is maximum coverage, k is the number of strains and $|S|$ is the number of SNPs. However, some implementation details were chosen poorly and the running time of the actual algorithm is slower. The Gray code scheme described above is not used for calculating the partition cost function Δ. This means that calculating Δ is $O(k+C)$ instead of $O(1)$. When moving from one SNP to another, the new partitions that extend the former partitions are calculated by first generating all partitions for the new SNP, then sorting them by the overlapping area, and then going through both the sorted list of former partitions and the sorted list of new partitions at the same time. The partitions are sorted with a comparison sort, so calculating the partition extensions is an $O(n \log n)$ operation. The final running time of the implementation is therefore $O(|S|\frac{k^C}{k!} \log \frac{k^C}{k!} + |S|C\frac{k^C}{k!})$.

The feature that only unpermutated forms are considered complicates finding the partition extensions. Since the unpermutated forms of the partitions' overlapping areas are used for finding the extensions, the partitions cannot be processed in the same order both at the current SNP and at the second SNP. One way of mitigating this is to use a radix sort for the partitions

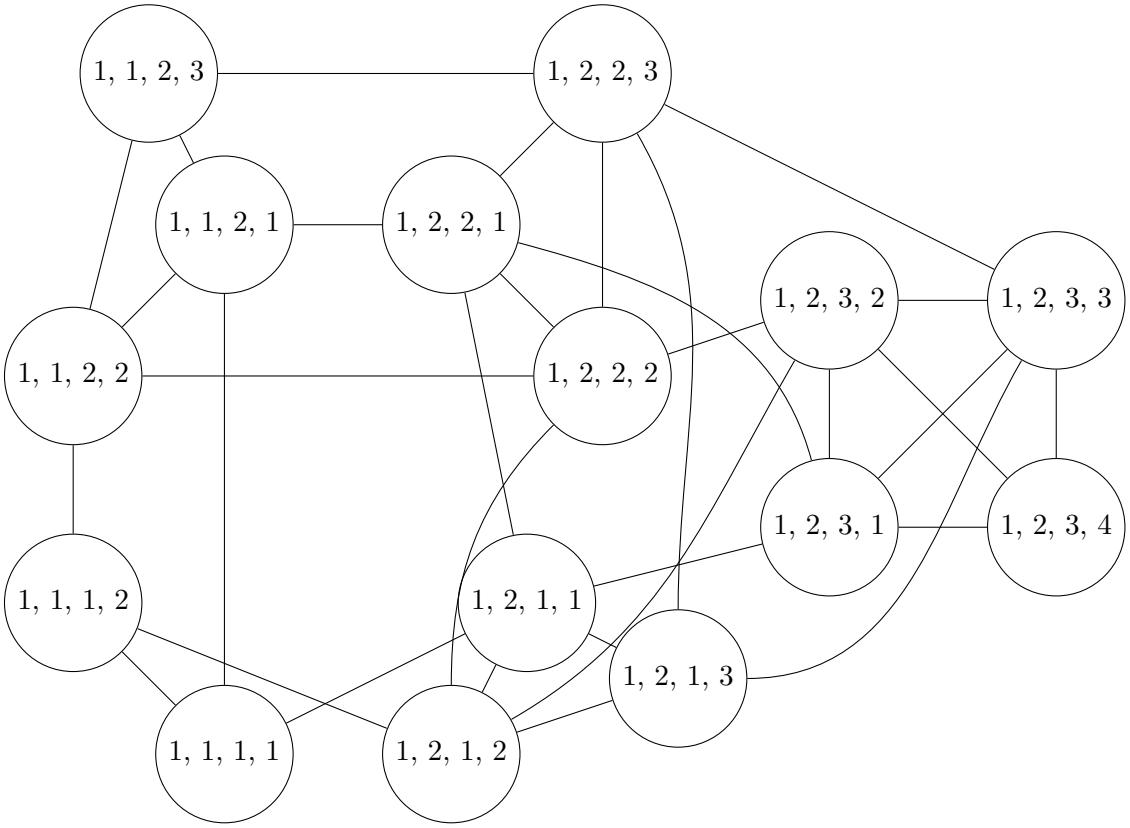

Figure 9: A graph of partitions for 4 reads

instead of a comparison sort, which brings the partition extension calculation down to $O(nC)$. The total running time would then be $O(|S|C\frac{k^C}{k!})$. The question of whether it is possible to enumerate the partition extensions in linear time without sorting remains open.

4 Experiments

The experiments were run on simulated data based on the *E. coli* genome. The simulated mutant genomes were created by taking the first ten thousand bases from the E. Coli genome and creating four mutant strains from it. The mutations were created with a uniform 1% probability of substitution per base. Only SNPs with substitutions were created. Reads were sampled from random locations with an average coverage of 20 for each strain, for a total average coverage of 80 before coverage reduction. All reads were created error-free.

Two different methods were used to build the haplotyping matrix from

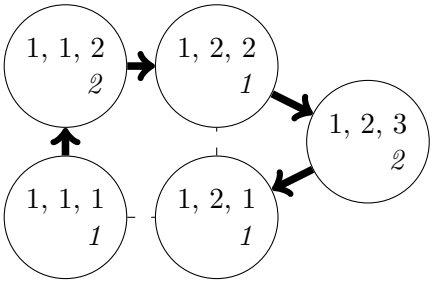

Figure 10: A hamiltonian path in the partition graph over 3 reads with nodes marked

the reads. In the first method, the *simulated* method, the SNPs were directly read from the mutated genomes and the haplotyping matrix was built with full knowledge of the genomes. The simulated method represents a very optimistic upper bound for the algorithm's accuracy. In the second method, the *DiscoSNP* method, SNPs were detected by DiscoSNP [URL+14] and processed exactly as described in the implementation section without any knowledge of the genomes. The DiscoSNP method represents a more realistic impression of the algorithm's accuracy.

Two experimental settings were used. First, the read length was varied from 100 bases to 2000 bases long reads, and the algorithm's accuracy was measured with each length. In the second setting, errors were introduced into the haplotyping matrix and the algorithm's accuracy measured for several read lengths. The errors were introduced directly into the haplotyping matrix immediately after creating it, before any coverage reduction. The reads passed to DiscoSNP were still error-free. This was done to make sure that the experiments only measure errors made by the implementation instead of the external tools.

4.1 Accuracy measures

Three measures were used for accuracy. First, to measure the accuracy of the coverage reduction step, *merge accuracy* was used. During coverage reduction, the program keeps track of which original reads are contained in which merged read. After coverage reduction, a read was merged correctly if all of its original reads were from the same strain. Merge accuracy is the fraction of correctly merged reads after coverage reduction.

The second accuracy measure is *partition accuracy*. Partition accuracy maps each strain post-phasing to an original strain, and then calculates the fraction of reads which were assigned to the correct strain. Each mapping of post-phased strains to original strains is tried and the best value is selected.

The third accuracy measure is *switch distance*. There are actually two

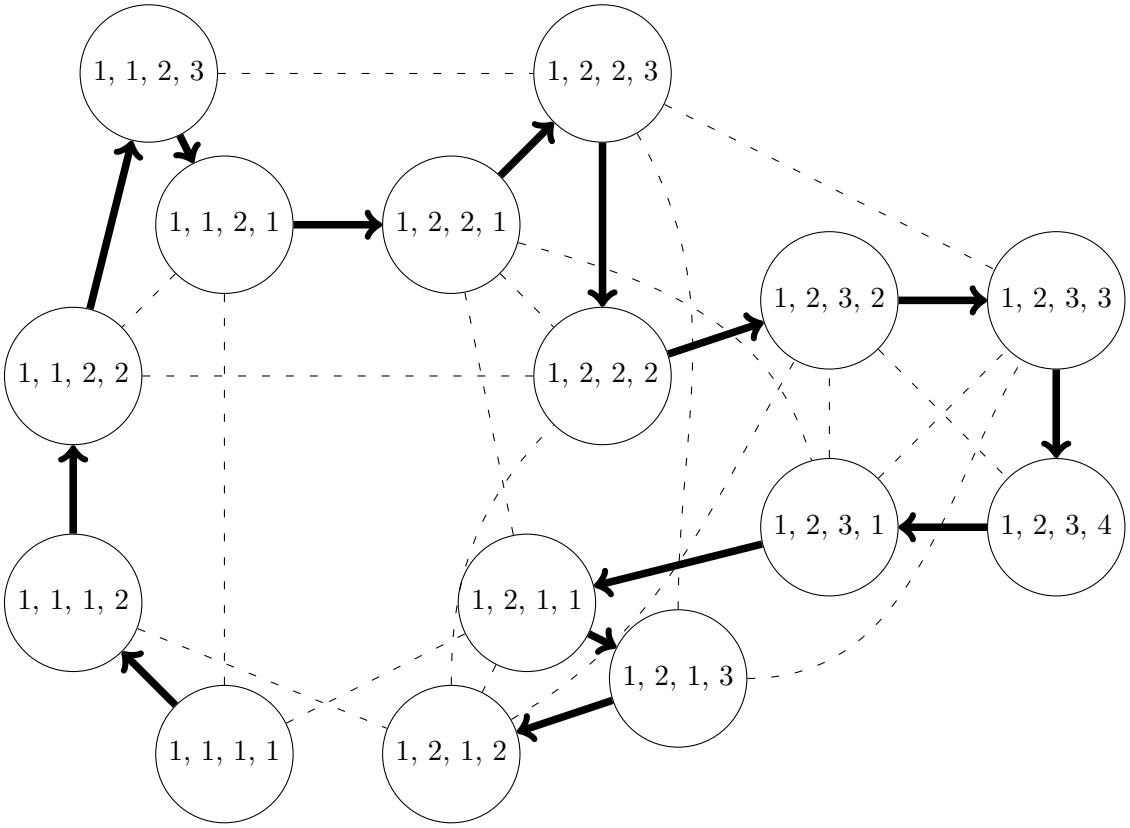

Figure 11: A hamiltonian path in the partition graph over 4 reads

similar but different measures used for switch distance. The switch distance measures are generalizations of the switch distance described by Lin et al. [LCZC02], which is defined for two strains. Informally, switch distance builds consensus genomes from its reads, and then aligns the consensus genomes to the actual genomes. Switch distance is then the number of times a consensus genome switches from one actual genome to another, allowing up to a certain number of alignment errors where a consensus genome's nucleotide differs from the actual genome's nucleotide. Figure 12 shows an example with three strains. The consensus sequences are colored red, green and blue, and the actual sequences are colored black. There are two switches, one at the start where two strain swap, and another at the end where all three strains swap. There is one alignment error in the middle of the bottom strain, where the actual sequence has an A and the aligned consensus sequence has a T.

To calculate the switch distance for multiple strains, a dynamic programming algorithm must be used. First, the nucleotide at each SNP position is calculated for both the consensus genomes and the actual genomes. The

Algorithm 1 Gray code enumeration

```
function GrayCode(maxPosition)
    mark ← {1, 1, ..1}                          ▷ |mark| = maxPosition
    path ← ∅                                    ▷ A list of partition vectors
    GCRec(1, maxPosition, {1})
    return path
procedure GCRec(position, maxPosition, currentPartition)
    if position = maxPosition then
        path ← path + currentPartition
        return
    if mark[position] = 1 then
        startNode ← 1
        endNode ← 2
        mark[position] ← 2
    else
        startNode ← 2
        endNode ← 1
        mark[position] ← 1
    GCRec(position + 1, maxPosition, currentPartition + startNode)
    for i ∈ [3.. min(max(currentPartition) + 1, k)] do
        GCRec(position + 1, maxPosition, currentPartition + i)
    GCRec(position + 1, maxPosition, currentPartition + endNode)
```

switch matrix $S_{i,p,s}$ holds the best score at SNP i for assignment p with s errors so far. At first SNP, the matrix is initialized with $S_{1,p,0} = 0$. At every SNP after that, the matrix is calculated by the recurrence

$$S_{i,p,s} = \min_{p'} S_{i-1,p',s-d(i,p')} + f(p, p')$$

The *assignment distance function* f measures the distance between two assignments. The *error function* d measures the number of alignment errors, or genomes where the nucleotide of the consensus genome is different from the nucleotide of the actual genome that the consensus genome is assigned to. The final switch distance is then calculated at last SNP by

$$switch\_distance = \min_{p,s} S_{n,p,s}$$

The matrix is bounded to allow s only up to some maximum number of errors. In the experiments, s was bounded to allow less than 10 errors.

As mentioned before, there are two similar measures used for switch distance. In the non-normalized *switch distance* mode, the assignment distance function counts the number of strains which were swapped between two assignments. For example, the left switch in figure 12 has two swaps,

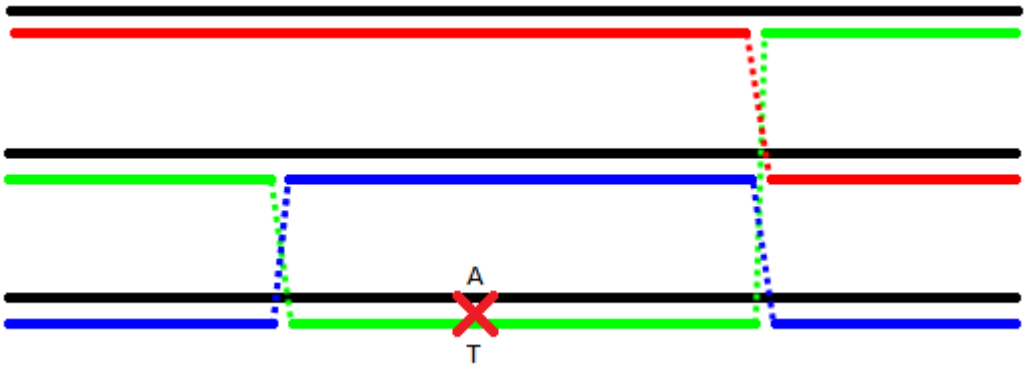

Figure 12: Switch distance with three strains

and the right switch has three. In the normalized mode, the function returns one if any switches happen and zero otherwise. The non-normalized mode is sensitive to the size of the error, as swapping all four genomes contributes a higher distance than swapping just one genome. The normalized mode measures the fraction of positions where genomes are switched, which is easier to interprete since it can be mapped to a number between zero and one. In the graphs, the normalized mode is further changed to *switch accuracy*, which is the inverse of the normalized switch distance, or the fraction of SNPs where genomes are not switched. This is done to preserve consistancy with the other normalized measurement methods where 1 is the best score and 0 is the worst.

4.2 Effect of varying read length

In the read length experiment, the implementation's accuracy was measured with varying read lengths. Read lengths were varied from 100 bases to 2000 bases. Figures 13, 14 and 15 show the partition accuracy, non-normalized switch distance and normalized switch accuracy, respectively.

The results show that accuracy depends greatly on read length. Even in the simulated method, reads shorter than about 300 bases have poor accuracy, and 100 bases long reads have an accuracy about as good as random guessing. On the other hand, long reads have a very high accuracy. The simulated method was completely accurate at 700 bases long, and the DiscoSNP method at 1600 bases, except for a strange dip at 2000 bases long reads. The experiment shows that the implementation cannot work for short reads even in the best case, but works well for long reads.

Another result is that the consensus of the reads is more accurate than the read partitioning. The partition accuracy graph shows that, for example, in the DiscoSNP method with 1000 bases long reads, less than 70% of reads

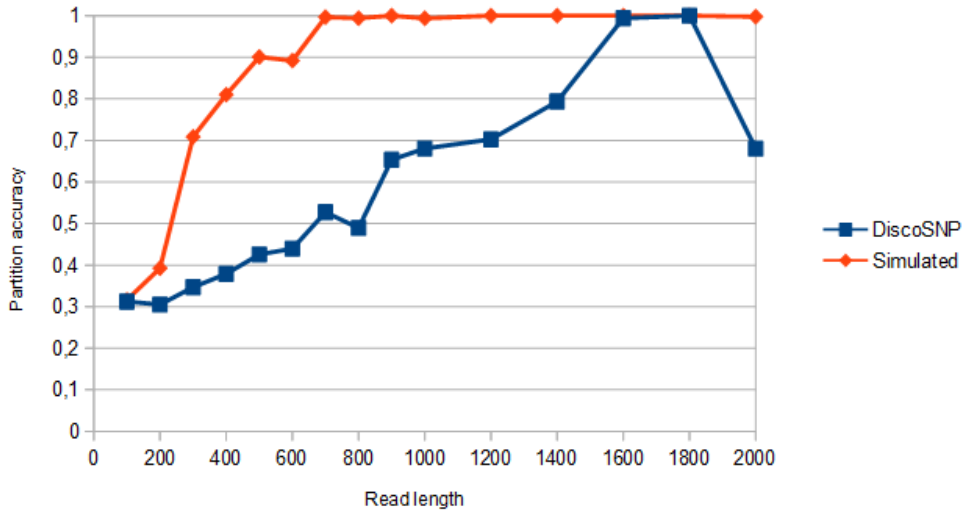

Figure 13: Partition accuracy as a function of read length

Read length	100	200	300	400	500	600	700	800	900	1000	1200	1400	1600	1800	2000
DiscoSNP	129	70	42	27	26	16	16	9	8	8	5	2	0	0	5
Simulated	102	27	6	4	2	2	0	0	0	0	0	0	0	0	0

Figure 14: Switch distance as a function of read length

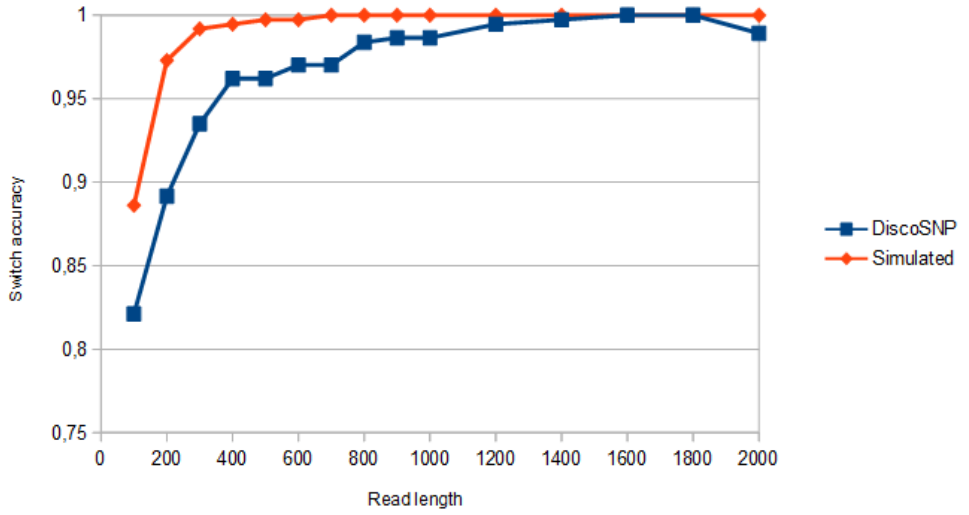

Figure 15: Switch accuracy as a function of read length

are assigned into correct strains. However, the switch distance shows that there were only 8 switches.

4.3 Effect of varying error rate

In the error rate experiment, the implementation's accuracy was measured with varying error rate. The error rate was varied from 0% to 15% chance of a substitution error per base. The errors were introduced into the haplotyping matrix immediately after building the matrix, before any coverage reduction. Read lengths were 1000, 1600 and 2000 bases long for the DiscoSNP method, and 500, 700 and 1000 bases long for the simulated method. Only uniformly distributed substitution errors were considered. All results are the average of 10 runs. Figures 16, 17, 18, 19, 20, 21 show DiscoSNP partition accuracy, simulated partition accuracy, DiscoSNP non-normalized switch distance, simulated non-normalized switch distance, DiscoSNP normalized switch accuracy and simulated normalized switch accuracy, respectively.

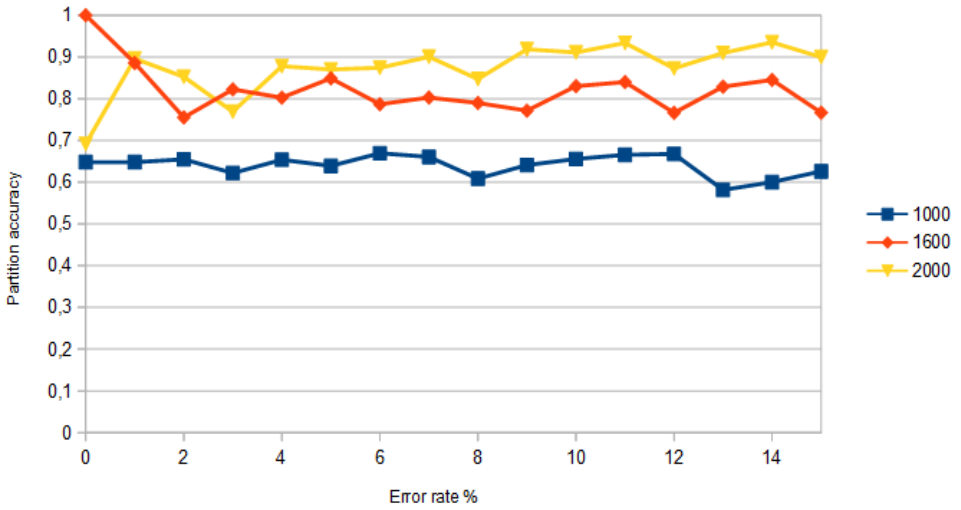

Figure 16: Partition accuracy as a function of error rate, DiscoSNP method

The results are different for the DiscoSNP method and the simulated method. In the simulated method, increasing the error rate lowers accuracy, however the effect seems to be small. Surprisingly, in the DiscoSNP method, varying the error rate has basically no effect on the implementation's accuracy. Even at the highest tested error rate, the effect from varying read length dominates the error rate's effect. This shows that uniformly distributed substitution errors are not an issue for the implementation even at high error rates, however the results should not be interpreted to mean anything about systematic read errors.

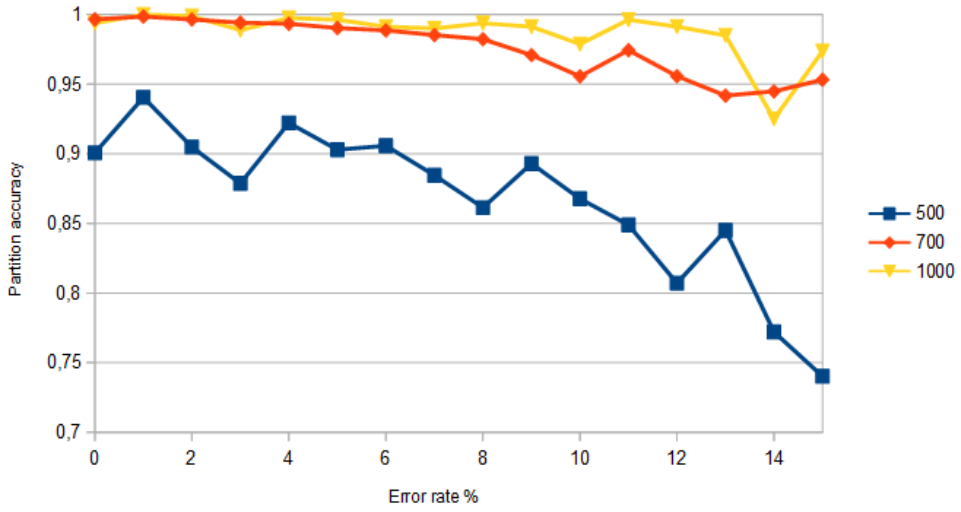

Figure 17: Partition accuracy as a function of error rate, simulated method

Error rate %	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16
1000	8,2	6	7,1	6	5,6	6	5,7	6,8	6,8	6,3	6	6,6	6,7	7,5	7,6	7,2
1600	0	1,4	3,2	1,9	2,7	2,4	2,4	2,2	2,6	2,3	1,8	1,8	2,6	1,6	1,7	3,5
2000	4,8	1,2	2,1	2,8	1,5	1,5	1,5	1	1,8	0,7	0,8	0,5	1,4	0,9	0,7	0,9

Figure 18: Switch distance as a function of error rate, DiscoSNP method

Error rate %	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16
500	2	1	1,8	2,2	1,4	1,8	1,3	2,1	2,3	1,8	2,2	2,4	3,6	2,6	3,5	4
700	0	0	0	0	0	0	0	0	0	0,3	0,4	0	0,2	0,2	0,4	0,2
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,2	0,1

Figure 19: Switch distance as a function of error rate, simulated method

Again, the consensus of the reads is more accurate than the read partitioning. In the DiscoSNP method with 1600 bases long reads, partition accuracy is about 80% while the switch distance is under 4.

4.4 Coverage reduction accuracy

The coverage reduction was not a separate experiment, instead the data was collected alongside the two other experiments. In all of the experiments, the number of merged reads after coverage reduction was about 5-10% of the original reads. The coverage reduction measures the accuracy of the

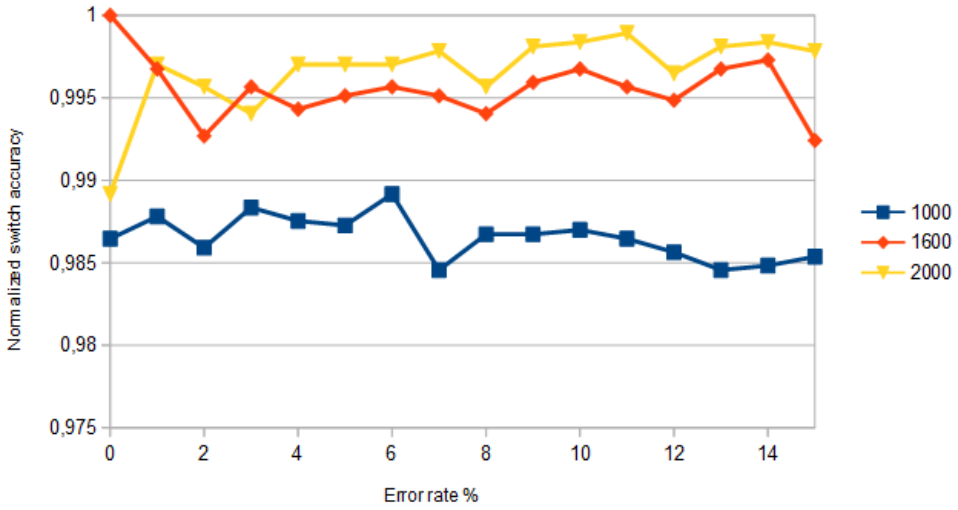

Figure 20: Switch accuracy as a function of error rate, DiscoSNP method

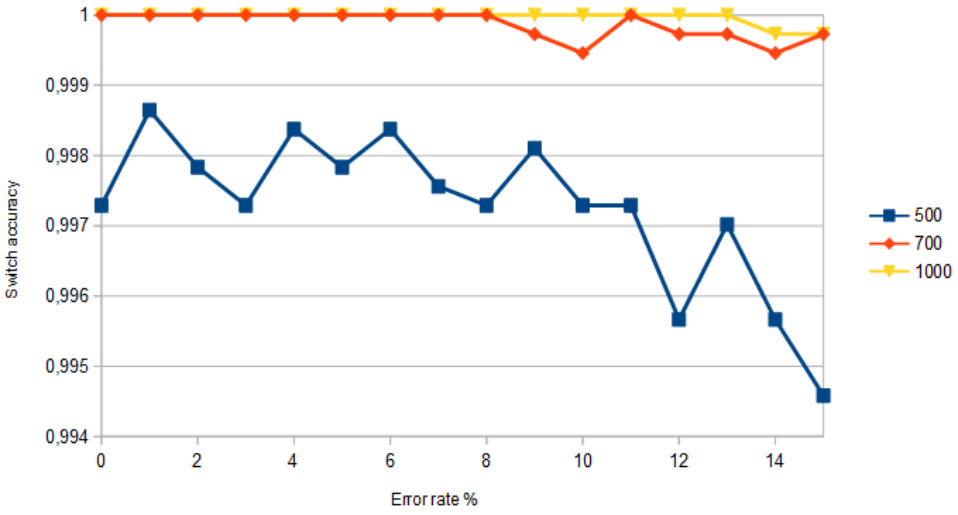

Figure 21: Switch accuracy as a function of error rate, simulated method

haplotyping matrix reduction process. Figures 22, 23 and 24 show accuracy in the read length experiment, accuracy for the DiscoSNP method in the error rate experiment and accuracy for the simulated method in the error rate experiment, respectively.

The results show that coverage reduction is more accurate with longer reads, and less accurate with erroneous reads. Merge accuracy drops with an increasing error rate, which makes the results of the error rate experiment

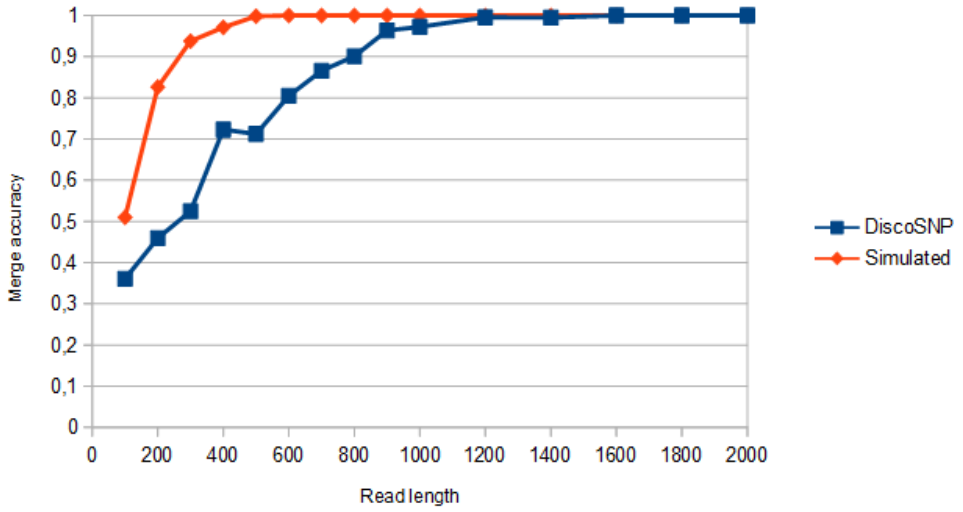

Figure 22: Merge accuracy as a function of read length

even more baffling. The merge accuracy for simulated reads with a length of 500 shows a strange upwards trend instead of the expected downwards trend, but this was due to random variation between runs.

5 Discussion

The algorithm works well for long reads and error rate has a surprisingly small effect. For short reads, the algorithm's accuracy is very poor and comparable to guessing randomly. The algorithm obviously cannot work with short read sequencing data, but methods for sequencing sufficiently long reads exist [SJ08] [EFG$^+$09] [FWL$^+$10] and the algorithm can handle high uniformly distributed error rates well.

The algorithm's performance with non-uniformly distributed errors was not tested. The results of the error rate experiment suggest that it might be able to handle high error rates, but non-uniform errors might behave very differently from uniformly distributed errors. One possible mechanism for this would be that uniformly distributed errors somehow cancel each others on average, but non-uniformly distributed errors might not.

The matrix banding step rearranges the SNPs and reads to lower the haplotyping matrix's coverage. In practice, the banding achieves good results in the sense that coverage is low. It was not tested whether this is because the banding's result is close to the original locations of the SNPs and reads, or whether it is an entirely different ordering that just happens to have low coverage. If the banding's result is close to the original, it might be possible to use matrix banding for estimating the locations of the reads and the SNPs

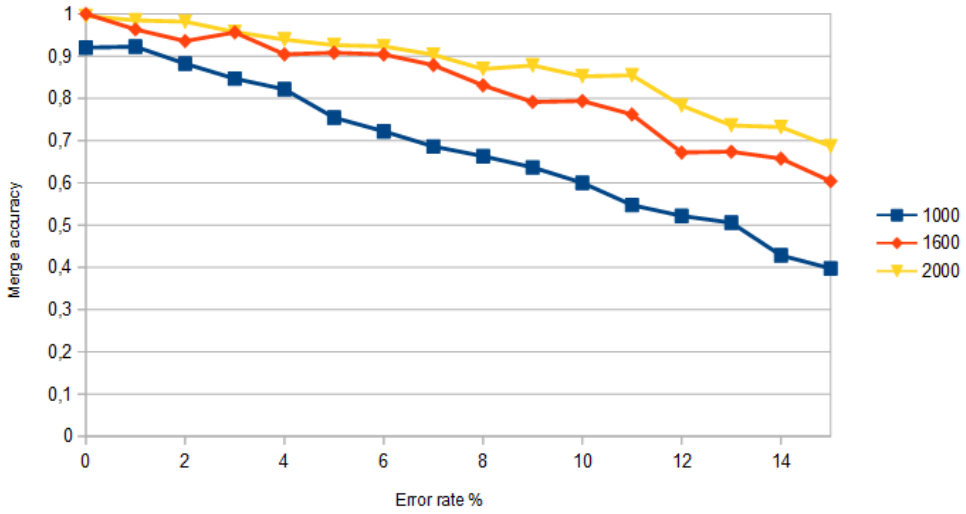

Figure 23: Merge accuracy as a function of error rate, DiscoSNP method

in the genome without a reference sequence.

The read merging process in coverage reduction discards information about lower weight variants. This makes the result dependant on the order in which the reads are merged, and can even produce the wrong consensus base for the merged read. Since the coverage reduction keeps track of which reads were merged, it could be possible to recalculate the consensus for the merged read afterwards. The experiments did not measure how often the merging process produced a wrong consensus.

The algorithm currently handles only SNPs with substitutions, but it could be extended to handle SNPs with indels. This would be accomplished by allowing the haplotyping matrix to have five alphabets for A, C, T, G and indel. The indel should not be confused with the non-covering marker. The partition error function δ might need to be adjusted to have different weight for the indel, depending on the sequencing method's ratio of indel errors to substitution errors.

The algorithm's performance on simulated data raises the question of whether it would work on real data. Some implementation details would need to be corrected to use the implementation on actual sequencing data. The very first part of the algorithm, mapping the reads to the SNPs, uses a custom aligner which assumes that the reads are error-free. Using a proper multiple read aligner program would enable it to work with erroneous reads, and most likely speed up that part as a side-effect. Alignment software for long reads exists [LD10] [LS12]. When initially building the haplotyping matrix, the implementation sets all weights to one. The weights should be based on the confidence of the read.

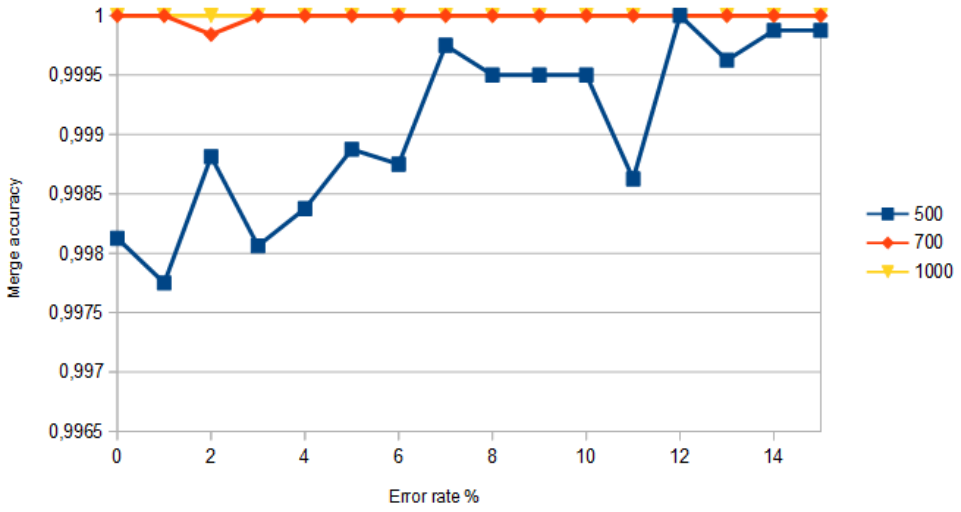

Figure 24: Merge accuracy as a function of error rate, simulated method

In conclusion, this thesis gives a proof of concept implementation of an exact haplotyping algorithm for multiple strains. The performance on simulated data is promising. The implementation probably would not work with real data as-is, but the corrections described above should make it possible.

References

[AI13] Derek Aguiar and Sorin Istrail. Haplotype assembly in polyploid genomes and identical by descent shared tracts. *Bioinformatics*, 29(13):i352–i360, 2013.

[ATM+11] Irina Astrovskaya, Bassam Tork, Serghei Mangul, Kelly Westbrooks, Ion Măndoiu, Peter Balfe, and Alex Zelikovsky. Inferring viral quasispecies spectra from 454 pyrosequencing reads. *BMC Bioinformatics*, 12, 2011.

[BAMR13] Shamsuzzoha Bayzid, Maksudul Alam, Abdullah Mueen, and Saidur Rahman. Hmec: A heuristic algorithm for individual haplotyping with minimum error correction. *ISRN Bioinformatics*, 2013, 2013.

[BYPB14] Emily Berger, Deniz Yorukoglu, Jian Peng, and Bonnie Berger. Haptree: A novel bayesian framework for single individual polyplotyping using ngs data. *PLOS Computational Biology*, 2014.

[CDW13] ZZ Chen, F Deng, and L Wang. Exact algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 29:1938–45, 2013.

[Cor62] Donovan Stewart Correll. *The potato and its wild relatives*. Texas Research Foundation, 1962.

[CvIKT05] Rudi Cilibrasi, Leo van Iersel, Steven Kelk, and John Tromp. On the complexity of several haplotyping problems. In Rita Casadio and Gene Myers, editors, *Algorithms in Bioinformatics*, volume 3692 of *Lecture Notes in Computer Science*, pages 128–139. Springer Berlin Heidelberg, 2005.

[DCW13] Fei Deng, Wenjuan Cui, and Lusheng Wang. A highly accurate heuristic algorithm for the haplotype assembly problem. *BMC Genomics*, 14, 2013.

[DV15] Shreepriya Das and Haris Vikalo. Sdhap: haplotype assembly for diploids and polyploids via semi-definite programming. *BMC Genomics*, 16(1):1–16, 2015.

[EFG+09] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, Arkadiusz Bibillo, Keith Bjornson, Bidhan Chaudhuri, Frederick Christians, Ronald Cicero, Sonya Clark, Ravindra Dalal, Alex deWinter, John Dixon, Mathieu Foquet, Alfred Gaertner, Paul Hardenbol, Cheryl Heiner, Kevin Hester, David Holden, Gregory Kearns, Xiangxu Kong, Ronald Kuse, Yves Lacroix, Steven Lin, Paul Lundquist, Congcong Ma, Patrick Marks, Mark Maxham, Devon Murphy, Insil Park, Thang Pham, Michael Phillips, Joy Roy, Robert Sebra, Gene Shen, Jon Sorenson, Austin Tomaney, Kevin Travers, Mark Trulson, John Vieceli, Jeffrey Wegener, Dawn Wu, Alicia Yang, Denis Zaccarin, Peter Zhao, Frank Zhong, Jonas Korlach, and Stephen Turner. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[FWL+10] Benjamin A Flusberg, Dale R Webster, Jessica H Lee, Kevin J Travers, Eric C Olivares, Tyson A Clark, Jonas Korlach, and Stephen W Turner. Direct detection of dna methylation during single-molecule, real-time sequencing. *Nature Methods*, 7:461–465, 2010.

[GHL04] Harvey J. Greenberg, William E. Hart, and Giuseppe Lancia. Opportunities for combinatorial optimization in computational biology. *INFORMS Journal on Computing*, 16(3):211–231, 2004.

[HCP+10] Dan He, Arthur Choi, Knot Pipatsrisawat, Adnan Darwiche, and Eleazar Eskin. Optimal algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 26(12):i183–i190, 2010.

[JHJ08] Vladimir Jojic, Tomer Hertz, and Nebojsa Jojic. Population sequencing using short reads: Hiv as a case study. In *Pacific Symposium on Biocomputing*, 2008.

[Jun11] Esa Junttila. *Patterns in permuted binary matrices*. PhD thesis, University of Helsinki, 2011.

[Kul14] Volodymyr Kuleshov. Probabilistic single-individual haplotyping. *Bioinformatics*, 30(17):i379–i385, 2014.

[LCZC02] Shin Lin, David J. Cutler, Michael E. Zwick, and Aravinda Chakravarti. Haplotype inference in random population samples. *American journal of human genetics*, 71:1129–37, 2002.

[LD10] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[LS12] Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 2012.

[LSLI02] Ross Lippert, Russell Schwartz, Giuseppe Lancia, and Sorin Istrail. Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem. *Briefings in Bioinformatics*, 3(1):23–31, 2002.

[NGB+08] Jost Neigenfind, Gabor Gyetvay, Rico Basekow, Svenja Diehl, Ute Achenbach, Christiane Gebhart, Joachim Selbig, and Birgit Kersten. Haplotype inference from unphased snp data in heterozygous polyploids based on sat. *BMC Genomics*, 9, 2008.

[PMAP13] Shruthi Prabhakara, Raunaq Malhotra, Raj Acharya, and Mary Poss. Mutant-bin: Unsupervised haplotype estimation of viral population diversity without reference genome. *Journal of Computational Biology*, 20:453–463, 2013.

[PMP+14] Murray Patterson, Tobias Marschall, Nadia Pisanti, Leo van Iersel, Leen Stougie, Gunnar W Klau, and Alexander Schönhuth. Whatshap: Weighted haplotype assembly for future-generation sequencing reads. *Journal of Computational Biology*, 22(00), 2014.

[SJ08] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature Biotechnology*, 26:1135–1145, 2008.

[SST+01] J. Claiborne Stephens, Julie A. Schneider, Debra A. Tanguay, Julie Choi, Tara Acharya, Scott E. Stanley, Ruhong Jiang, Chad J. Messer, Anne Chew, Jin-Hua Han, Jicheng Duan, Janet L. Carr, Min Seob Lee, Beena Koshy, A. Madan Kumar, Ge Zhang, William R. Newell, Andreas Windemuth, Chuanbo Xu, Theodore S. Kalbfleisch, Sandra L. Shaner, Kevin Arnold, Vincent Schulz, Connie M. Drysdale, Krishnan Nandabalan, Richard S. Judson, Gualberto Ruaño, and Gerald F. Vovis. Haplotype variation and linkage disequilibrium in 313 human genes. *Science*, 293(5529):489–493, 2001.

[SWBC08] Shu-Yi Su, Jonathan White, David J Balding, and Lachlan JM Coin. Inference of haplotypic phase and missing genotypes in polyploid organisms and variably copy number genomic regions. *BMC Bioinformatics*, 9, 2008.

[TBT+11] Ryan Tewhey, Vikas Bansal, Ali Torkamani, Eric J. Topol, and Nicholas J. Schork. The importance of phase information for human genetics. *Nature Reviews Genetics*, 12:215–223, 2011.

[URL+14] Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated snps. *Nucleic Acids Research*, 2014.

[ZBEB11] Osvaldo Zagordi, Arnab Bhattacharya, Nicholas Eriksson, and Niko Beerenwinkel. Shorah: estimating the genetic diversity of a mixed sample from next-generation sequencing data. *BMC Bioinformatics*, 12, 2011.