Identification of variant compositions in related strains without reference

Mikko Rautiainen, Leena Salmela, and Veli Mäkinen

Helsinki Institute for Information Technology
Department of Computer Science, University of Helsinki, Finland

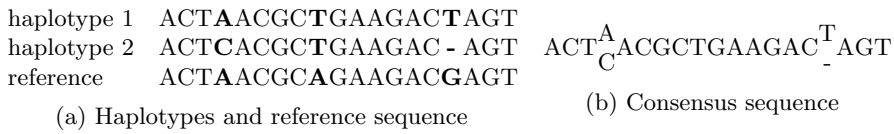
Abstract. Current DNA sequencing technologies do not read an entire chromosome from end to end but instead produce sets of short *reads*, i.e. fragments of the genome. *Haplotype assembly* is the problem of assigning each read to the correct chromosome in the set of chromosomes in a homologous group, with the aid of the reference sequence. In this paper, we extend an existing exact algorithm for haplotype assembly of diploid species (Patterson et al, 2014) to the *reference-free*, polyploid case. A reference-free method does not exploit a reference genomic sequence of a species and thus we cannot exploit a known linear order for the reads and resulting variant positions. Therefore we obtain an unordered *variant composition* as a result. This setting can be also applied to the study of relative abundances of related bacterial strains.

Keywords: variant composition, reference free sequence analysis

1 Introduction

The *genome* of an organism is a sequence or sequences of the nucleotides, A, T, C, and G. Eukaryotic genomes are 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 other. For example, humans' genomes are arranged into pairs of homologous chromosomes, one of which is inherited from the 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 [7], 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 other 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 nucleotide in the chromosomes, or indels (insertion-deletion), where a nucleotide 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 1a shows an example of SNPs. Two chromosomes



(a) Haplotypes and reference sequence

(b) Consensus sequence

Fig. 1. The haplotypes of a genome and the corresponding consensus sequence

from an organism are compared to a *reference genome*. The sequences have three SNPs. In the beginning there is a heterozygous SNP with a substitution, in the middle there is a homozygous SNP with substitutions, and in the end there is a heterozygous SNP with an indel and a substitution.

Current sequencing technologies produce sets of *reads*, also called *fragments*, which are short sequences of DNA from anywhere in the organism's genome. 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. Furthermore, polyploid genomes are more difficult to sequence than diploid sequences. Another similar case is a population of closely related bacteria or virii. 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 1b shows an example with two sequences having a common consensus with two SNPs. However, the consensus does not tell whether the A and the T 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 [19, 21], thus having just a list of SNPs misses *phase* information, which describes which variants occur in the same chromosomes or strains. Identification of *variant compositions* aims to correct this. A variant composition is an assignment of the variants to the chromosomes or strains.

The process of assigning the variants of a SNP to the chromosomes is called *phasing*. In this paper we generalize an existing exact phasing algorithm for diploid genomes [17] to the reference-free, polyploid case. The phasing algorithm's time and space complexities are both exponential in the maximum *coverage* of the input. The coverage of a position is the number of reads that cover the position. See Figure 2 for an example.

2 Previous work and our contribution in context

Identification of variant compositions is closely related to the *haplotype assembly problem* which seeks to solve the sequences of an organism's haplotype. A variant composition is a list of variants for each haplotype at the SNPs, which can be used to calculate the haplotype sequences given a reference genome. Haplotype assembly problem has multiple similar formulations [14]. These formulations are

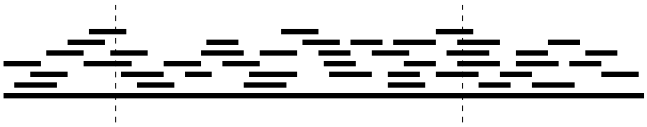


Fig. 2. The long, solid line is the genome, which is unknown, and the short solid lines represent the reads and their location in the genome. The position at the first dashed line is covered by three reads and thus has a coverage of three. Similarly, the second dashed line has a coverage of five.

based on building a *haplotyping matrix*, where the reads are rows and SNPs are columns, and partitioning 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, and then the problem is to modify the matrix the least to make such a partition possible. Lippert et al. [14] suggest three formulations, of which we follow the *minimum error correction* that aims to flip the least amount of cells in the matrix. Haplotype assembly is NP-hard both in general [14] and also in the case when there are no gaps between reads [6].

Identification of variant compositions is also related to the *quasispecies spectrum reconstruction problem* [2], which aims at reconstructing the sequences of the strains in a population of bacteria or virii and their frequencies. Variant compositions can be used to reconstruct the sequences, once reference sequences are given.

Haplotype assembly for diploid organisms, organisms only having two chromosomes, has many algorithms available, both exact [5, 10, 9, 17] and probabilistic [3, 12]. Solving the haplotype assembly for polyploid organisms, organisms having more than two chromosomes, is a more recent field of research. Both exact [16, 8] and probabilistic algorithms [1, 4, 20] are available.

Deng et al. [9] published in 2013 an exact algorithm for 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. Patterson et al. [17] extended the algorithm to the weighted minimum error correction formulation, and improved the time complexity to $O(|S|2^{C-1})$. In this paper we further extend 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 our implementation, we use DiscoSNP [22] to detect SNPs in the reads in a reference-free manner. DiscoSNP first builds a de Bruijn graph of the reads and then uses it to find the SNPs.

3 Extending haplotyping to multiple strains without reference

Our method has three stages. As preprocessing, a *haplotyping matrix* is built from the reads. The second stage manipulates the haplotyping matrix to lower

(a) A binary matrix before banding (b) A binary matrix after banding

Fig. 3. Matrix banding

coverage. The third stage, the *phasing* stage, assigns the reads into strains. Due to the lack of space, we describe the coverage reduction and phasing stages that form the algorithmic core of our contribution. More details are available in [18].

3.1 Coverage reduction

After the prerocessing stages, we assume to have a haplotyping matrix F that 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 "−" indicating 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. Additionally we have a weight matrix W where the entry $W(i, x)$ describes the certainty of the haplotyping matrix entry $F(i, x)$.

Because the phasing algorithm is exponential in coverage, reducing the coverage is necessary. Coverage is either *essential*, when a read supports some variant for a SNP, or *accidental* when a read does not directly support a SNP, but supports SNPs both before and after it.

Since a reference genome is not used, the reads cannot be ordered by aligning them to a reference genome. Instead we use matrix banding [11] to reduce accidental coverage. The haplotyping matrix is treated as a binary matrix, where cells with 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. Figure 3 shows 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.

We use methods presented in [11] for matrix banding. First we find an approximate *consecutive-ones property* (C1P) on the haplotyping matrix. The C1P is a permutation of the columns such that in any row, the ones are in consecutive positions. Then *barycentric sorting* is used. For each row, a centerpoint is calculated as the average position of the ones and the rows are 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*.

After banding, methods similar to the ones in [15] are used to still lower the essential coverage.

3.2 Phasing

The algorithm by Patterson et al. [17] works with two strains. Here we extend the algorithm for an arbitrary number of strains. 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.

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 $u(P)$. In the unpermutated form 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* which is a bijection on strain numbers. A *renumbering vector* describes how the strains are re-labeled. Given a partition P and a renumbering vector R, the renumbered partition P' is given by $P'_i = R_{P_i}$. The renumbering vector for the example above is $\{2, 3, 1, 4\}$. 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, and then 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 active reads for SNP i are denoted as $\alpha(i)$.

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 *conflict-free* if all reads in a strain assign the same variant at each SNP. For example, the haplotyping matrix in Figure 4 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 in conflict since it is different from the consensus variant of strain 2 for SNP 2.

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

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

Fig. 4. A haplotyping matrix

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, we 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, we 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 is the minimum cost to make the partition conflict-free. These functions are generalizations of the cost functions described in [17], 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 cost of a partition at the current SNP is

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

which is the sum of the cost for that partition at the current SNP and the cost of the best-scoring partition of the previous SNP that is extended by the current partition. To get the final result, the table C can be backtraced and the partitions at each SNP must be merged.

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 and it works only when the two partitions actually extend each other, 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. In the above example another consistent merging is $\{1, 2, 3, 4, 3, 3, 2\}$. Therefore some strains may be 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 of assigning a strain to a nucleotide. At each read in the partition, the cost of 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

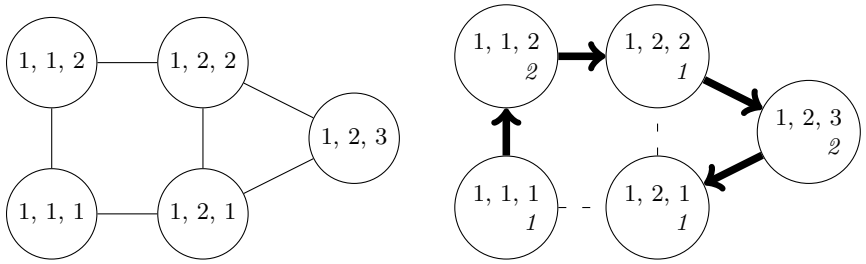
$$\Delta(P, i) = \sum_{s \in [1..k]} \max(x_{s,A}; x_{s,T}; x_{s,C}; x_{s,G}) - (x_{s,A} + x_{s,T} + x_{s,C} + x_{s,G})$$

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

The second optimization uses 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

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

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



(a) A graph of partitions for 3 reads (b) A hamiltonian path in the partition graph over 3 reads with nodes marked

Fig. 5. Example of partitions for three reads

by Patterson et al. [17]. However, extending it to multiple strains requires a special Gray code that orders only the unpermutated forms of the partitions.

Consider a graph of the partitions, where nodes are partitions, and edges connect partitions which differ by exactly one element. A Gray code over the partitions is possible if the graph has a Hamiltonian path, or a path visiting 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 5a and 6 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 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 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, for each node that is 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 5b and 7 show an example of extending a path in the graph of 3 reads to the graph of 4 reads. 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 in the number of partitions.

The algorithm has 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.

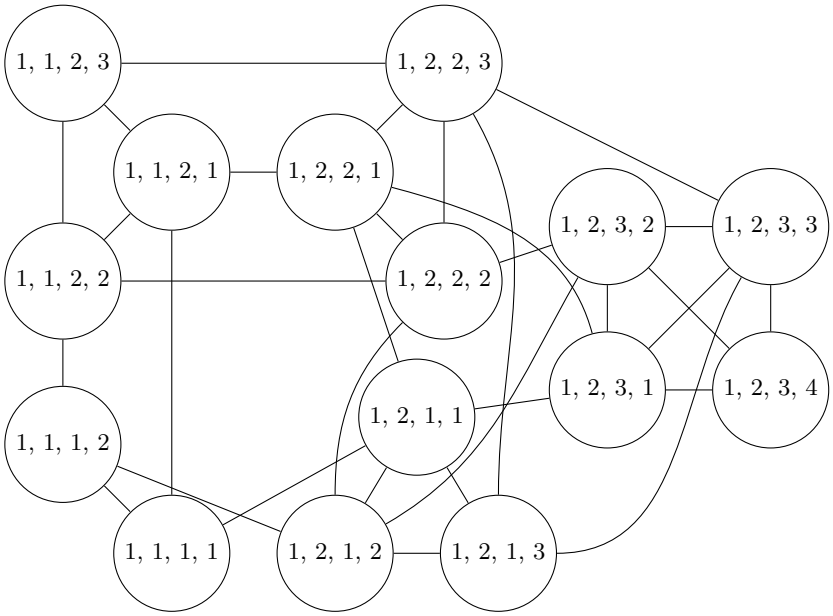


Fig. 6. A graph of partitions for 4 reads

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 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 [22] and processed to form the haplotyping matrix. 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, 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.

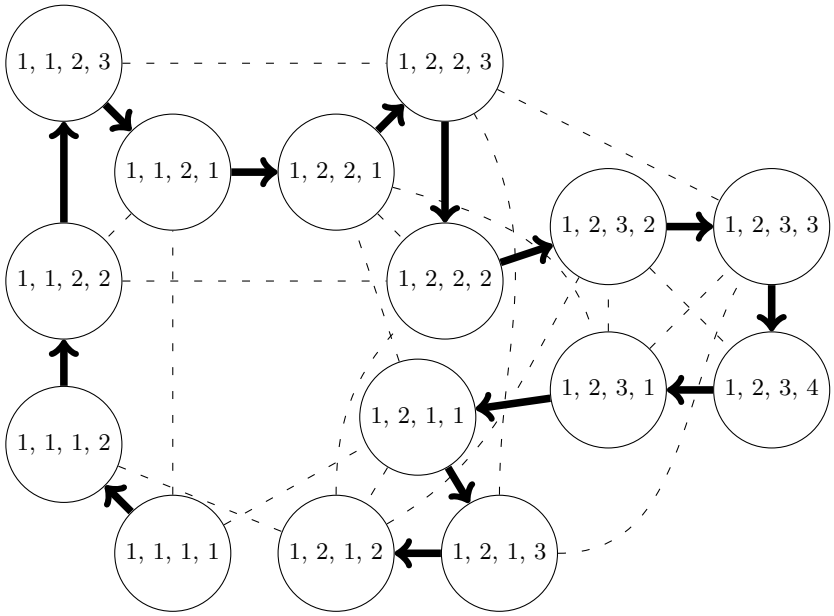


Fig. 7. A hamiltonian path in the partition graph over 4 reads

This was done to make sure that the experiments only measure errors made by the implementation instead of the external tools.

We used *switch distance* to measure the accuracy of predictions. Originally switch distance is defined for two strains [13]. 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.

To calculate the switch distance for multiple strains, we developed a simple dynamic programming algorithm, whose details can be found in [18]. Informally, this extended measure calculates the fraction of positions where genomes are switched. In the graphs, we report *switch accuracy*, which is the inverse of the switch distance, or the fraction of SNPs where genomes are not switched.

4.1 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. Figure 8 shows the switch accuracy.

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

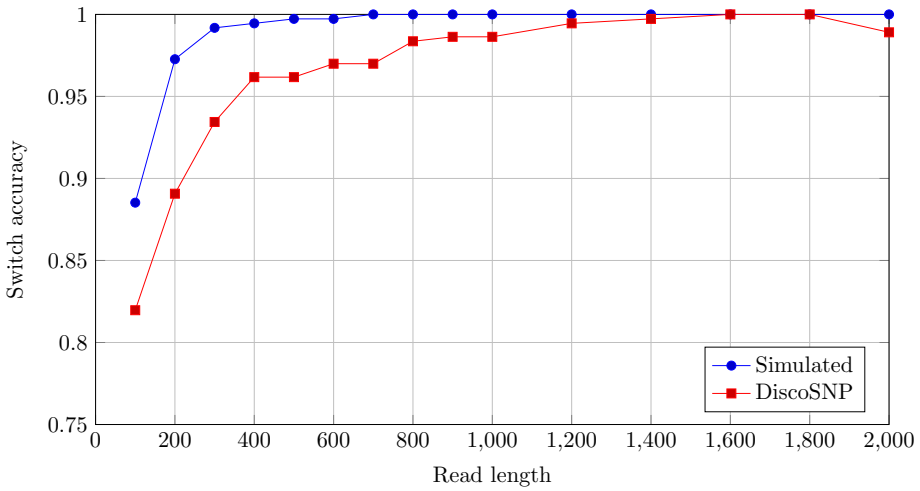


Fig. 8. Switch accuracy as a function of read length

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.

4.2 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. The switch accuracy stayed over 0.98 accuracy level in all these settings, and the increase in read length improved the accuracy level to 1 still at error level 8% in simulated data.

5 Conclusion

In this paper we gave a proof of concept implementation of an exact haplotyping algorithm for multiple strains. The performance on simulated data is promising.

Acknowledgements

This work was supported in part by the Academy of Finland (grants 267591 to L.S. and 284598 (CoECGR)).

References

1. D. Aguiar and S. Istrail. Haplotype assembly in polyploid genomes and identical by descent shared tracts. *Bioinformatics*, 29(13):i352–i360, 2013.
2. I. Astrovskaya et al. Inferring viral quasispecies spectra from 454 pyrosequencing reads. *BMC Bioinformatics*, 12(Suppl 6):S1, 2011.
3. S. Bayzid et al. Hmec: A heuristic algorithm for individual haplotyping with minimum error correction. *ISRN Bioinformatics*, 2013:10, 2013.
4. E. Berger et al. Haptree: A novel bayesian framework for single individual polyplotyping using ngs data. *PLOS Computational Biology*, 10(3):e1003502, 2014.
5. Z. Chen, F. Deng, and L. Wang. Exact algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 29(16):1938–1945, 2013.
6. R. Cilibrasi et al. On the complexity of several haplotyping problems. In R. Casadio and G. Myers, editors, *Algorithms in Bioinformatics*, volume 3692 of *Lecture Notes in Computer Science*, pages 128–139. Springer Berlin Heidelberg, 2005.
7. D. S. Correll. *The potato and its wild relatives.* Texas Research Foundation, 1962.
8. S. Das and H. Vikalo. Sdhap: haplotype assembly for diploids and polyploids via semi-definite programming. *BMC Genomics*, 16:260, 2015.
9. F. Deng, W. Cui, and L. Wang. A highly accurate heuristic algorithm for the haplotype assembly problem. *BMC Genomics*, 14(Suppl 2):S2, 2013.
10. D. He et al. Optimal algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 26(12):i183–i190, 2010.
11. E. Junttila. *Patterns in permuted binary matrices.* PhD thesis, University of Helsinki, 2011.
12. V. Kuleshov. Probabilistic single-individual haplotyping. *Bioinformatics*, 30(17):i379–i385, 2014.
13. S. Lin et al. Haplotype inference in random population samples. *American journal of human genetics*, 71(5):1129–1137, 2002.
14. R. Lippert et al. Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem. *Briefings in Bioinformatics*, 3(1):23–31, 2002.
15. V. Mäkinen et al. Interval scheduling maximizing minimum coverage. *CoRR*, abs/1508.07820, 2015.
16. J. Neigenfind et al. Haplotype inference from unphased snp data in heterozygous polyploids based on sat. *BMC Genomics*, 9:356, 2008.
17. M. Patterson et al. Whatshap: Weighted haplotype assembly for future-generation sequencing reads. *Journal of Computational Biology*, 22(6):498–509, 2015.
18. M. Rautiainen. Identification of variant compositions in related strains without reference. Master's thesis, University of Helsinki, 2016.
19. J. C. Stephens et al. Haplotype variation and linkage disequilibrium in 313 human genes. *Science*, 293(5529):489–493, 2001.
20. S.-Y. Su et al. Inference of haplotypic phase and missing genotypes in polyploid organisms and variably copy number genomic regions. *BMC Bioinformatics*, 9:513, 2008.
21. R. Tewhey et al. The importance of phase information for human genetics. *Nature Reviews Genetics*, 12:215–223, 2011.
22. R. Uricaru et al. Reference-free detection of isolated snps. *Nucleic Acids Research*, 43(2):e11, 2014.