strainMiner: Combining Integer Programming and Data Mining Techniques for Strain-level Metagenome Assembly

Roland Faure[1,2*†], Tam Khac Minh Truong[1†], Victor Epain[3], Riccardo Vicedomini[1], Rumen Andonov[1*]

[1]GenScale, Univ. Rennes, Inria RBA, CNRS UMR 6074, Rennes, France.
[2]Service Evolution Biologique et Ecologie, Université libre de Bruxelles (ULB), Brussels, Belgium.
[3]Unaffiliated, independent researcher, Lorient, France.

*Corresponding author(s). E-mail(s): roland.faure@irisa.fr; rumen.andonov@irisa.fr;
†These authors contributed equally to this work.

Abstract

Metagenomic assembly is crucial for understanding microbial communities, but standard tools often struggle to differentiate bacterial strains of the same species, especially with low-accuracy reads from technologies like PacBio CLR and Oxford Nanopore. Current de novo assembly methods typically reconstruct bacterial genomes at the species level but fall short in distinguishing individual strain genomes. Our study presents a novel approach by reformulating the haplotyping problem as a matrix partitioning problem. We address this using Integer Linear Programming (ILP) combined with data mining techniques to improve computational efficiency. We introduce strainMiner, a strain-separation module integrated into an established pipeline to produce strain-separated assemblies. On real and simulated datasets with error rates ranging from 2.5% to 12%, strainMiner compares favorably to state-of-the-art methods in terms of assembly quality and strain reconstruction while significantly reducing computational requirements.

Keywords: Metagenomics, Strain-level assembly, Haplotype phasing, Integer Linear Programming, Hierarchical Cluster Analysis

1 Introduction

Metagenomics is a fairly new research field that consists of the analysis of sequencing data characterizing a mixture of microorganisms within an environment of interest [1]. One of the steps for accomplishing this task is through the precise identification of the organisms that are present in such an environment. This problem often requires reconstructing the genomes of the sequenced species, a problem called *metagenome assembly*. Reconstructing and identifying bacterial genomes within a metagenome from sequencing data is an extremely challenging task due to the need of distinguishing and assembling DNA fragments of distinct microorganisms [2]. Furthermore, genomes may also exhibit widely distinct levels of abundance and relatedness, making it difficult to discern sequence variability from errors [3]. For example, conspecific strains (*i.e.*, strains of the same species) could share sequence identity above 99% and, in practice, are often assembled into species-level consensus sequences which hide strain variability [4]. Being able to precisely identify distinct strains is nevertheless important for studying a microbial environment at a functional level, due to the high phenotypic variability exhibited by conspecific strains [5]. A classical example is *Escherichia coli* which could be found as a probiotic [6] or pathogenic [7] strain.

The challenge posed by the "strain separation" problem, as outlined in [4], arises from two primary factors: (i) the unknown number of strains and (ii) the variable abundance within a sample. Moreover, the precise characterization of what constitutes a "strain" is also not always clear. In this study, we will define a strain as a bacterial haplotype, i.e. a contiguous sequence of nucleotides observed jointly and in sufficient abundance by sequencing reads, in accordance with previous works [4]. Furthermore, we will use the terms strain and haplotype interchangeably.

In the last decade the strain separation problem has been extensively studied, either without (de novo) or with the availability of a reference sequence. Previous works attempted to tackle the de novo problem exploiting data from different sequencing technologies such as short reads [8–11], long reads [4, 12–14], or a combination of the two [15].

The increased accessibility of long-read sequencing (Oxford Nanopore and PacBio) for metagenomic data allows nowadays to accurately reconstruct complete genomes of bacterial species even from complex environments [16], especially using the low-error-rate PacBio HiFi technology [17–19]. At the same time, long reads are able to span far-apart strain-specific variants, offering the possibility to identify and reconstruct bacterial genomes even at the strain level.

Several methods have been recently proposed for the de novo strain-level assembly with long-read metagenomic data, namely Strainberry [4], stRainy [12], Floria [13], and HairSplitter [14]. These approaches take as input a "reference" species-level assembly (*e.g.*, built with a standard metagenome assembly tool) along with a set of long reads. A read alignment against the input assembly is then used to identify single-nucleotide polymorphisms (SNPs) which allow to partition reads likely belonging to the same haplotype. Strain-resolved and unphased sequences are finally represented within a graph in order to output more contiguous strain-resolved sequences.

Strainberry [4] was the first long-read-based tool proposed for the reconstruction of individual strains at the scale of a full metagenome. It exploits HapCUT2 [20] (a

diploid phasing tool based on likelihood optimization through graph-cuts) which is applied iteratively until no more strains need to be separated. While Strainberry does not require long reads from a specific technology, it is mainly limited to low-complexity metagenomes, i.e. containing no more than five conspecific strains.

stRainy [12] constructs a "connection" graph that encodes overlapping reads, sharing and agreeing on SNPs. Then, it recursively clusters reads using a community detection algorithm [21] with increased sensitivity. As opposed to the other approaches, stRainy has been mainly evaluated on long reads with fairly low error rates (i.e., PacBio HiFi, Nanopore R10, simulated reads with error rate up to 3%).

Floria [13] is based on the Minimum Error Correction (MEC) model, typically applied to (genomic) polyploid haplotype phasing. However, it uses an heuristic method on overlapping windows to locally identify strain counts and read partitions. A directed acyclic graph is then constructed from such partitions in order to solve a flow problem whose solution is then used to extract vertex-disjoint paths representing haplotypes.

HairSplitter [14] introduces a novel statistical approach to distinguish sequencing errors from SNPs, making it suitable for all long-read technologies. In order to cluster reads, HairSplitter runs over non-overlapping windows and implements a k-nearest-neighbour algorithm to correct reads at SNP loci. It finally clusters the reads using the Chinese Whispers algorithm [22] in order to identify (and assemble) haplotypes.

The local clustering of reads in strain-specific partitions is at the core of methods for the strain separation problem. In this article, we introduce and study an original mathematical formulation for this specific task. More precisely, we model the problem as a *tiling problem* on a binary matrix defined over a set of SNP positions. The rationale is to identify high (or low) density sub-matrices representing SNPs in which reads share the same nucleotides (within a given tolerance for errors). We formalize this problem as an Integer Linear Programming (ILP) problem and we integrate it within HairSplitter [14], which implements a complete strain-level metagenome assembly pipeline. We named the resulting method strainMiner. This article completes and extends a previously published conference paper [23].

We evaluated strainMiner on mock communities (based on real and simulated data) in which it either improved or compared favorably with respect to competing tools to recover strain-specific sequences from long read sequencing data, while being either an order of magnitude faster or more memory-efficient.

2 Methods

2.1 Pipeline overview

The strainMiner pipeline takes in input a reference sequence (a draft assembly or a reference genome) and a set of long reads. It then mirrors the four main stages of HairSplitter [14] (see Figure 1). These stages include: (i) aligning the reads to the draft assembly or reference genome, (ii) identifying putative SNPs and partitioning the reads, (iii) producing haplotype assemblies from the read partitions, and (iv) enhancing assembly contiguity through scaffolding. Our contribution lies in an original method to tackle the second step of the pipeline, that is separating aligned reads by

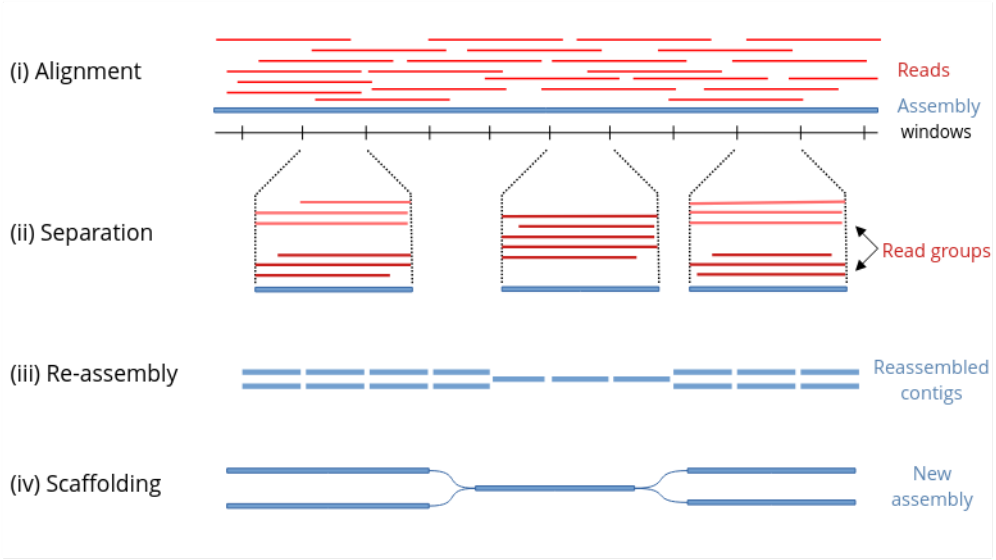


Fig. 1 The strainMiner pipeline [23]: (i) Reads are aligned on the reference or draft assembly, (ii) on each window of the assembly, reads are separated by haplotype of origin - only three windows are shown here, (iii) all groups of reads are locally reassembled and (iv) the locally reassembled contigs are scaffolded to produce longer contigs.

haplotype of origin. The other steps of the pipeline are the same ones implemented in HairSplitter [14] (version 1.6.10).

The problem we are tackling can be defined as follows: given a set of reads aligned to a reference sequence, the goal is to separate the reads into groups based on their haplotype of origin. Ideally, all reads originating from the same strain would be grouped together. However, this level of separation across the entire genome is not always achievable. It is impossible to phase two consecutive variants if they are too far apart to be covered by at least one read. Therefore, our objective is to partition reads locally. Specifically, we consider the input reference in non-overlapping windows of length w, where w should be smaller than the average read length (we set w to 5000 by default).

2.2 Statistical signal

The intuition behind strainMiner is similar to the one behind HairSplitter [14], originally introduced in [24]. The idea is to consider multiple loci simultaneously in order to group reads by their haplotypes of origin. Considering only one locus is not sufficient, as alignment artifacts and error rates can introduce errors at a single locus that cannot be distinguished from alternative alleles, especially when the alternative alleles are rare. On the contrary, by exploiting the correlation between several columns, it is possible to differentiate errors from true polymorphisms.

For instance, consider a hypothetical scenario involving a mixture of two strains, where strain A constitutes 99% of the mix and strain B a mere 1%. Consider also a collection of a thousand reads spanning two polymorphic sites, denoted as a and b.

In an ideal, error-free pileup, conducting a chi-square test for independence with one degree of freedom between the two loci yields a p-value smaller than 10^{-215}.

The presence of errors can greatly decrease the statistical power of detecting correlations between loci. In our simulations, we incorporated random substitution errors with a probability of $p = 0.1$ for all bases across 10,000 simulations. Despite this, the p-value for the correlation between loci a and b remained low, with an average of 10^{-16} and a maximum of 10^{-6} in the worst-case scenario. However, it is crucial to note that thousands of non-polymorphic positions may potentially exhibit correlations with locus a in a single pileup. Furthermore, alignment artifacts can introduce more complex errors with locally higher error rates, which can further decrease the statistical power of detecting correlations.

Including more loci unequivocally eliminates the risk of spurious correlations stemming from artifacts. In this same example, we introduced a third locus, denoted as c, while maintaining the 0.1 error rate. We applied the one-degree of freedom chi-square test to assess the relationship between the three positions. This time, the probability of encountering three non-polymorphic positions with correlations as strong as those observed between a, b, and c by chance was found to be below 10^{-200} in all 10,000 simulations. While this example simplifies the complexities of pileup errors, it emphasizes two fundamental aspects of the method: a) the joint observation of multiple loci significantly enhances the statistical power to distinguish between errors and polymorphism, and b) even low-abundance strains can be reliably identified.

2.3 Reference windows as binary matrices

In each window, strainMiner considers only reads that span at least 60% of the window's length. Subsequently, strainMiner transforms the read alignment pileup into a binary matrix, where each row corresponds to a read, each column corresponds to a position, and cell (i, j) contains the number one if the base at position j of read i matches the dominant base at that position, the number zero if it matches the second most frequent base, and remains empty otherwise. Empty cells can occur if a read does not cover a position or if the base in a read at a given position is not among the two most common bases at that position.

Next, columns are filtered to retain only those where the most common base constitutes at most a proportion p of the aligned reads. By default, strainMiner sets p to 0.95, striking a balance between computational efficiency and precision. However, users aiming to recover low-abundance strains can set p to a higher value.

To populate the empty cells, strainMiner implements the well-known K-nearest-neighbor imputation strategy [25], used for example in [26]. It identifies for each read its "nearest neighbors" which, in this context, are the reads with the smallest Hamming distance. Then, for each empty cell in a row, strainMiner uses a majority vote from the five closest neighbors that have non-empty cells at that position to decide whether the missing value is a 0 or a 1.

The result is a binary matrix $A = (a_{ij})$ of size $|R| \times |L|$, where the set of rows R corresponds to the reads, the set of columns L corresponds to the retained polymorphic loci/positions within the window, and $a_{ij} \in \{0, 1\}$ for all $1 \leq i \leq |R|, 1 \leq j \leq |L|$.

strainMiner aims to identify groups of highly similar columns in this matrix, which statistically can only represent true SNPs if the group is sufficiently large. Reads will then be separated into groups based on their alleles at these positions.

2.4 Definitions and problem formulation

In this section, we translate the statistical signal observed above in a formal problem on the binary matrix.

Quasi-bicluster of ones and zeros

To begin, we first introduce some preliminary definitions. The density $dens(A)$ of a binary matrix A is defined as the total number of ones divided by the total number of elements or, more formally,

$$dens(A) = \frac{\sum_{i \in R} \sum_{j \in L} a_{ij}}{|R| \times |L|}.$$

Furthermore, given a threshold $\gamma \in (0.5, 1]$, a *γ-bicluster of ones* (or *quasi-bicluster of ones*) is any sub-matrix M of A such that $dens(M) \geq \gamma$ holds. When γ is chosen to be close to 1, the corresponding matrix is called *dense*, i.e. a matrix mostly containing ones and tolerating only a small proportion of zeros to account for various sequencing errors. Conversely, a *$(1-\gamma)$-bicluster of zeros* (or *quasi-bicluster of zeros*) is a sparse binary matrix, i.e. characterized by low density (close to 0).

The quasi-bicluster dimensions are constrained by a minimum number of columns (width) and by a minimum number of rows (height), in order to ensure the significance of the statistical signal captured by the quasi-bicluster.

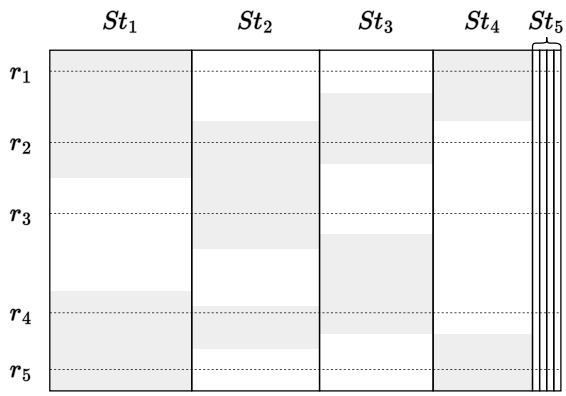
Tiling formulation

We approach the strain separation problem as a specific tiling problem: given the binary matrix A constructed as described in the previous section, we aim to permute and partition its columns into vertical bands, also referred to as *strips* (see Figure 2a). A strip is a group of columns where the rows can be divided into two groups — one with the dominant allele forming a bicluster of ones and the other with alternative alleles forming a bicluster of zeroes. Each strip partitions the reads into a group with the dominant allele and a group with the alternative allele.

By design, a strip groups multiple positions that are highly correlated. Larger strips include a greater number of correlated positions and likely witness the presence of actual SNPs in a multi-haplotype region. On the other hand, narrow strips (i.e., characterized by a number of columns below the width threshold for biclustering) are excluded because they lack sufficient statistical significance to confirm that their columns represent actual SNPs.

Read characteristic vectors

Given a set of strips it is then possible to characterize each read with a binary vector whose size is equal to the number of strips. If the i-th strip of a read is a quasi-bicluster



(a)

	St_1	St_2	St_3	St_4
r_1	1	0	0	1
r_2	1	1	1	0
r_3	0	1	0	1
r_4	1	1	1	0
r_5	1	0	0	1

(b)

Fig. 2 Separation of the domain into four vertical strips. (a) Each strip is a bipartition of rows. Quasi-biclusters of ones (resp. of zeros) are shown in gray (resp. white). Columns in strip 5 (St_5) are not considered because the width of this strip is below the specified threshold. **(b)** Characteristic vectors of the five reads/rows highlighted with a dashed line in the left figure. The comparison of the characteristic vectors results in three strains respectively labelled by a blue square, an orange triangle and a purple circle: the first strain contains r_1 and r_5, the second r_2 and r_4 and the third only r_3.

of ones, the i-th coefficient of this binary vector is set to 1; if not, it is set to 0. This vector is referred to as the *characteristic vector* of the corresponding read. Reads having identical characteristic vectors are grouped together as belonging to the same strain - in biological terms, this corresponds to grouping reads that are identical at all identified polymorphic loci. In practice, and as outlined in section 2.6.2, strains with a small number of reads are not considered, and their reads are possibly re-assigned to other strains.

Figure 2b provides an example of the characteristic vectors for a subset of five reads highlighted in Figure 2a. Among these reads, we observe three strains: r_1 and r_5 compose the first one, r_2 and r_4 the second, and the third strain only contains r_5.

Problem objective

Given three parameters corresponding to the desirable density γ (and sparsity $(1-\gamma)$), and two thresholds to indicate the minimum bicluster width and height, our strategy attempts to cover a maximum width of matrix A with strips.

2.5 A hybrid approach for finding strips

In order to find strips in the matrix, we have developed an Integer Linear Programming approach (denoted here as ILP-QBC), which will be detailed in later. In practice, however, the ILP-QBC does not scale well for large matrices. To overcome this issue, we decided to combine ILP-QBC with a preprocessing step, Hierarchical Cluster Analysis (HCA), a well-known technique frequently used in data mining, to reduce the size of the problem. Figure 3 explains the pipeline.

2.5.1 Strip identification with hierarchical clustering

The search of strips consists of the following two steps:

1. Hierarchical clustering of columns;
2. Checking if groups of columns are strips;

Hierarchical clustering of columns.

We utilize a standard hierarchical clustering to group columns and identify candidate strips. Specifically, we employ the Hamming distance and a complete-linkage strategy. Complete linkage defines the distance between two sub-matrices as the distance between their most distant columns (one from each sub-matrix). Groups of columns are iteratively merged until all groups have more than 35% divergence with all others. Groups of columns that do not satisfy the minimum required number of columns (5 by default) are not further processed by strainMiner.

Checking if groups of columns are strips

For each group of columns identified in the previous step, we apply HCA to partition the reads (rows) into two groups. In the best-case scenario, the group of columns is "unambiguous" and form a strip: the groups form a $(1-\gamma)$-bicluster of zeros and a γ-bicluster of ones. The identified strips are outputted and removed from the matrix, while the remaining "ambiguous" columns are provided as input to the ILP-QBC (Figure 3). The rationale behind this approach is that HCA very efficiently identifies "easy-to-spot" strips, reducing the size of the problem that needs to be solved by ILP-QBC.

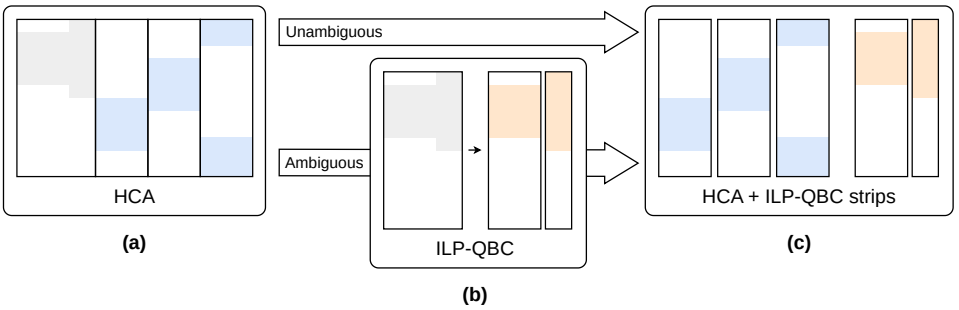


Fig. 3 Hybrid strip search HCA + ILP-QBC. In each of the sub-figure, the colored areas are dense binary sub-matrices. **(a)** The initial matrix with the groups of columns found via HCA. The first one (grey areas) is not a strip, as the bipartitioning does not respect γ. The other three (blue tiles) are strips. **(b)** The remaining matrix is sent to ILP-QBC to find strips respecting γ. In this example, ILP-QBC finds two strips (orange tiles). **(c)** The final set of strips is composed of the HCA-based strips and the ones found by ILP-QBC.

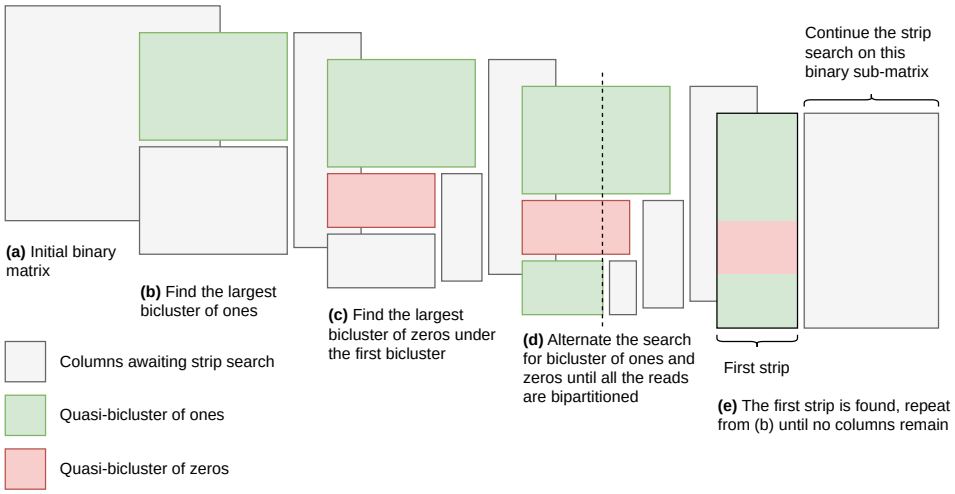


Fig. 4 The ILP-QBC strip iterative search strategy [23].

2.5.2 Strip identification with Integer Linear Programming

Our method, Integer Linear Programming for Quasi-Bicliques identification (ILP-QBC) iteratively identifies strips until all columns of the input matrix have been processed (see Figure 4).

We address the problem of identifying a strip heuristically and iteratively as follows:

1. The largest quasi-bicluster of ones in the matrix is found (Figure 4b);
2. Consider the sub-matrix restricted to the columns of the found quasi-bicluster but consisting of the reads that are not included in it (see the gray sub-matrix highlighted in Figure 4b);. Find the largest quasi-bicluster of zeros (Figure 4c);
3. Restrict the matrix and repeat steps from 2 alternating the search of quasi-bicluster of ones and zeros until no reads remain to be partitioned (Figure 4d).
4. Stop when no further quasi-bicluster can be found (given γ and the minimum height and width)

For the strip under consideration, if the number of remaining reads is below a certain threshold (5 in our case), the strip is considered valid – the remaining reads corresponding to particularly noisy reads. Each remaining read is assigned to the quasi-bicluster of ones if it has a majority of ones; otherwise, it is assigned to the quasi-bicluster of zeros. The columns retained in the last iteration define the width of the strip (Figure 4e). They are removed from the matrix and the search for another strip begins.

If the number of remaining columns is above the threshold, it means that no wide enough strip could be found in the matrix, and the search for strips stops. The remaining columns often correspond to loci that are not polymorphic but particularly prone to sequencing errors (e.g. homopolymers).

To find quasi-bicluster of ones or of zeros, we employ an Integer Linear Programming (ILP) model.

ILP model

Given a binary matrix, the purpose of our ILP model is to select a group of rows and columns that maximizes the count of a specific value (either zeros or ones) respecting the γ parameter. For a given matrix $A \in \mathbb{Z}_2^{|R|\times|L|}$ and a threshold γ, we use binary variables x_{ij}, u_i and v_j to denote the selection (value equals 1) or non-selection (value equals 0) of a cell, row, and column, respectively. The following ILP searches for a quasi-bicluster of ones:

$$\max \sum_{i\in R}\sum_{j\in L} a_{ij}x_{ij} \tag{1}$$

$$x_{ij} \leq u_i, \ \forall i \in R, \forall j \in L \tag{2}$$

$$x_{ij} \leq v_j, \ \forall i \in R, \forall j \in L \tag{3}$$

$$x_{ij} \geq u_i + v_j - 1, \ \forall i \in R, \forall j \in L \tag{4}$$

$$\sum_{i\in R}\sum_{j\in L}(1-a_{ij})x_{ij} \leq (1-\gamma) \times \sum_{i\in R}\sum_{j\in L} x_{ij} \tag{5}$$

$$u_i, v_j \in \{0,1\}, \ x_{ij} \in \{0,1\} \quad \forall i \in R, \ \forall j \in L \tag{6}$$

The function to maximize (1), counts for the number of ones in a sub-matrix determined by the binary variables having value 1. Constraints (2), (3), (4) mean that cell (i, j) is selected into the solution (i.e., $x_{ij} = 1$) if and only if both its corresponding row i and column j are also included into the solution (i.e., $u_i = 1$ and $v_j = 1$).

The coefficient a_{ij} represents the value of the cell at position i and j. It is directly used when searching for occurrences of 1s in the matrix. When the search is for 0s, however, the coefficient is reversed to $(1 - a_{ij})$ as follows:

$$\max \sum_{i\in R}\sum_{j\in L}(1-a_{ij})x_{ij} \tag{7}$$

Constraint (5) ensures that the sub-matrix contains at least a proportion γ of ones. This constraint can also be reversed when necessary to ensure a minimum proportion of zeros:

$$\sum_{i\in R}\sum_{j\in L} a_{ij}x_{ij} \leq (1-\gamma) \times \sum_{i\in R}\sum_{j\in L} x_{ij} \tag{8}$$

Complexity analysis

Biclustering is closely related to bipartite graph partitioning. If we consider reads and positions as vertices of a graph and our binary matrix as the adjacency matrix of this graph, finding a submatrix of ones in the adjacency matrix is equivalent to finding a biclique in a bipartite graph. Finding the maximum biclique is an NP-hard problem [27]. Our problem is even more challenging, as we allow a proportion γ of non-existing links, effectively searching for a maximum quasi-biclique.

2.6 From strips to strain groups

2.6.1 Read characteristic vectors

The final strip set contains the strips generated both from HCA and ILP-QBC. Each strip corresponds to a biclustering of the rows: a row is either labelled one or zero. We associate a binary characteristic vector to each read, where the vector's length corresponds to the number of strips (see Figure 2).

2.6.2 Definition of strain groups

Two reads participate in the same strain group if they have identical characteristic vectors - biologically, this means that these reads are identical at all polymorphic positions. In practice, however, some groups have fewer reads than a specified threshold (5 in our case). These orphan reads generally correspond to noisy reads that have been erroneously grouped in some strips. They are rescued by being reassigned them to bigger groups.

Reassigning orphan reads

To reassign reads, we consider the binary sub-matrix induced by the reads of each big group. The process can be summarized as follows:

1. Compute the representative binary row-vector V_g of each big group g. The j-th column of the row-vector equals to:

$$V_g[j] = \left\lfloor \frac{\sum_{0 \leq i < m} A_g[ij]}{m} \right\rceil$$

   where $A_g \in \mathbb{Z}_2^{m \times |L|}$ is the binary matrix induced by group g with m reads, and $\lfloor x \rceil$ rounds x to its nearest integer.
2. Compute the Hamming distance between all the orphan reads and all the representatives V_g.
3. For each orphan read and its Hamming-distance-closest-group g:
   - if the Hamming distance is less than a given threshold (here 0.1), assign the read to group g;
   - otherwise, do not assign the read to any group.

2.7 Producing a strain-level assembly

At this point, reads are partitioned into strain groups on all windows.

Merging windows

To simplify the assembly process, consecutive identical windows are combined. Specifically, two consecutive windows are deemed identical if they partition the reads into the same groups. In practical terms, two groups are considered identical if more than

70% of the reads from the group in the first window are also found in a corresponding group in the second window, and vice versa (considering only the reads present in both windows). These identical windows are then merged to create longer windows.

Reconstructing the sequences

In each window, a new contig is generated for each group of reads. These contigs are computed using Racon [28] to polish the corresponding (strain-oblivious) reference sequence towards the strain-specific sequence the group of reads represents. Consequently, each original contig from the draft assembly can be divided into multiple windows, and each window can contain several strain-specific contigs (one for each strain). Finally, the resulting contig graph is processed through GraphUnzip [29] to resolve repeats and generate a more contiguous final set of contigs.

3 Results

3.1 Datasets

We decided to evaluate strain-level assembly methods on mock communities based on real and simulated datasets of varying complexity in terms of error rate, number of strains, and level of abundance. This approach allows us to estimate the performance of the methods in a controlled scenario where the sequences of the ideal output are known a priori.

The first dataset we considered is a mixture of five *Vagococcus fluvialis* strains barcoded and sequenced in [30]. These genomes were sequenced with barcodes using a R9.4.1 Nanopore flowcell. By ignoring the barcodes, we obtain a simple mock community containing five different strains of roughly the same abundance. Specifically, this dataset is characterized by three strains whose genomes are almost identical, likely turning the problem into the distinction of three *V. fluvialis* strains, one of which is dominant.

The second dataset is the Zymobiomics gut microbiome standard, a mock community sequenced independently with Nanopore 10.4.1 (error rate of 2.5%) and Nanopore R9.4.1 (error rate of 5%) flowcells. These two samples are available in the European Nucleotide Archive (ENA) with accession numbers `SRR17913199` and `SRR17913200`, respectively. This community consists of 21 genomes of bacteria, archea and yeast, with high variability in terms of abundance. Among the bacteria there are five different strains of *Escherichia coli* characterized by equal abundance, making these samples an ideal dataset for comparing strain separation techniques with two different error rates.

The third type of dataset is simulated and aims to evaluate strainMiner not only with respect to a higher number of strains but also in presence of a strain present at different levels of abundance. More precisely, we adopted a protocol similar to the one outlined in [4] and [14]. We thus simulated a mock community based on 10 strains of *Escherichia coli* to investigate the impact of strain coverage on the ability to recover strain genomes. The *E. coli* strains are the same as those previously employed to evaluate HairSplitter [14] and their complete reference sequences were retrieved from NCBI. More precisely, these include the strains 12009 (GCA_000010745.1), IAI1 (GCA_000026265.1), F11 (GCA_018734065.1), S88

(GCA_000026285.2), Sakai (GCA_003028755.1), SE15 (GCA_000010485.1), UMN026 (GCA_000026325.2), HS (GCA_000017765.1), K12 (GCF_009832885.1), and *Shigella flexneri* (GCF_000006925.2). Then, for each reference sequence, we simulated a 50X depth of coverage of Nanopore reads with 5% error rate. Reads were generated with Badreads [31] using the "nanopore2023" model. All the set of simulated reads were then merged into a single mixture. In order to evaluate the influence of coverage on assembly completeness, we created four additional 10-strain mixures by downsampling exclusively the 12009 strain at 30X, 20X, 10X, and 5X. All simulated reads are available at https://zenodo.org/records/10362565.

3.2 Assembly of the datasets

All datasets were assembled using metaFlye (v2.9.2-b1786) with parameters `--meta` and `--nano-raw`. We chose metaFlye as it is the only long-read metagenome assembler thought to work with Oxford Nanopore reads. The obtained (species-level) assembly was used as input for all the strain-aware assemblers we evaluated. Moreover, an alignment of the reads against the input assembly was produced with Minimap2 [32] (parameter `-x map-ont`) for the software that required it.

We ran the latest available versions of Strainberry (v1.1), Floria (v0.0.1), HairSplitter (v1.9.4), and strainMiner (v1.6.10) with default parameters. Since Strainberry and Floria have been designed to phase no more than 5 strains by default, when running on the 10-strain *E. coli* datasets, we additionally provided the parameters `-n 10` and `-p 10`, respectively, to increase this limit.

It is important to note that Floria does not perform any SNP calling nor does it output a base-level assembly, but rather provides a collection of haplotype-resolved read clusters. SNP calling was carried out using Longshot [33] (v1.0.0). Base-level assemblies of each read cluster were, instead, produced with wtdbg2 [34] (v2.5) following the assembly pipeline suggested in Floria's documentation. The set of assembled haplotypes was finally complemented with the metaFlye contigs that were not phased (i.e., not part of Floria's output).

3.3 Evaluation Metrics

Evaluating metagenome assemblies is a complex task, specifically in a strain-aware context, due to the limited knowledge of the organisms within a metagenomic sample. Standard qualitative metrics should in fact be treated with caution as they might be the result of strain differences rather than errors. For this reason, we assessed the performance of the strain-aware assembly methods on real and simulated mock communities for which we precisely know the sequences of the strains we aim to reconstruct.

For each generated assembly, we computed the following metrics: assembly size, N50, reference fraction percentage, duplication ratio, number of misassemblies, number of mismatches, and number of indels. The assembly size and N50 (*i.e.*, the length such that all contigs of that length or longer cover at least half of the assembly size) provide a quantitative view of the assembly. The other metrics, instead, provide a more qualitative assessment.

Table 1 MetaQUAST metrics on the real and simulated datasets. For each assembly we report the following metrics: assembly size (Total size), N50, reference fraction, duplication ratio (Dup. ratio), number of misasseblies (# Mis.), number of mismatches per 100 kb, and number of indels per 100 kb. The best values among of the four strain-separated assemblies is in bold font.

	Assembler	Total size (Mb)	N50 (kb)	Reference frac. (%)	Dup. ratio	# Mis.	Mismatches /100 kb	Indels /100 kb
V. fluvialis (14 Mb)	metaFlye	4.57	151	26.9	1.036	40	360.12	406.10
	Strainberry	5.71	104	33.1	1.112	45	78.19	513.38
	Floria	8.03	**142**	47.7	1.117	**20**	102.61	636.40
	HairSplitter	**9.34**	74	**58.2**	**1.066**	31	102.49	410.95
	strainMiner	8.84	30	54.5	1.080	48	**50.80**	**340.26**
Zymo Q9 (78 Mb)	metaFlye	65.9	1797	63.0	1.001	136	96.97	58.52
	Strainberry	**79.9**	**224**	61.2	1.170	122	133.31	110.07
	Floria	76.0	64	58.6	1.212	114	230.67	245.47
	HairSplitter	94.6	33	**76.7**	1.179	114	115.38	76.49
	strainMiner	76.7	220	73.4	**1.044**	**95**	**69.90**	**55.46**
Zymo Q20 (78 Mb)	metaFlye	60.4	388	60.2	1.007	142	115.86	76.49
	Strainberry	69.4	**117**	69.4	1.037	141	109.49	67.46
	Floria	**71.0**	59	70.7	1.050	117	80.41	79.81
	HairSplitter	69.2	68	**70.9**	**1.013**	113	69.17	66.36
	strainMiner	69.5	51	70.8	1.022	**109**	**67.98**	**65.14**
E. coli 10 strains (48 Mb)	metaFlye	14.0	70	25.2	1.029	94	477.68	310.42
	Strainberry	19.6	**63**	34.2	**1.082**	175	367.48	138.68
	Floria	35.3	50	60.3	1.155	**147**	171.95	104.12
	HairSplitter	54.8	44	**93.6**	1.182	311	87.12	**52.30**
	strainMiner	**50.1**	56	91.1	1.127	335	**81.73**	72.80

These metrics were obtained with MetaQUAST [35] (v5.2.0) with the option `--unique-mapping` to ensure that each assembled sequence (a contig or part of it) is mapped exclusively to the best location among the reference sequences. This is crucial because MetaQUAST, which relies on sequence alignment, may suffer from sub-optimal mappings on very similar references, such as strains of the same species. For this reason, we complemented MetaQUAST's evaluation metrics by computing the k-mer completeness ($k = 27$) with KAT [36] (v2.4.2). This metric represents the percentage of k-mers in the reference genomes that are also present within an assembly and provides a more accurate view on the strain-specific content that was successfully recovered.

3.4 Assembly evaluation

Table 1 summarizes the MetaQUAST metrics computed for each strain-level assembly tool on each of the evaluated datasets. To manage the size of the table, in the case of the 10-strain *E. coli* datasets, we display only the one in which all strains have the same abundance. The downsampling experiments, however, exhibited similar statistics.

On the *V. fluvialis* dataset, strainMiner is able to yield the lowest amount of mismatches and indels, while HairSplitter is able to achieve the best results in terms

of assembly size, reference fraction, and duplication ratio. strainMiner however is on par with HairSplitter with respect to these metrics. Floria on the other hand is able to provide the most contiguous assembly (N50 equal to 142 kbp) and the lowest number of misassemblies at the expense of a more duplicated and less accurate assembly. While displaying an average performance on most the evaluation metrics, Strainberry was only able to recover the 33% of the reference sequences. The high reference fraction of both strainMiner and HairSplitter, compared to the other tools, is also confirmed by the highest k-mer completeness (Figure 5).

On the Q9 and Q20+ versions of Zymobiomics datasets, results follow the same trend as for the *V. fluvialis* dataset. Specifically, strainMiner generates the most accurate assembly, as witnessed by the lowest number of misassemblies, mismatches, and indels. In terms of contiguity, Strainberry achieves the highest N50 but also at the expense of a high number of misassemblies. There are however some key differences for the assemblies generated with the Q9 and Q20+ Nanopore reads. In the first case, strain-level assemblers are characterized by a lower reference fraction and higher duplication ratio. The only exception is strainMiner, which is able to provide comparable results, thus proving to be more tolerant to lower error rates. As for the *V. fluvialis*, the k-mer completeness is consistent with the highest reference fraction of strainMiner and HairSplitter for both the Q9 and Q20+ datasets (Figure 5).

The simulated 10-strain *E. coli* dataset offered a more challenging scenario due to the presence of a higher number of closely related genomes. Table 1 shows that strainMiner and HairSplitter are the only two tools able to achieve a high reference fraction ($> 90\%$), thus recovering almost completely the 10 different strains. They are also the two tools with the lowest number of mismatches and indels (as for the other datasets). Strainberry and Floria, on the other hand, display a much lower reference coverage (34.2% and 60% respectively) and seem to struggle with high numbers of conspecific strains.

Finally, the right-hand side of Figure 5 shows the k-mer completeness of the 10-strain dataset in which the *E. coli* 12009 strain is characterized by different level of abundance. As expected, strainMiner and HairSplitter exhibit a better ability to retrieve strains with low coverage, even when it is as low as 5X (representing only 1.1% of the total mix), and displayed a k-mer completeness always higher than 0.9. Nevertheless, a positive correlation between the coverage and completeness is noticeable for all the evaluated methods.

Overall, there is no tool that outperforms the others with respect to all evaluation metrics on the different datasets. All the evaluated tools offer different trade-offs between contiguity and accuracy. Floria, for example, is able to achieve higher contiguity and a comparable number of misassemblies at the expense of a lower base-level accuracy and reference coverage. On the other hand, when looking at qualitative metrics, strainMiner generates more accurate assemblies (or comparable with the other competing tools) and a low amount of duplicated sequences, especially with higher error rates.

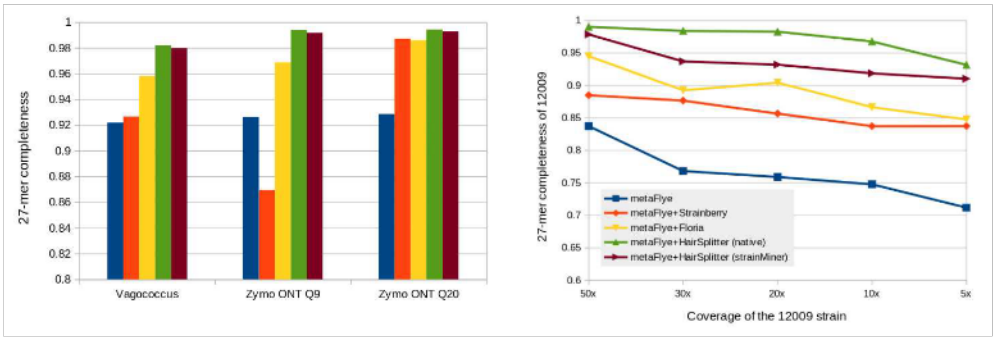


Fig. 5 Assembly k-mer completeness. On the left-hand side, the 27-mer completeness of assemblies obtained from real sequencing data is depicted. The level of completeness in the Zymo communities is based exclusively on the *E. coli* genomes within the samples. On the right-hand side, the 27-mer completeness of the 12009 strain is shown for the 10-strain simulated *E. coli* datasets. The analysis was performed for varying depth of coverage of the 12009 strain (x-axis), while the other nine strains were consistently kept at 50X.

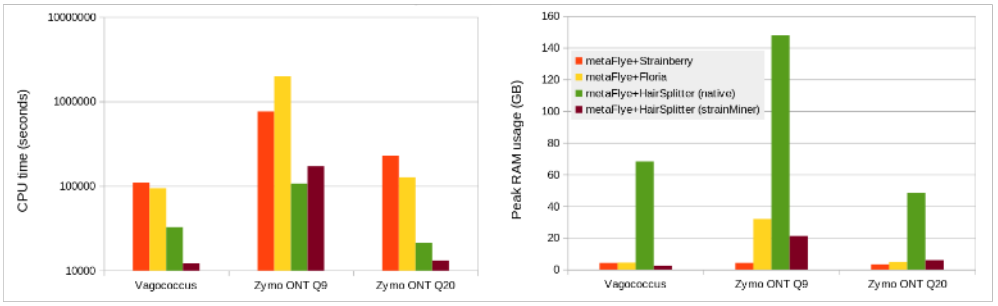


Fig. 6 CPU time and maximum memory usage. Comparison of Strainberry, Floria, HairSplitter, and strainMiner on the three real datasets in terms of CPU time (left) and maximum memory usage (right). The legend applies to both the plots.

3.5 Resource usage

All results were obtained on a server housing 16 Intel Xeon CPUs with four cores each, running at 2.7 GHz. 3.1 TB of RAM was available.

Although strainMiner, like the other competing tools, is trivially parallel, we ran it on a single thread due to limitations imposed by the Gurobi license. For this reason we decided to simply report CPU times.

Across all datasets, strainMiner consistently exhibited a processing speed more than tenfold faster than Strainberry, while its runtime was approximately on par with that of HairSplitter (Figures 6).

On the real sequencing data, strainMiner used between 7 and 30 times less peak memory than HairSplitter (Figure 6).

As a whole, strainMiner significantly diminishes the memory usage of the HairSplitter pipeline without impacting negatively on its speed and arguably improving the quality of the assembly.

4 Discussion

In this work we introduced strainMiner, a method to assemble individual strain genomes from metagenomic sequencing data. Unlike other metagenome assembly methods, strainMiner is based on a novel formulation of the "strain separation" problem as a tiling problem on a binary matrix. We proposed and implemented an Integer Linear Programming (ILP) model in order to efficiently partition such a matrix and to cluster sequences (reads) that likely belong to the same haplotype. The ILP-based formulation allows us to exploit a well-established and highly optimized solver such as Gurobi. This, along with the use of heuristics inspired from data mining, allow strainMiner to require considerably less time and memory compared to other competing software.

In order to assess its capability to distinguish and reconstruct strains, strainMiner is implemented as a fork of HairSplitter (a previously developed tool for strain-level metagenome assembly), from which we replaced the read-clustering step with our approach. On both real and simulated datasets, strainMiner compared favorably to state-of-the-art methods in terms of strain recovery and base-level accuracy. We also showed that strainMiner's output is less affected by reads with high error rates or metagenomes characterized by a high number of distinct strains. At the same time, strainMiner is able to recover strains with a depth of coverage as low as 5X.

Nevertheless, the assemblies obtained with strainMiner had often much lower contiguity compared to the one obtained with HairSplitter. This could be due to the fact that strainMiner is based on the version 1.6.10 of HairSplitter, while in our comparison we used the latest available version (1.9.4). Upgrading strainMiner to this version might further improve output's contiguity. Moreover, it is also possible that the properties of the read clusters computed by strainMiner could be quite different compared to those obtained with HairSplitter. A tailored scaffolding step could thus improve contiguity and would merit further investigation. A possible approach would be to consider overlapping fixed-length windows. As a matter of fact, the use of non-overlapping windows generates clean input matrices for the ILP solver but completely relies on the GraphUnzip module of HairSplitter to improve assembly contiguity. Considering overlapping windows, however, could better take advantage of the co-occurrence of strain-specific nucleotides, allowing to identify longer haplotypes beforehand.

Finally, one additional limitation is the use of the Gurobi solver: an academic license is free but limited to three instances of Gurobi running at the same time. Attempts to use the free CBC solver showed a decrease in performance.

Software availability

strainMiner is open source and available online with the GPL3 licence. The strainMiner used to generate the results is available at https://github.com/rolandfaure/strainminer. A more recent version, under development, is available at https://gitlab.com/haplotype-tiling/strainminer-py.

Acknowledgements

We wish to thank Dominique Lavenier, who formulated the first version of the optimization problem.

We acknowledge the GenOuest bioinformatics core facility https://www.genouest.org for providing the computing infrastructure.

References

[1] Ghurye, J. S., Cepeda-Espinoza, V. & Pop, M. Metagenomic assembly: Overview, challenges and applications. *The Yale journal of biology and medicine* **89**, 353–362 (2016).

[2] Almeida, A. *et al.* A unified catalog of 204,938 reference genomes from the human gut microbiome. *Nature biotechnology* **39**, 105–114 (2021).

[3] Olson, N. D. *et al.* Metagenomic assembly through the lens of validation: recent advances in assessing and improving the quality of genomes assembled from metagenomes. *Briefings in bioinformatics* **20**, 1140–1150 (2019).

[4] Vicedomini, R., Quince, C., Darling, A. E. & Chikhi, R. Strainberry: automated strain separation in low-complexity metagenomes using long reads. *Nature Communications* **12**, 4485 (2021). URL https://www.nature.com/articles/s41467-021-24515-9.

[5] Albanese, D. & Donati, C. Strain profiling and epidemiology of bacterial species from metagenomic sequencing. *Nature communications* **8**, 2260 (2017).

[6] Sonnenborn, U. Escherichia coli strain nissle 1917—from bench to bedside and back: history of a special escherichia coli strain with probiotic properties. *FEMS microbiology letters* **363**, fnw212 (2016).

[7] Frank, C. *et al.* Epidemic profile of shiga-toxin–producing escherichia coli o104:h4 outbreak in germany. *New England Journal of Medicine* **365**, 1771–1780 (2011). URL https://doi.org/10.1056/NEJMoa1106483. PMID: 21696328.

[8] Quince, C. *et al.* DESMAN: a new tool for de novo extraction of strains from metagenomes. *Genome Biology* **18**, 181 (2017). URL http://genomebiology.biomedcentral.com/articles/10.1186/s13059-017-1309-9.

[9] Quince, C. *et al.* Metagenomics Strain Resolution on Assembly Graphs. preprint, Bioinformatics (2020). URL http://biorxiv.org/lookup/doi/10.1101/2020.09.06.284828.

[10] Baaijens, J., Aabidine, A., Rivals, E. & Schönhuth, A. De novo assembly of viral quasispecies using overlap graphs. *Genome Research* (2017).

[11] Kang, X., Luo, X. & Schönhuth, A. StrainXpress: strain aware metagenome assembly from short reads. *Nucleic Acids Research* **50**, e101–e101 (2022). URL https://academic.oup.com/nar/article/50/17/e101/6625806.

[12] Kazantseva, E., Donmez, A., Pop, M. & Kolmogorov, M. stRainy: assembly-based metagenomic strain phasing using long reads. preprint, Bioinformatics (2023). URL http://biorxiv.org/lookup/doi/10.1101/2023.01.31.526521.

[13] Shaw, J., Gounot, J.-S., Chen, H., Nagarajan, N. & Yu, Y. W. Floria: Fast and accurate strain haplotyping in metagenomes. *Bioinformatics* **40**, i30–i38 (2024).

[14] Faure, R., Lavenier, D. & Flot, J.-F. Hairsplitter: haplotype assembly from long, noisy reads. *bioRxiv* (2024). URL https://www.biorxiv.org/content/early/2024/06/14/2024.02.13.580067.

[15] Bertrand, D. *et al.* Hybrid metagenomic assembly enables high-resolution analysis of resistance determinants and mobile elements in human microbiomes. *Nature Biotechnology* **37**, 937–944 (2019). URL http://www.nature.com/articles/s41587-019-0191-2.

[16] Kolmogorov, M. *et al.* metaFlye: scalable long-read metagenome assembly using repeat graphs. *Nature Methods* **17**, 1103–1110 (2020). URL https://www.nature.com/articles/s41592-020-00971-x.

[17] Bickhart, D. M. *et al.* Generating lineage-resolved, complete metagenome-assembled genomes from complex microbial communities. *Nature biotechnology* **40**, 711–719 (2022).

[18] Feng, X., Cheng, H., Portik, D. & Li, H. Metagenome assembly of high-fidelity long reads with hifiasm-meta. *Nature Methods* **19**, 1–4 (2022).

[19] Benoit, G. *et al.* High-quality metagenome assembly from long accurate reads with metamdbg. *Nature Biotechnology* 1–6 (2024).

[20] Bansal, V. Hapcut2: A method for phasing genomes using experimental sequence data. *Methods in molecular biology* **2590**, 139–147 (2022).

[21] Raghavan, U. N., Albert, R. & Kumara, S. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* **76**, 036106 (2007).

[22] Biemann, C. Chinese whispers: An efficient graph clustering algorithm and its application to natural language processing problems. *Proceedings of TextGraphs* 73–80 (2006).

[23] Truong, T. K. M., Faure, R. & Andonov, R. *Assembling close strains in metagenome assemblies using discrete optimization*, 15th International Conference on Bioinformatics Models, Methods and Algorithms, BIOINFORMATICS, February 21-23, 2024, Rome, Italy. URL https://bioinformatics.scitevents.org.

[24] Feng, Z., Clemente, J., Wong, B. & Schadt, E. Detecting and phasing minor single-nucleotide variants from long-read sequencing data. *Nature Communications* **12**, 3032 (2021).

[25] Fix, E. & Hodges, J. L. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique* **57**, 238–247 (1989). URL http://www.jstor.org/stable/1403797.

[26] Troyanskaya, O. *et al.* Missing value estimation methods for dna microarrays. *Bioinformatics* **17**, 520–525 (2001). URL https://doi.org/10.1093/bioinformatics/17.6.520.

[27] Peeters, R. The maximum edge biclique problem is NP-complete. *Discrete Applied Mathematics* **131**, 651–654 (2003).

[28] Fang, L. & Wang, K. Polishing high-quality genome assemblies. *Nature Methods* **19**, 649–650 (2022). URL https://www.nature.com/articles/s41592-022-01515-1.

[29] Faure, R., Guiglielmoni, N. & Flot, J.-F. Graphunzip: unzipping assembly graphs with long reads and hi-c. *bioRxiv* 2021–01 (2021).

[30] Rodriguez Jimenez, A. *et al.* Comparative genome analysis of vagococcus fluvialis reveals abundance of mobile genetic elements in sponge-isolated strains. *BMC Genomics* **23** (2022).

[31] Wick, R. Badread: simulation of error-prone long reads. *Journal of Open Source Software* **4**, 1316 (2019). URL http://joss.theoj.org/papers/10.21105/joss.01316.

[32] Li, H. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* **34**, 3094–3100 (2018). URL https://academic.oup.com/bioinformatics/article/34/18/3094/4994778.

[33] Edge, P. & Bansal, V. Longshot enables accurate variant calling in diploid genomes from single-molecule long read sequencing. *Nature communications* **10**, 4660 (2019).

[34] Ruan, J. & Li, H. Fast and accurate long-read assembly with wtdbg2. *Nature methods* **17**, 155–158 (2020).

[35] Mikheenko, A., Saveliev, V. & Gurevich, A. Metaquast: Evaluation of metagenome assemblies. *Bioinformatics* **32**, btv697 (2015).

[36] Mapleson, D., Accinelli, G., Kettleborough, G., Wright, J. & Clavijo, B. Kat: A k-mer analysis toolkit to quality control ngs datasets and genome assemblies. *Bioinformatics (Oxford, England)* **33** (2016).