Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Hybrid genome assembly

Magisterská práca

2020
Matúš Zeleňák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Hybrid genome assembly

Magisterská práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Tomáš Vinař PhD.

Bratislava, 2020
Matúš Zeleňák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Matúš Zeleňák
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Algorithms for Assembly of Hybrid Genomes from Long Reads
Algoritmy na skladanie hybridných genómov z dlhých čítaní

Anotácia: Hybridné genómy obsahujú dve alebo viacero mierne odlišných kópií tej istej množiny chromozómov. V závislosti od miery podobnosti jednotlivých kópií, tradičné algoritmy na skladanie genómov buď premiešajú dve kópie alebo kópie zoskladajú ako separátne sekvencie. Cieľom tejto práce je navrhnúť nové algoritmy, ktoré využijú novú informáciu, ktorú do tohto problému prenieslo sekvenovanie dlhých čítaní.

Vedúci: doc. Mgr. Tomáš Vinař, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 15.12.2017

Dátum schválenia: 15.12.2017 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Matúš Zeleňák
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Algorithms for Assembly of Hybrid Genomes from Long Reads

Annotation: Hybrid genomes can contain two or more slightly different copies of the same set of chromosomes. Depending on the level of similarity between the copies, traditional algorithms for genome assembly either mix the two copies or assemble the copies separately. The goal of this thesis is to develop new algorithms by utilizing new information brought to the problem with long read sequencing.

Supervisor:	doc. Mgr. Tomáš Vinař, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	15.12.2017

Approved: 15.12.2017 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgement: I would like to thank Tomáš Vinař and Broňa Brejová for their guidance and patience and to Gareth Cooker for creating wonderful soundtracks that would carry me through the feverish nights of coding.

Abstrakt

Hybridné genómy predstavujú výzvu pri skladaní haplotypov organizmov, keďže klasické nástroje na skladanie buď kópie chromozómov premiešajú do jednej sekvencie, alebo oddelia úplne. V našej práci pristupujeme ku problému kategorizácie čítaní podľa ich príslušnosti k hybridu ako ku grafovému problému. Na základe špecifických k-tic zostrojíme spojenia medzi čítaniami a následne vytvárame čoraz väčšie komponenty, pričom sa snažíme zachovať nízku mieru medzihybridnej kontaminácie. Naše výsledky na zmiešaných simulovaných čítaniach dvoch kmeňov E.coli baktérie ukazujú, že kategorizácia čítaní diploidného genómu je v princípe možná s vysokou mierou presnosti a malými výpočtovými požiadavkami. Naše metódy aplikujeme na dosiaľ nepreskúmanú kvasinku Magnusiomyces spicifer a vyhodnotíme ako výsledky, tak aj možný budúci vývoj v oblasti skladania hybridných genómov.

Kľúčové slová: DNA, Haplotyp, kategorizácia, spektrálne zhlukovanie

Abstract

Hybrid genomes pose a challenge during read assembly of organism haplotypes, as standard assembly tools either mix the chromosome copies into one sequence, or separate the copies entirely. In our work we approach the problem of categorizing reads according to their origin hybrid as a graph clustering problem, using specific k-mers to construct connections between the reads and subsequently creating incrementally larger components while trying to maintain low cross-hybrid read contamination. Our results on mixed simulated reads from different E.coli strains show that in principle categorization of diploid genomes is possible with high degree of accuracy and small computational cost. We apply our methods to a novel yeast strain Magnusiomyces spicifer and examine both the results and possible future direction of development in the area of hybrid genome assembly.

Keywords: DNA, Haplotype, categorization, spectral clustering

Contents

List of Figures

List of Tables

Introduction

0.1 DNA

All known living creatures store the information essential for their correct functioning (such as growth and reproduction) in a molecule called deoxyribonucleic acid (or DNA for short).

This molecule consists among other parts of four building blocks called bases. These are adenine, thymine, cytosine and guanine (for brevity annotated as A, T, C, G respectively) . DNA can be visually imagined as a helix structure of two strands. Each strand contains a sequence of aforementioned bases and it encodes the functions of the carrier organism. The two strands complement one another, with base pairs $\{A, T\}$ and $\{C, G\}$ connecting them along their entire length.[3]

For purposes of our work, we will refer to the genome as a set of strings consisting of letters $\{A, C, G, T\}$. Each such string is called a DNA sequence, or sequence for short. Note, that to obtain the full information contained in the genome, we need to know the composition of only one of the strands, as the other can be determined through complementary base pairs.

0.2 Chromosomes, diploid and polyploid organisms

The genome of an organism consists of one or more DNA molecules. Each molecule is encased in a structure called chromosome. Chromosomes themselves have no physical connections to one another. For the purpose of our work, we will use the term "chromosome" to refer to both the organic structure and the sequence contained within interchangeably.

Simple prokaryotic organisms such as bacteria have only one chromosome, humans for example have 23 pairs of chromosomes and there are species of fish that have over a hundred.

Chromosome pairs are of particular interest for us: such pairs are created during reproduction, when the offspring organism receives one copy of each chromosome from both parents. Lifeforms with this behaviour are called diploid, because the offspring

Platform	**Instrument**	**Avg. r. length**	**Error type**	**Error rate**
Illumina	HiSeq	150	mismatch	0.1%
PacBio	RS II P6 C4	13500	indel	12%
Oxford Nanopore	MinION Mk	9545	indel/mismatch	12%

Table 1: List of sequencing technolofies

ends up with two copies of each chromosome. A set of chromosomes inherited from one parent is called a haplotype.

From the description above, it closely follows that determining a complete haplotype for a multi-chromosome organism from the offspring genome alone proves difficult, if not straight up impossible. While it is in theory possible to find haplotype pairs for all the chromosomes, determining which of the two chromosome sequences belongs to each parent is a coin toss, unless we also possess the reference genome of both the parents with which we can perform alignment.

0.3 Genome sequencing and assembly

In an effort to retrieve a genome, many different sequencing methods have been developed. While every method works differently, in the end their output can be converted into base-called reads, which for our purposes can be simply represented as strings of letters $\{A, C, G, T\}$. In theory, each such read can be mapped back to an interval of bases in the reference genome. In practice however, sequencing errors often render such mapping difficult.

In order to increase the likelihood that every base position (nucleotide) in the genome has been read, prior to sequencing the extracted DNA samples are amplified in the process that creates copies of DNA fragments.[18] Amplification is performed to a degree where a desired coverage is obtained, where coverage refers to the expected number of occurrences of any one base position in the resulting genome reads. Coverage for a specific genome of length G and a sequencing platform producing N reads with average length R can be therefore be computed as $\frac{N \cdot R}{G}$.

Another benefit of amplification is that by having several reads of a particular base position sequencing errors can be corrected.

Sequencing methods differ in parameters such as error distribution, read length distribution, cost per sequenced base etc. Table 1 presents selected few methods (and associated devices) along with their characteristics.[13]

Chapter 1

Problem statement and related work

1.1 Problem statement

Given a set of reads R and an unknown function $h : R \rightarrow \{A, B\}$ assigning a read to its haplotype, we are tasked with partitioning R into two or more disjoint subsets $R_1, R_2, ...R_n$ such that:

1. n is minimized
2. $abs(|\{r \in R_i : h(r) = A\}| - |\{r \in R_i : h(r) = B\}|)$ is maximized

In other words, we want to create at best two subsets of R, while keeping the proportion of reads from one haplotype in every subset as high as possible.

1.2 Related work

We draw inspiration from the "Complete assembly of parental haplotypes with trio binning".[14] In their work, authors managed to isolate haplotype genomes of a hybrid offspring between two cattle subspecies by identifying haplotype-specific k-mers from high-quality reads of both the parents, and using them to bin long reads from the offspring. The main distinguishing factor of our work is that we do not have access to the parental genomes of the sequenced offspring and are instead attempting to identify haplotype-specific k-mers from the high-quality reads of the offspring itself, relying on statistical characteristics emerging from sequence amplification in the process of sequencing.

Chapter 2

Used datasets and prerequisites

2.1 Used datasets

In order to confidently validate if our methods are successful, we need to know the parental haplotypes of our subject organism, so that we can align the categorized reads to a reference. Obtaining read data of a diploid organism for which we also know the parental haplotype genomes is however difficult. As will become apparent later, we moreover require reads of the same sample from two different sequencing platforms, which narrows down the options for real datasets even further. As a consequence, we decided to treat two E.coli strains as our reference haplotypes and use reads created by simulation software.

The strains used were MG1655 and UTI89. These strains were chosen for several reasons:

1. High quality assemblies exist - lack of unknown bases means our read simulators won't create gaps between reads that would result in multiple contings during assembly, making not only the evaluation of our methods difficult, but also rendering some of our methods invalid

2. The genomes of these strains are very similar (megablast[5] reports 97.87% identity), which in case of successful categorization gives hope for categorizations of real haplotypes, as those are highly similar as well. Human genome for example can reach heterozygosity as low as 0.1%[14]

To simulate reads, we used the following tools:

1. Art Illumina for Illumina HiSeQ reads[12]

2. Nanosim-H for Oxford Nanopore reads[206]

3. PaSS for PacBio SMRT reads[22]

Name	Reference A	Reference B	Simulator	Coverage
EIL30	E.coli MG1655	E.coli UTI89	Art Illumina	30x
ENP75	E.coli MG1655	E.coli UTI89	Nanosim-H	75x
EPB75	E.coli MG1655	E.coli UTI89	PaSS	75x
RIL30	Random 500000b	Random 500000b	Art Illumina	30x
RNP	Random 500000b	Random 500000b	Nanosim-H	50x

Table 2.1: List of used datasets

All but the Art Illumina sequencers have built-in error profiles for E.coli bacteria, so we expect the simulated reads to be close to reality. One of the main reasons for using simulated reads for the development of our methods is the fact that they also include useful metadata, such as the position in the reference sequence where the reads originate from. We use this particular data point extensively for tuning our algorithm.

In addition to real organism genomes, we also create a small (500000 bases), randomly generated sequence and its slightly mutated (3% substitution rate) copy for demonstration purposes in cases where computational complexity of algorithms does not allow for use of real, larger genomes. Reads from this sequence are of course, simulated as well.

All the datasets are listed in Table 2.1.

2.1.1 Prerequisites

In order for the methodology introduced in our work to function properly, several prerequisites have to be met by the input data:

1. For reads used for calculation of discriminative kmers, there should not be a significant amplification bias of the sequenced genome. We will be using Illumina reads for this purpose. The PCR amplification in Illumina sequencing seems to mostly affect GC-rich regions of genome[9]

2. For reads used in categorization it should hold that for every pair of neighboring SNPs, there exists at least one read that contains them both, otherwise construction of one contig per haplotype chromosome will not be possible. For Nanopore and PacBio reads that average several thousands bases per read, this should not pose a problem even in cases when haplotypes have high identity

Chapter 3

Discriminative k-mers

3.1 Definition and characteristics

A common way of distinguishing between haplotypes is through single nucleotide polymorphisms (or SNPs for short).[7] If we were to align the two haplotypes, SNPs would mark the positions in the alignment where the haplotypes differ by a single base (be it a substitution, deletion or insertion)

During the sequencing, the SNPs are captured in the "context" of their surrounding bases by k-mers - a window of k consecutive bases. If we choose k large enough that a chance of two identical k-mers occurring in both haplotypes by sheer chance (excluding repeats that occur naturally) is negligible, we could expect each SNP to be captured by up to k k-mers. Thanks to the k-mers being unique, the reverse would also be true - every such k-mer uniquely identifies certain SNPs. Therefore, for each haplotype we expect a set to exist consisting of k-mers that do not occur in the other haplotype. We call such k-mers discriminative.

For purposes of k-mer counting we use the notion of a canonical k-mer. A canonical k-mer deterministically represents both a k-mer and its reverse complement, and is usually computed as a lexicographic minimum of the two. Usage of canonical k-mers is

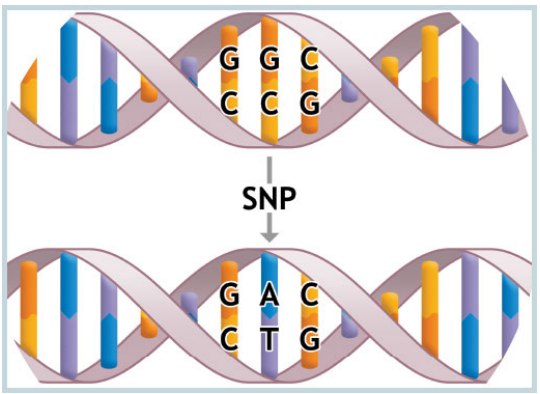

Figure 3.1: SNP[10]

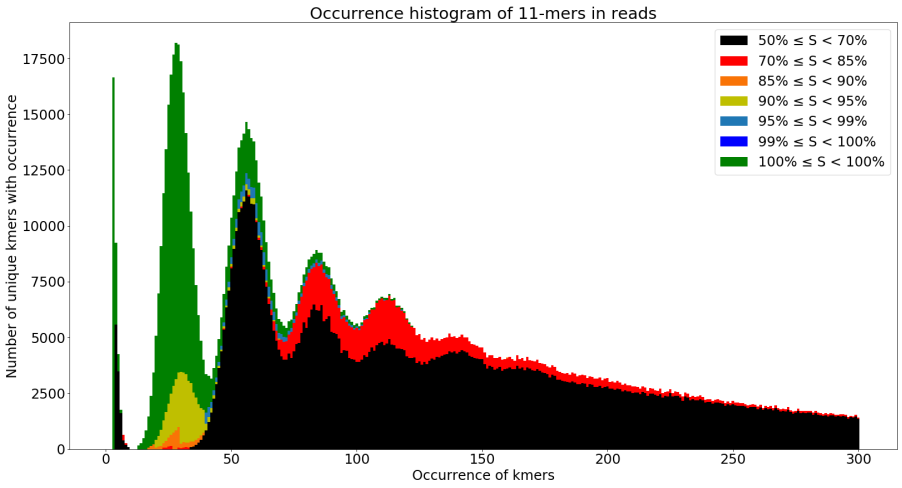

Figure 3.2: k-mer histogram of EIL30 dataset with too small k value. Peaks on the right represent k-mers that occur more than once in one haplotype

necessary, as reads can originate from any of the two strands, yet we want to capture SNPs regardless of the strand we operate on.

Thanks to having a high coverage in the read data set, discriminative k-mers can be differentiated from the rest by computing a histogram of all k-mers contained in the reads. Since discriminative k-mers occur in only one haplotype, while the remaining k-mers occur in both, assuming that all the haplotype regions were amplified uniformly we can expect to find two peaks: one with the x value close to the coverage, and one with a value close to the coverage divided by two. The k-mers that occur at half the coverage have a high probability of being discriminative. Thus, we can obtain a set of discriminative k-mers by collecting those which can be found in the leftmost peak of such histogram.

3.2 Determining the value of k

The first step in obtaining the discriminative k-mer is the selection of the right k value. At first glance it might seem like choosing a very high k would yield good results - after all, we want our k-mers to be unique within a haplotype and a longer k-mer is less likely to occur by chance. However, we have to take into account sequencing errors which will become more pronounced the longer our k-mers are. Setting the value of k too high can potentially lead to a scenario, where many k-mers get pushed into the low-occurrence side of our histogram, because they frequently overlap regions in reads affected by sequencing errors. Setting k too high is also detrimental to performance, as each incrementation has the potential of growing the set of k-mers we need to track.

We choose an empirical approach of choosing k that depends on availability of high-

quality reads, such as those obtained from Illumina technology. In the hypothetical case of perfect sequencing, the number of unique k-mers found in the reads would be at most equal to the true number of k-mers in the genome. Sequencing errors however cause new k-mers to emerge. Using high-quality reads such as Illumina alleviates this problem. We can therefore determine a good value of k by trying incrementally higher values and halting whenever the rate of increase of the number of k-mers reaches a set threshold. The algorithm can be written as follows:

```
def get_unique_kmer_length():
    k = 11
    previous_count = count_kmers(k)
    while k < 33:
        next_count = count_kmers(k + 2)
        if difference(previous_count, next_count) < threshold:
            return k
        previous_count = next_count
        k += 2
    return k
```

Figure 3.3: Algorithm for determining

Note: odd values of k are used frequently, since odd k-mers cannot be the mirror images of their complement. We limit the k value by 32, since we represent k-mers succinctly by 64 bit integers.

In order to count the number of k-mers quickly and without using large amounts of memory, we use the HyperLogLog algorithm.[11] The threshold we set as a halting condition of our algorithm is a 20% increase in the count of k-mers.

For the EIL30 dataset, the determined value of k is 19. Relevant histogram is shown on figure 3.4

The percentages in the histogram legends describe what proportion of a particular k-mers occurrences belong to one of the haplotypes. Every percentage lower than 100 indicates a k-mer that occurrs in both haplotypes.

3.3 Computation of histogram

In order to obtain k-mer occurrence data, we use Jellyfish[17] counting and dumping tools. Knowing how many times each k-mer occurs in the reads, we can compute the histogram. All that remains is to identify the occurrence values resulting in the two peaks, and determining the range of occurrences that we take as a boundary for selecting k-mers which we assume to be discriminative. From now on, we will refer

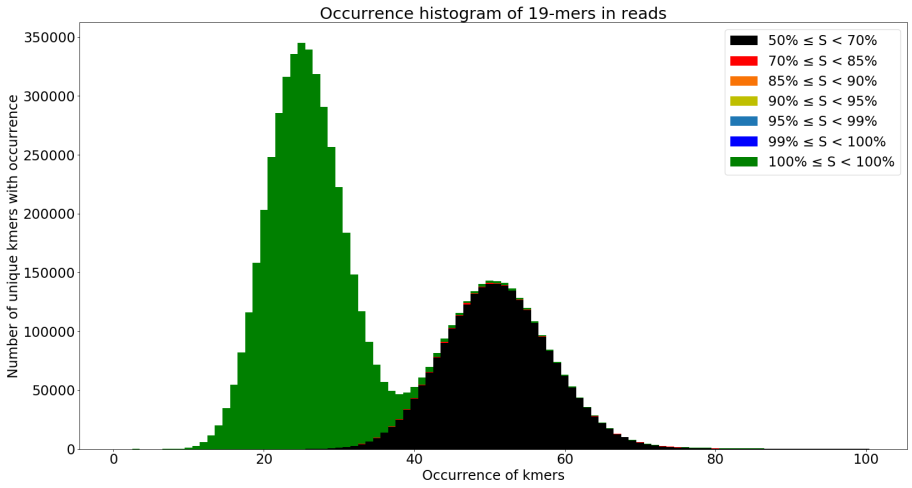

Figure 3.4: k-mer histogram of EIL30 dataset with k=19

to this set of assumed discriminative k-mers as suspected discriminative k-mer set (or SDK set for brevity).

In our experiments taking k-mers that occur between five times and P-times, where P is the number of occurrences at the height of the leftmost peak yielded a good tradeoff between the total number of k-mers exported and the ratio of their contamination.

For EIL30 dataset, the selected occurrence range is 10 to 25, resulting in 2221830 SDKs of which 2221512 are discriminative.

3.4 Discriminative k-mer contamination

As can be seen from figure 3.4, not all k-mers with occurrence lower or equal to half the coverage are discriminative. There is some contamination caused by k-mers that should reside in the peak on the right, but due to sequencing errors and separation of nucleotides caused by read ends had their occurrences lowered enough to land in the leftmost peak. Contamination of this kind is even more prominent in the case of Nanopore/PacBio reads, as can be seen on the figure 3.6. This fact, along with the practical computation of optimal k-value makes Illumina reads a requirement.

It is necessary to keep the rate of contamination as low as possible early on, as applying the SDKs on reads with higher error rate will increase it further by turning discriminative k-mers into contaminants through spurious occurrences caused by sequencing errors. For instance, while the exported SDK set for EIL30 has less than 0.1% of contaminating k-mers, the same set yields 5.6% contamination in the context of ENP75 dataset.

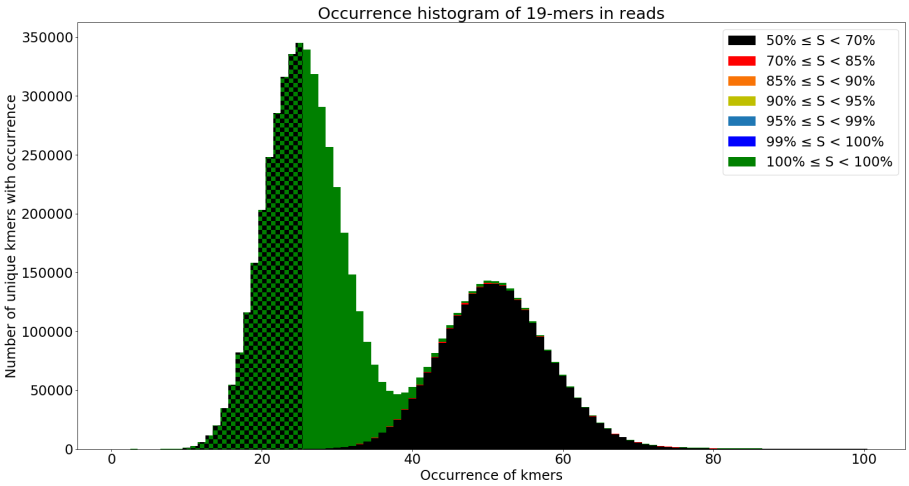

Figure 3.5: *k*-mer histogram of EIL30 dataset with k=19 and exported SDKs highlighted by checkerboard

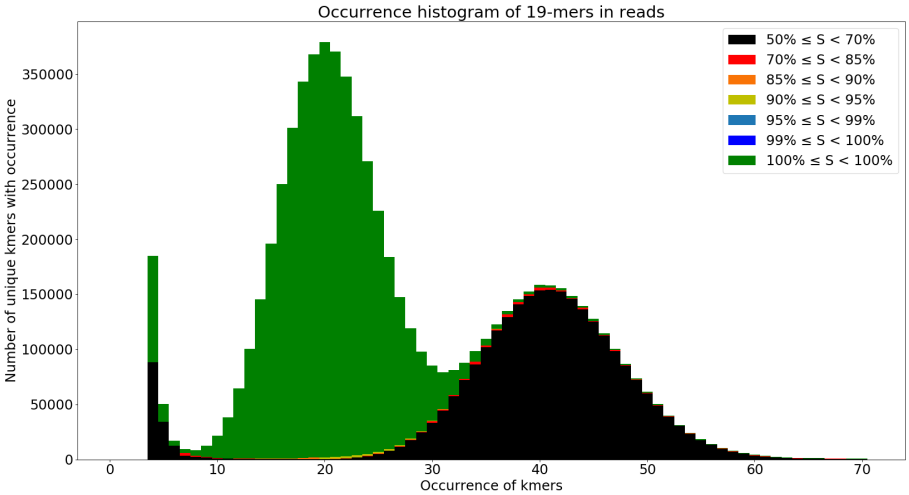

Figure 3.6: *k*-mer histogram of ENP75 dataset. There is noticeable contamination in the left peak.

Chapter 4

Read categorization

Assuming that the majority of obtained SDKs are in fact discriminative, we can postulate a hypothesis : the more SDKs two reads share, the more likely they are to originate from the same haplotype.

Using the number of shared SDKs as a metric, we can restate our categorization problem as a graph problem. We can treat the reads as vertices and the number of shared SDKs between reads as weights of edges between the vertices. From here on out, we will be using the terms “read” and “vertex” interchangeably. We will refer to edges connecting reads or components of the same haplotype as "correct" and to those connecting different haplotype reads as "incorrect".

Our goal then is to identify components in the graph, such that their size is maximized, while also maximizing their “purity” - that is, keeping the proportion of reads from one haplotype in the component as high as possible.

We cannot expect to always find only two large components - that can only be done for a single chromosome diploid organism with high enough density of SDKs. The best we can hope for is finding $2N$ large components for a diploid organism with N chromosomes. In case of our ENP75 and EPB75 datasets $N = 1$, since E.coli only has one chromosome.

We also do not hope to categorize all of the reads - inevitably, there will be many reads that are too short, contain too many errors, or do not carry enough information in their SDKs to be unambiguously categorized. However, as long as the reads in both determined categories span all the regions in the reference haplotype with sufficient coverage, eventual assembly of the reads should still yield high quality contigs.

Small inconsistency in categorized reads is also acceptable using the same reasoning. As long as we end up with significantly more reads of the correct origin covering the same region compared to the incorrect ones, we can expect the assembler to deal with such inconsistencies in the same way it deals with sequencing errors - by creating a consensus sequence.

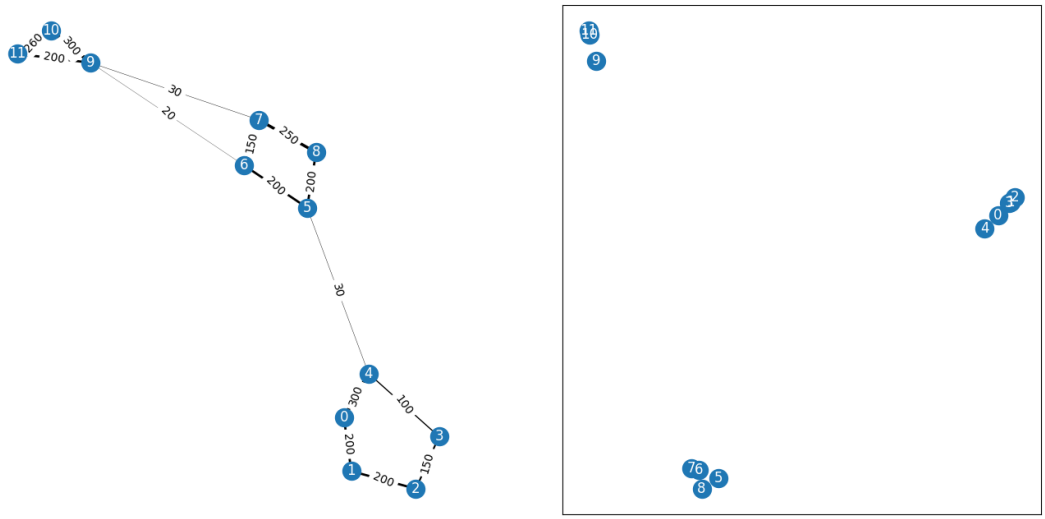

Figure 4.1: Example of spectral embedding. On left side force-directed layout, on right side embedding into 2D space

4.1 Spectral clustering

Assuming that vertices connected by edges with strong weights should belong into the same component, we can approach the problem of creating said components through clustering.

While the sample graph on the left side of figure 4.1 is suggestively drawn in 2 dimensions with vertices already pre-arranged, the graphs we are really dealing with have no such form.

In order for ordinary clustering algorithms (such as K-means) to work, we first have to project our graph into a Euclidian space in such a way that the vertices we expect to be clustered together are separated only by a small distance, while those not clustered together are comparatively far apart.

One such way is spectral embedding.[16] Suppose we have N objects with a defined similarity metric. Given a $N \times N$ similarity matrix A, a spectrum of a graph is computed as eigenvalues and eigenvector of Laplacian of A. Spectral embedding enables us to use the columns of the first d eigenvectors (when sorted by their respective eigenvalues) as coordinates in a d-dimensional coordinate space. Through this embedding, elements that have a high similarity as specified in A are placed close (in terms of standard euclidian distance) together, while the dissimilar elements are further apart.

An example of such embedding into two dimensions can be seen on the right side of figure 4.1. As can be seen from a visualization, a graph embedded using this process is quite viable as an input to a run-of-the-mill clustering algorithm.

For demonstration purposes, we ran spectral clustering (a combination of embedding and clustering) on the RIL50 dataset (≈6400 reads) with SDK set chosen as

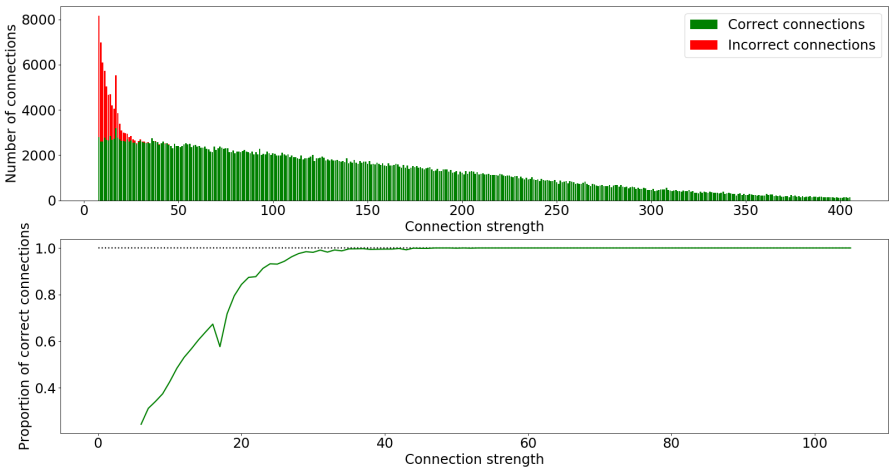

Figure 4.2: On top, histogram of connections in RNP50 color coded by their correctness. On bottom, the relation between number of shared SDKs and a proportion of correct connections.

described in the previous chapter. More specifically, we use Self-Tuning Spectral Clustering[21] embedding into a 10-dimensional space. In order not to overload the graph with a lot of spurious edges that connect unrelated reads, we only considered the connections with strength at least 5. A histogram of these connections can be observed on figure 4.2.

To construct an entry in the similarity matrix for reads r_1 and r_2, we rescale the strength of the connection (r_1, r_2) to a range $[0.3, 20]$ and exponentiate it. This is done as to accentuate the differenced between weakly and strongly connected reads.

$$A[r_1, r_2] = e^{rescale(strength(r_1, r_2))}$$

The results are encouraging : the combination of spectral embedding and clustering yields seven 100% pure components with evenly distributed sizes containing almost all of the reads.

However, applying the same pipeline to a more realistic dataset, such as that of E.coli strains is not feasible due to computational complexity. Calculating the eigenvectors of an $N \times N$ matrix takes in general $O(N^3)$ time, but even if that could be reduced, the $O(N^2)$ memory required for the graph Laplacian matrix is enough of a disqualifying factor for datasets of hundreds of thousands of reads.

If we are to use a method such as spectral embedding, a drastic reduction in the number of treated objects is necessary. As such, we first apply a different heuristic resulting in high-purity components the quantity of which is several orders of magnitude lower.

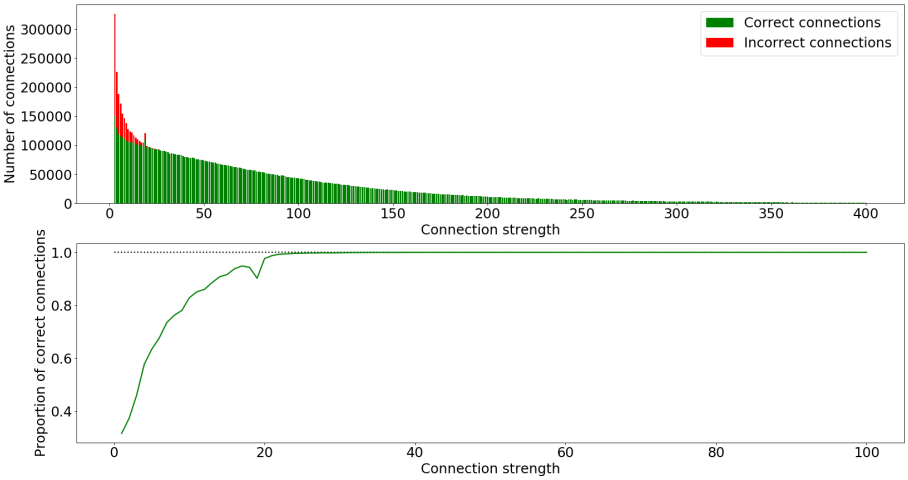

Figure 4.3: Same type of histogram as 4.2, but for ENP75

4.2 Forming scaffold components

If we take a closer look at the figure 4.3, we can see that beyond a certain threshold value for the number of shared SDKs, all of the implied edges in our graph are correct.

The exact value of this threshold seems to depend on a few factors, such as:

- The length distribution of the reads - shorter reads contain less SDKs, so the threshold is also lower
- Error profile of the used sequencing platform - errors eliminate SDKs from the reads, lowering the threshold
- Fraction of k-mers in SDK set that are not true DK - higher contamination rate implies higher threshold
- Abundance of SDKs - the higher percentage of reads in k-mers are SDKs, the higher the threshold, since along with true DK we also grow the set of contaminant SDKs

In all of our experiments with ENP75 and EPB75 datasets, this validity threshold however never exceeded 70 shared k-mers and there is little reason to suspect that the value should be dramatically higher for real datasets - after all, the purpose of the simulation software is to model the first two points of the above list as closely to reality as possible.

Knowing there is a safe threshold, we can guess its upper bound, or rather what percentage of all edges sorted by their strengths in descending order we can consider to be "correct". We set our guess to be the top 15% of all the edges (for ENP75 and EPB75 datasets the actual safe threshold would be met even with top 50% of edges).

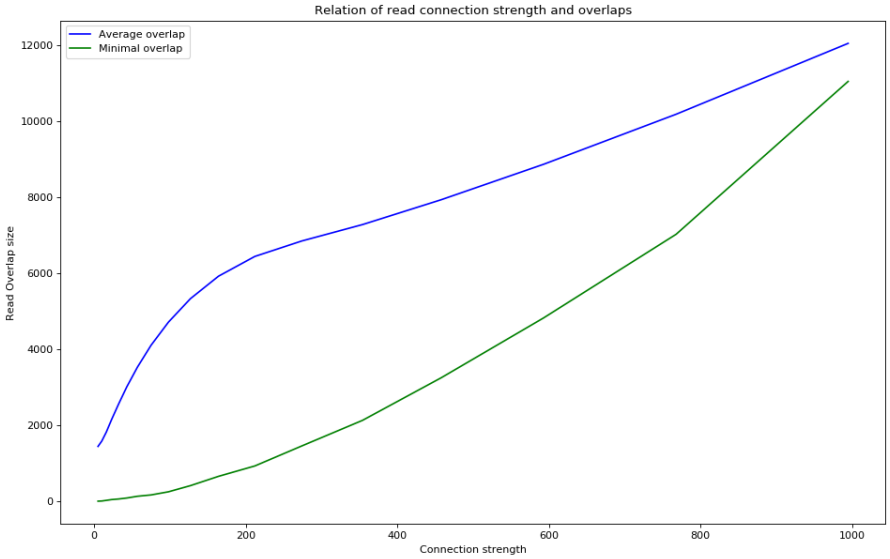

Figure 4.4: Relation between number of shared SDKs between reads and the size of their overlap (as determined by read metadata)

Assuming that all of our guessed edges are correct, we can use these to connect their respective vertices into components and expect these components to be 100% pure at the end. The fastest way to realize these connections is using a union-find algorithm, with the edges sorted by their weights in descending order on the input.

On the output of our union-find we receive a list of formed components, along with a spanning tree of edges used to form them. We call the components created by this first round of union-find scaffold components.

At this point, it is important to realize just what the strong connections represent: reads that share a high number of suspected discriminative k-mers not only come from the same haplotype, but they also cover intersecting intervals of bases in the haplotype genome.

Therefore, by finding strong connections between reads we in a way also find local alignments between reads. It closely follows that by performing union-find on a set of reads we are also creating a spanning tree of read alignments.

Moreover, for adjacent reads in the spanning tree that have overlapping intervals (which in the case of our trees are all the adjacencies), we can construct a larger interval by overlapping them. By induction, it would follow that every path in the tree corresponds to one interval in the haplotype genome.

We visualize one such tree on figure 4.5 using a force-directed layout, where vertices repel each other but are held together by edges.

The vertices in figure 4.5 are color coded according to the value of the middle of their interval. The larger the value, the farther right the vertex is on a green-to-red

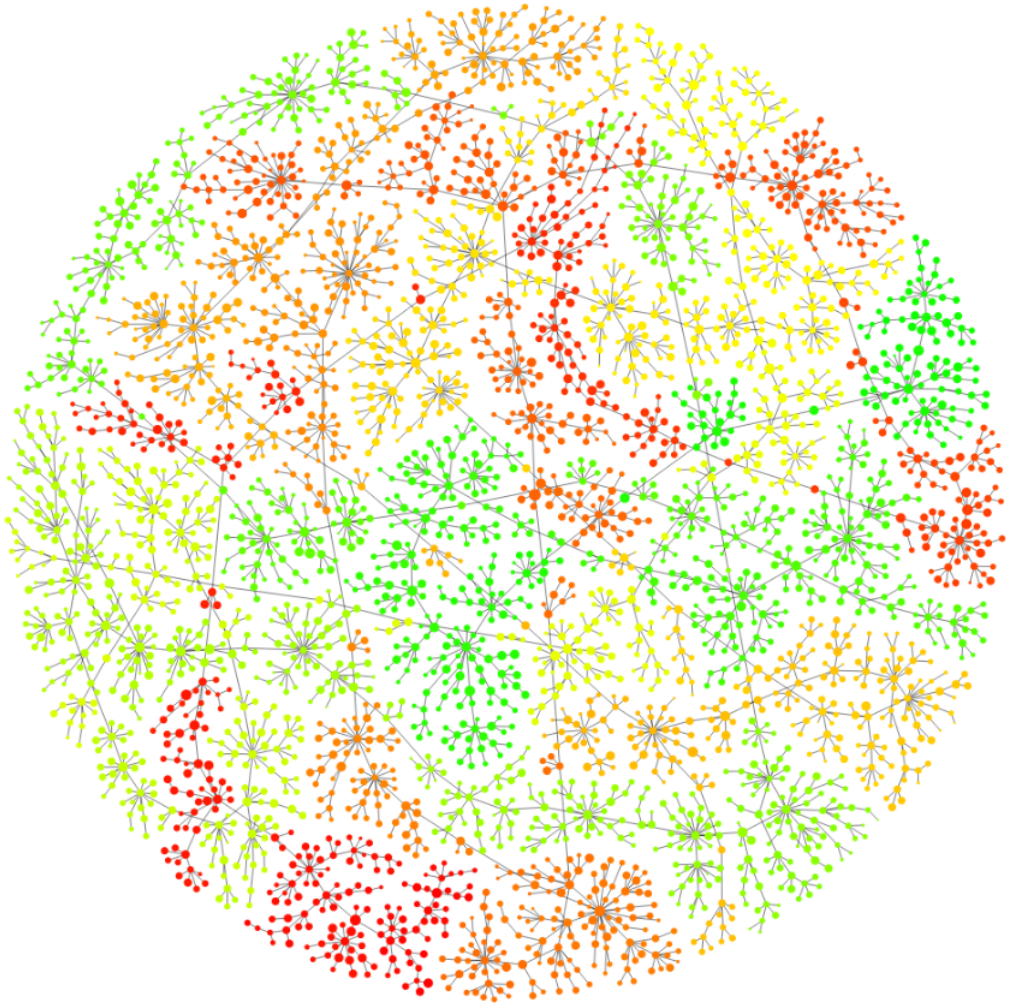

Figure 4.5: Scaffold component rendered using its spanning tree, with edges connecting reads

spectrum, and the larger the dot, the longer the visualized reads are. As can be seen, the only instances when the two neighboring vertices have a noticeably different color happen between pairs, where at least one vertex is relatively big. This can be expected, as in case of Nanopore reads, the read lengths often exceed tens of thousands of bases.

For the ENP75 dataset the described union-find step with connection strength threshold manually set to the lowest possible safe value (51 shared SDKs) managed to create two scaffold components containing reads, the intervals of which covered the vast majority of their respective haplotypes.

With our conservatively chosen 15% of top connections however, the number of found scaffold components rises, but not dramatically - at worst we obtained a couple dozen components. We expect to get no more than a few hundred components even for very large datasets with larger than bacteria genomes. Note that our union-find procedure connects together only a fraction of all the reads in our dataset - in the case of ENP75, it was about two thirds of all reads. We will deal with the remaininder at a later point.

4.3 Component intervals

Let us define a function *overlap_intervals* operating on the interval data provided by simulated reads:

```
type Position = int
type IsStart = bool
Interval = namedtuple(start=int, end=int)

def overlap_intervals(intervals: List[Interval]) -> List[Interval]:
    interval_endpoints: List[Tuple[Position, IsStart]]
    for interval in intervals:
        interval_endpoints.append((read.start, True))
        interval_endpoints.append((read.start, False))
    interval_endpoints.sort()

    overlapped_intervals = []
    opened_start = -1
    opened_intervals_count = 0
    for position, is_start in interval_endpoints:
        if is_start:
            opened_intervals_count += 1
            if opened_intervals_count == 1:
                opened_start = position
        else:
            opened_intervals_count -= 1
            if opened_intervals_count == 0:
                overlapped_intervals.append(Interval(opened_start, position)
    return overlapped_intervals
```

Figure 4.6: Method that overlaps intervals, forming larger interval(s)

Function *overlap_intervals* creates the largest intervals it can by overlapping the intervals of reads contained within a component.

Let us use the function Intervals on the calculated scaffold components of ENP75. The output for each haplotypes components can be seen in tables 4.1 and 4.2 and is sorted by the start of the interval(s).

Note : the intervals wrap around because E.coli has a circular genome

As we can see, for each component, only one interval was created. This is understandable, since all the adjacent reads in the component overlap with their intervals. What is crucial however, is that the intervals between scaffold components of the same category overlap as well. Interestingly, they only seem to overlap by the tails of their intervals : this can be explained by a following thought experiment:

The overlapping tails T_1 and T_2 of intervals correspond to two sets of reads R_1 and R_2 that share a (potentially empty) set of SDKs S. Suppose we grow the overlap, and

Component size	Interval start	Interval end	Overlap with next
4204	462832	1154974	11380
10295	1143594	2736534	11691
3736	2724843	3341829	4680
495	3337149	3452186	6647
3050	3445539	3945210	15198
1531	3930012	4181787	19827
1060	4161960	4357072	9031
4867	4348041	465672	2840

Table 4.1: Component sizes, intervals, and interval overlaps for MG1655 reads

Component size	Interval start	Interval end	Overlap with next
141	194306	235740	12817
1676	222923	471448	10355
4766	461093	1204808	12363
3799	1192445	1781958	9452
297	1772506	1837607	10216
6415	1827391	2794482	9736
506	2784746	2882149	13047
4605	2869102	3562204	3918
461	3558286	3648611	13086
58	3635525	3666623	6569
273	3660054	3731855	11106
3045	3720749	4207882	7347
741	4200535	4327655	8092
6487	4319563	200444	6138

Table 4.2: Component sizes, intervals, and interval overlaps for UTI89 reads

therefore both the tails, along with the sets R_1, R_2 and as a consequence grow the set S. The more we grow the set S however, the more likely it is that there exists a read R_{common} which is sufficiently long and with sufficiently many SDKs contained in S that would bridge the two tails during the union-find phase, contradicting the creation of these tails in the first place. Therefore, the only overlaps we find are relatively short - in our case, their length is only a few average read lengths high.

From the above findings it follows that if we had a way of locating the overlaps, we could merge the scaffold components into even larger components, potentially ending up with only one very large component per each haplotype. Moreover, it can be seen that if we were to construct an interval by using these overlaps, the result would cover the entirety of the haplotype sequence.

At this point it would be tempting to use the same method of calculating connections through shared SDKs as before, but this time between the scaffold components. This however does not work : as can be seen, the component intervals only overlap by a small percentage of their lengths. This means that only the shared SDKs between reads that constitute these overlapping "tails" of components can be trusted to create a correct connection. Since we do not know which these reads are within a component, we would need to consider all the SDKs of a component when calculating a connection. This however results in the contaminating SDKs often outweighing the "trustworthy" SDKs, causing the strongest connections to be incorrect.

4.4 Connecting scaffold components

4.4.1 Longest path in a spanning tree

Recall the analogy of a path in the spanning tree of the scaffold component that represents one interval in the haplotype. If we could find the path representing the longest interval, we would also find the reads that form the aforementioned tails of said interval. On real data, we of course do not know the interval for any read - we only have an approximation (due to insertions and deletions) of its length. We also do now know how long the overlaps between the connected reads are. This however, we can approximate using the SDKs.

Let us define a function *estimate_overlap* operating on a pair of reads. We assume a pre-computed nested hash-map *kmer_positions* containing the positions of SDKs within reads is available.

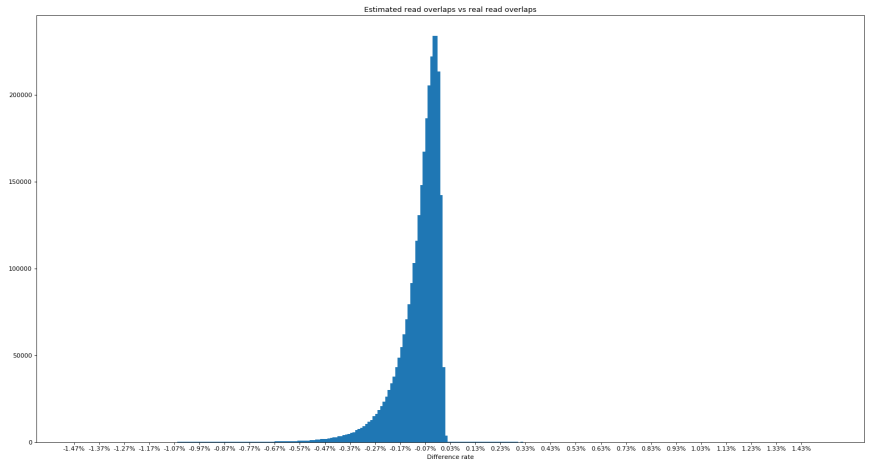

Figure 4.8: Error distribution of calculated overlap lengths vs. real overlap lengths

```
def estimate_overlap(Read x, Read y) -> int:
    shared_kmers = intersection(x.discriminative, y.discriminative)

    x_positions, y_positions = [], []
    for kmer in shared_kmers:
        x_positions.append(kmer_positions[x][kmer])
        y_positions.append(kmer_positions[y][kmer])

    max_x = max(x_positions)
    min_y = min(x_positions)
    max_y = max(y_positions)
    min_y = min(y_positions)

    return max(max_x - min_y, max_y - min_y)
```

Figure 4.7: Method for calculating approximate overlap between reads

Function *estimate_overlap* calculates for a pair of reads the approximate length of their overlap based on the positions of k-mers that the two reads share. Assuming that most k-mers in the SDK set are really discriminative, this approximation will in an overwhelming proportion of cases undervalue the true length of overlap, as can be seen on figure 4.8 depicting the distribution of error for ENP75. This undervaluation however, is not significant (less than half a percent in most cases).

Knowing the approximate overlap size between reads along with their lengths, we define a length metric on the spanning tree of reads as follows:

Let $P = (v_1, v_2, ...v_k)$ be a path in the spanning tree of reads. Let $read\_length(v_i)$

be a function outputting the length of a read represented by vertex v_i. We define a length function on paths to be:

$$path\_length(P) = \sum_{i=1}^{k} read\_length(v_i) - \sum_{i=1}^{k-1} estimate\_overlap(v_i, v_{i+1})$$

We are using the length function *path_length* as a substitute for the unavailable read intervals. As will become apparent later, the approximation provided by this function is quite accurate.

Now that we have a length metric for the paths in the spanning tree which is usable for real data, we can proceed with finding the longest path, and eventually the vertices on the tails. Fortunately, finding a longest path in an acyclic undirected graph such as ours is relatively easy. We will use an adaptation of an algorithm ordinarily used for finding the diameter of a tree.[8] Let us define a function *distance_bfs* operating on an adjacency list constructed from a spanning tree.

```
type Vertex = int
type Distance = int

def distance_bfs(start: Vertex, adjacency: dict):
    vertex_queue = Queue()
    vertex_queue.push(starting)
    visited = set()
    dist = {start: read_length[start]}

    while (!vertex_queue.empty()):
        c = vertex_queue.pop()
        visited.add(c)
        if c not in adjacency:
            continue

        for n, distance in adjacency[c].items():
            if n in visited:
                continue
            dist[n] = dist[c] + read_length[n] - estimate_overlap(n, c)
            vertex_queue.push(n)

    return distances
```

Figure 4.9: Method for calculating distances from one vertex in the spanning tree to all the other

v_{left} **start**	v_{right} **end**	**component int.**	$Tail_{left}$ **start**	$Tail_{right}$ **end**
469260	1154083	462832-1154974	462832	1154974
4172298	4352140	4161960-4357072	4161960	4357072
3453190	3945210	3445539-3945210	3445539	3945210
3349388	3452186	3337149-**3452186**	**3339053**	3452186
2725664	3341829	2724843-3341829	2724843	3341829
3930012	4176215	3930012-4181787	3930012	4181787
1144647	2731271	1143594-2736534	1143594	2736534

Table 4.3: Intervals of scaffold components compared to computed boundary read and tails intervals

Function $distance\_bfs$ calculates distances from a starting vertex to all other vertices using the length function $path\_length$ as distance-increasing metric.

In the first run we start from an arbitrary vertex v_{start} of our tree T. We find a vertex $v_{farthest}$ that is the most distant from v_{start}. It can be proved by contradiction that $v_{farthest}$ is one of the boundary vertices on the longest path.

Starting from $v_{farthest}$, we run $distance\_bfs$ again, obtaining a map of distances D_{left}. Again, we query the vertex from D_{left} with the maximum distance. We will call this vertex v_{left}.

Finally, we run $distance\_bfs$ a third time from v_{left}, obtaining a map of distances D_{right}. The vertex with maximum distance in this map is the same one as $v_{farthest}$. We relabel it as v_{right}.

The path that can be traced from v_{left} to v_{right} is the longest path in the tree T. In the first three columns of table 4.3 we can observe how closely related are intervals for the found boundary reads v_{left}, v_{right} with the overarching interval of the entire tree (scaffold component). For convenience, we occasionally swapped values of v_{left} and v_{right}, as their order is not important for our calculations.

We can see that while not perfectly accurate, the boundary reads v_{left}, v_{right} cover intervals the ends of which are very close to the actual ends of the interval for the whole component. The inaccuracy is caused by the estimative nature of both read lengths and overlaps.

We can however do better. If one recalls, the point of our interest are reads that constitute the tails of the component interval - and we do not necessarily have to only take the two edge-most ones. Instead of only taking the two reads v_{left}, v_{right} that are the farthest apart, we can instead take a set of reads $Tail_{left}, Tail_{right}$, where $Tail_{left}$ is a set that is "sufficiently distant" from v_{right} and $Tail_{right}$ is a set "sufficiently distant" from v_{left}.

In our case, we define "sufficiently distant" as being no more than two average read

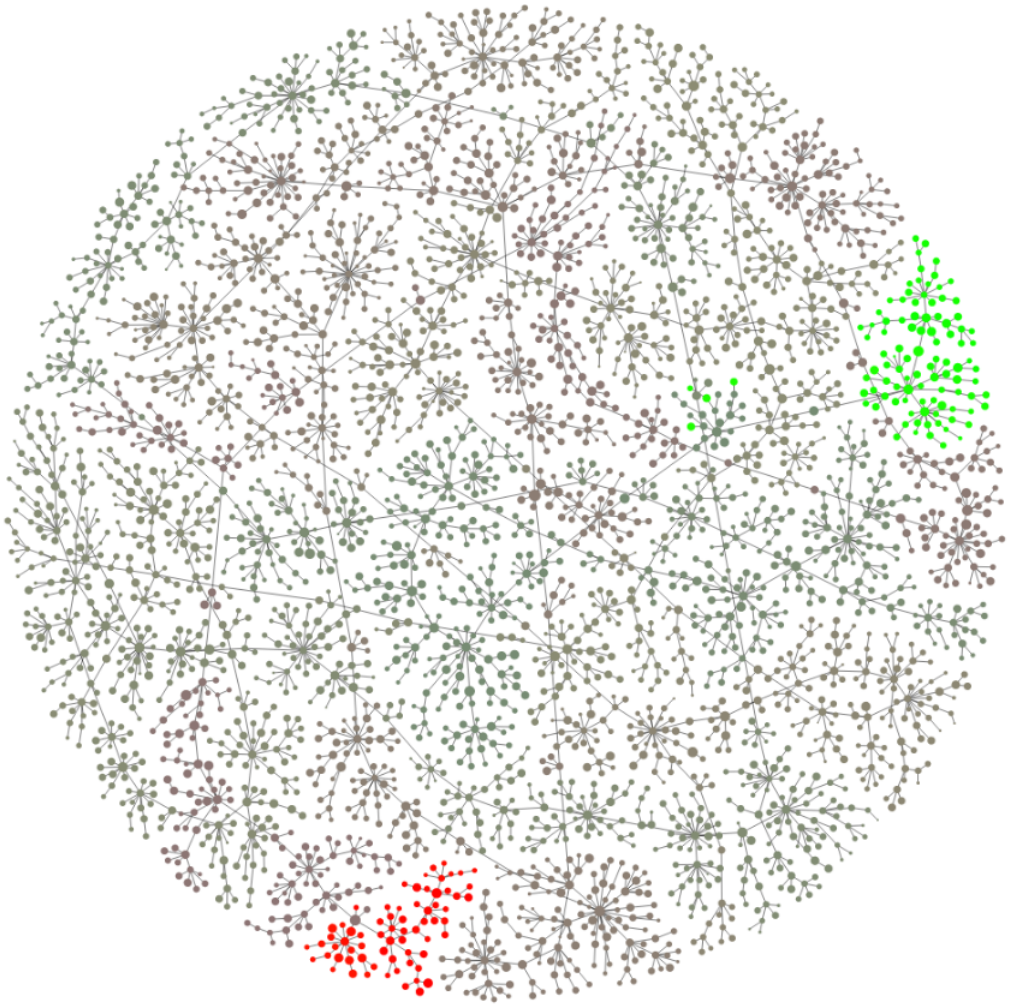

Figure 4.10: Highlighted sets of vertices $Tail_{left}$ and $Tail_{right}$

lengths less distant, than the most distant vertex. This variable is set as such based on the observed overlap length between scaffold component intervals from tables 4.1 and 4.2 which does not seem to depend on the overall interval length.

As can be seen per last three columns of table 4.3, the bounds of intervals for tails strike the component interval bounds dead on with only one exception. To give a visual clue as to which vertices are included in the tail sets, we present figure 4.12, an altered version of figure 4.5. In the figure, the left tail vertices are highlighted in green, while the right tail vertices are red.

Now that we have located the tail reads/vertices of our component spanning trees, we can proceed with calulating connections between them. It's only logical that we again use the number of shared SDKs as the metric, but this time the SDKs for a tail are an aggregate of the SDKs for the reads contained within said tails. In order to ensure our tails contain as many SDKs from the tail interval as possible, we perform an additional step of their "amplification". The idea is, that since we have many reads that went unused during the formation of scaffold components, we can link some of

these to our tails to ensure we capture as many SDKs belonging to the tails intervals as possible. As we will see later, vertices that we temporarily "borrow" this way will end up in the same component eventually.

As such, during amplification we calculate connections from the tail vertices into the set of unused vertices, and incorporate the SDKs of strongly connected (a set constant for number of shared SDKs) into the set of SDKs for the tail.

```
def amplified_tail_SDKs(tail: List[Vertex]) -> Set[Kmer]:
    amplified_tail = tail[::]
    for connection in get_connections(from=tail, minimal_score=40):
        amplified_tail.append(connection.to)

    amplified_tail_SDKs = {}
    for vertex in amplified_tail:
        amplified_tail_SDKs.add(vertex.discriminative_kmers)

    return amplified_tail_SDKs
```

Figure 4.11: Method for amplifying a tail of a spanning tree

We can now proceed with calculation of connections between scaffold components. For each pair of components C_1 and C_2, we compute the number of shared SDKs between the:

- Right tail of C_1 and right tail of C_2
- Right tail of C_1 and left tail of C_2
- Left tail of C_1 and right tail of C_2
- Left tail of C_1 and left tail of C_2

We take only the maximum of the four computed values and set it to be the connection strength between C_1 and C_2.

For the ENP75 dataset which yielded 22 scaffold components in the union-find step, we end up with 151 cross-tail connections with strength higher than zero. A graph of this size is small enough to reintroduce spectral clustering as a viable clustering method.

Using Self-Tuning spectral clustering,[4] we managed to reduce the number of components down to 4, with two components per each haplotype.

We call the components on the output of spectral clustering Core components. From this point on, we will not merge any two Core components together.

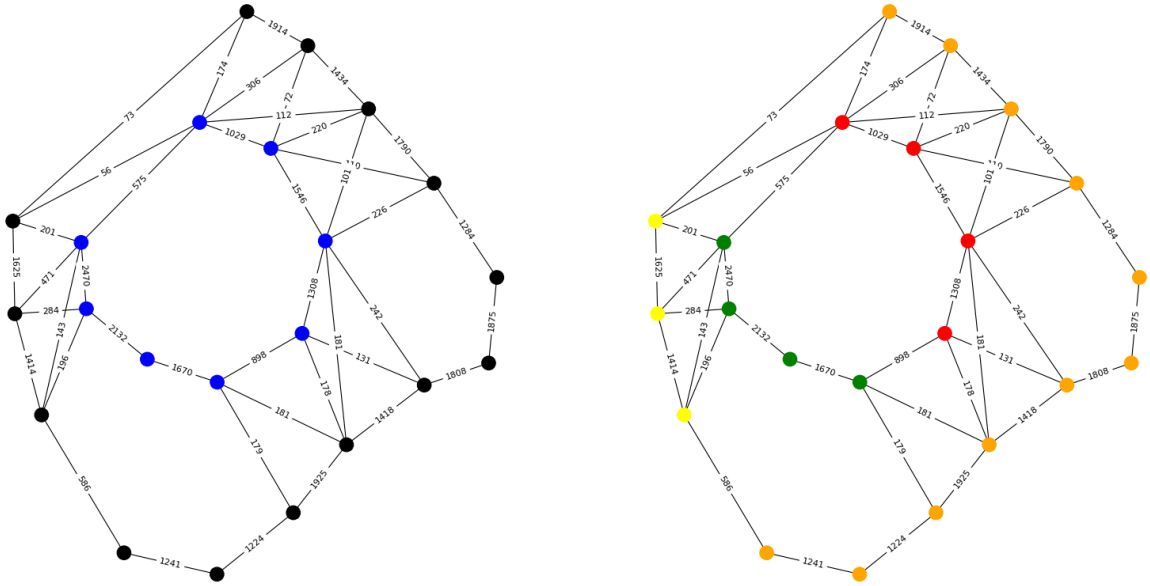

Figure 4.12: Graph of scaffold components and tail connections between them. Correct categorization on the left, clusters found by spectral clustering on the right. Edges with strengths lower than 50 not displayed for brevity

4.5 Enrichment of core components

Now that we have core components which are few in number and together likely span the entirety of all haplotypes, we can direct our focus on resolving the status of all the reads that were left out during the formation of scaffold components.

The process of enrichment is similar to the union-find step we already used, except this time, we calculate only the connections between core components and the unresolved reads, and the union-find is restricted from ever joining together core components. Thanks to this fact, upon calculation of connections we can already tell to which core component a read is going to be assigned, allowing us to plot a figure of realized connections, as shown on figure 4.13.

The practical effect is that most unresolved reads are assigned to the core component to which they have the strongest connection. Again, we set a limit to the minimal strength of a connection to be used during the union find. We found that for the E.coli dataset, going to as low as 20 shared SDKs is viable without introducing more than one percent of impurity to our core components.

The difference in coverage caused by enrichment is depicted on figures 4.14 and figure 4.15.

The enriched components constitute the final result of our algorithm.

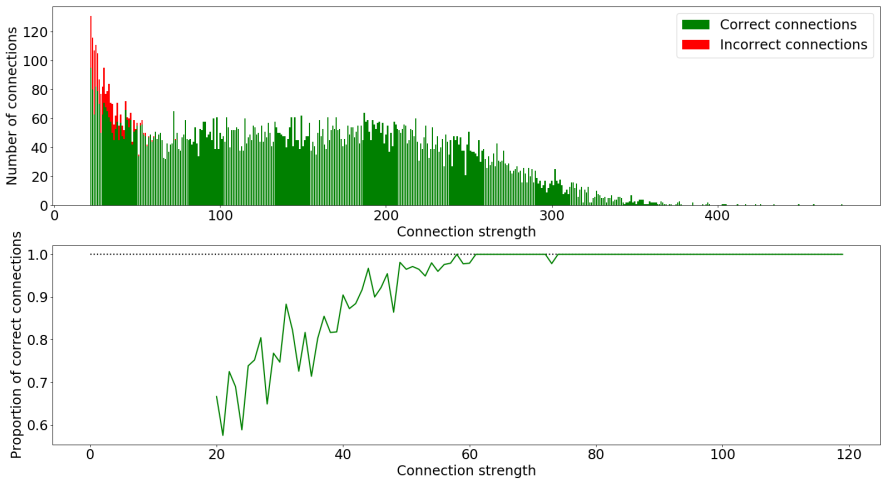

Figure 4.13: On top, histogram of connections in ENP75 color coded by their correctness. On bottom, the relation between number of shared SDKs and a proportion of correct connections.

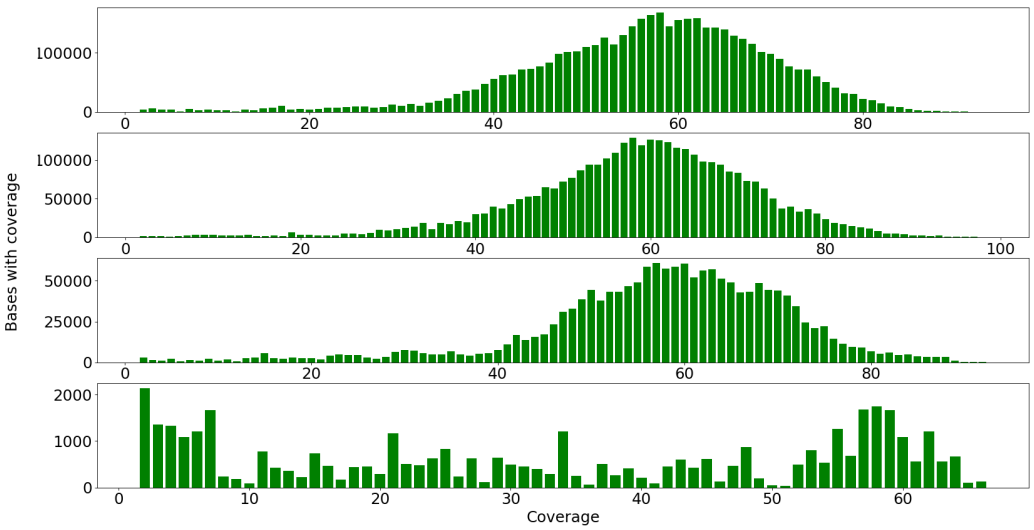

Figure 4.14: Histogram of coverage within components before enrichment

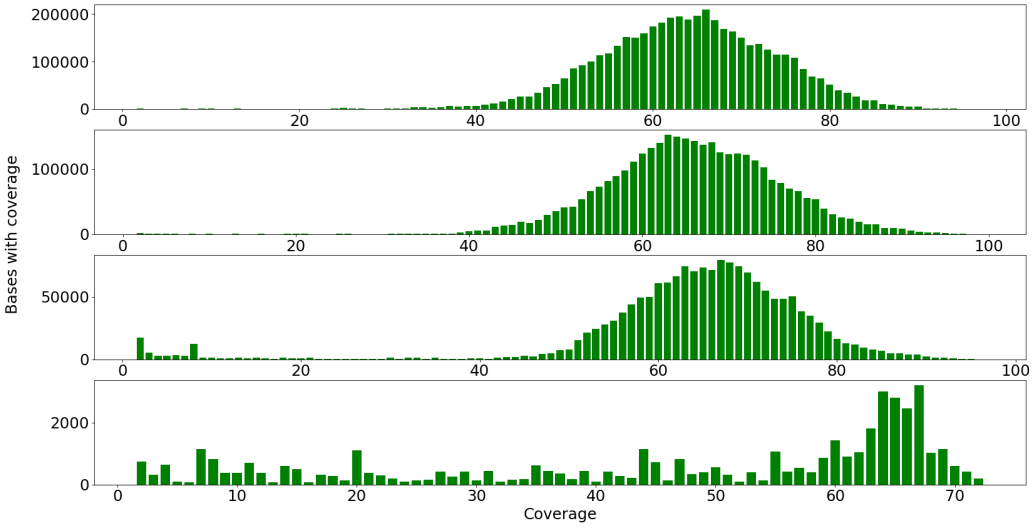

Figure 4.15: Histogram of coverage within components after enrichment. We can see that enrichment visibly improved the overal coverage in the last component

Chapter 5

Evaluation

For ENP75 dataset, our algorithm outputs as little as 4 components of the characteristics specified in table 5.1.

We can see that for the largest components, the purity is maintained above 98%, and the coverage of the regions spanned by components was enough for long contigs to be formed.

To assemble the sequence, we used a pipeline consisting of

1. Mapping the reads using Minimap2
2. Assembling the mapped reads using Miniasm
3. Polishing of the assembled contigs using Racon

For illustration we present a dot plot in figure 5.1 resulting from megablast[5] alignment of the contigs in the first listed component to both MG1655 and UTI89 reference genome.

The results for aligning all the other contigs were similar - alignment with the organism whose reads dominate in the component yields over 99% identity score every time compared to no more than 97.6% with the organism the reads of which are in minority.

MG1655 reads	**UTI89 reads**	**Assembled contig sizes**
634	34345	2019783, 2228493, 163042, 19549
22958	0	2985850
12714	0	1663641
83	4521	666732

Table 5.1: Final components of the ENP75 dataset

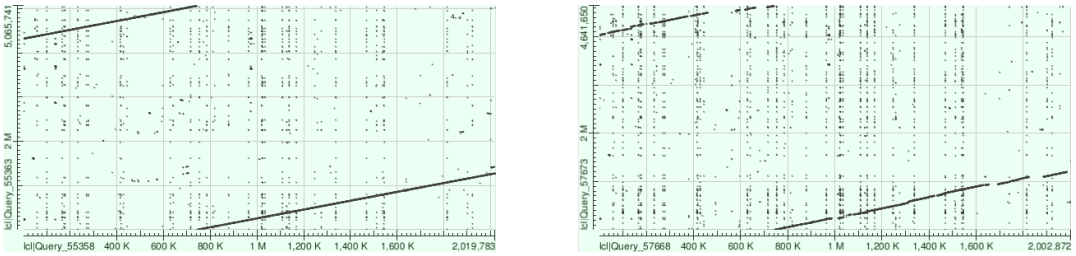

Figure 5.1: On the left alignment to UTI89, on the right alignment to MG1655 reference

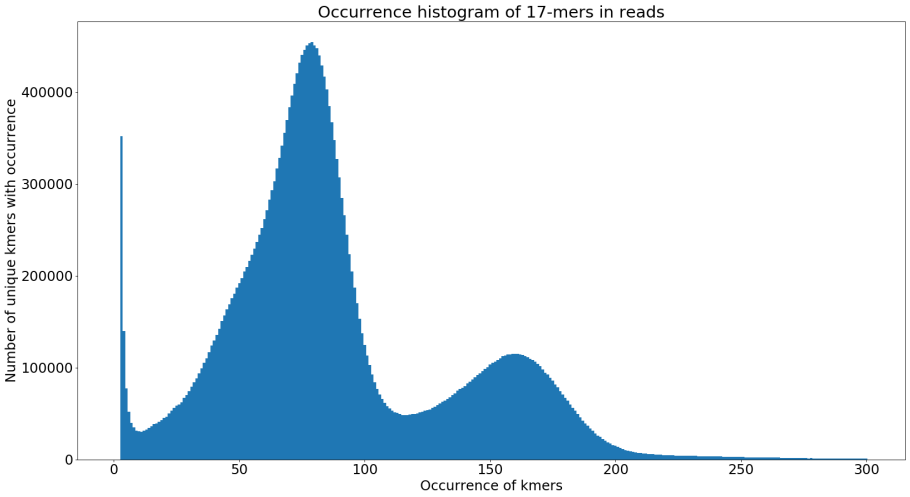

Figure 5.2: k-mer histogram for Magnusiomyces spicifer

5.1 Hybrid genome application

For the sake of simplicity of explanation, we presented the application of our methods only to a case of a diploid organism. Neither the methods nor our implementation however warrant a restriction to two haplotypes. In fact, haplotyping is only a better defined instance of a more general problem of categorization of what is called a "hybrid genome". For example, some yeast strains undergo hybridization events, where different strains exchange fragments of genomes, increasing their genomic complexity.[15]

At our disposal are Illumina, PacBio and Nanopore reads of a novel yeast strain Magnusiomyces spicifer the reference genome of which is as of yet unknown, but we suspect it carries a hybrid genome. Presented on Figure 5.2 is a histogram of k-mers obtained from the Illumina reads.

Two well defined peaks might indicate a hybrid genome. We select kmers occurring between 10 and 78 times, and downsample them to 40%, yielding a little over 5 million SDKs.

Our algorithm however runs into problems during the first union-find phase. Even with manually selected very high threshold value for selection of connections (>500 shared SDKs), our algorithm immediately collapses most of the affected reads into a

single component. After the final enrichment, this single component contains 89% of all the reads in the dataset.

This outcome offers itself to a few possible scenarions:

- The right peak in the histogram 5.2 does not actually represent k-mers that are shared between several copies of chromosomes, but rather k-mers that exist in a section of a genome that repeats itself twice. However, we consider this unlikely
- The union-find step of our algorithm is too sensitive to incorrect connections with high strength. This is possible, since even one unfortunately placed incorrect connection can collapse two large pure components into one that is mixed
- The SDK set is more contaminated than the results from simulated reads would lead us to believe. It is entirely possible that the safe threshold of connection strengths we envisioned when devising our methodology does not exist at all

In conclusion, we think the best course of action in to future is to make our algorithm more resilient to fuzzy information. Revisiting spectral clustering as a method used directly on reads may be viable. As we point out in the next chapter, every read in a dataset with coverage C overlaps with an average of $C - 1$ other reads in case all the SDKs are discriminative. This provides hope that even with mild contamination, the adjacency matrix of a graph of reads could be sparse. Paired with a hardware acceleration, such as provided by CUDA[1] and utilizing related software tools such as cuSPARSE,[2] a practical running time might be achievable even for real datasets.

5.2 Running time

The program was run on an Intel Core i5-8250U CPU with 32GB of RAM. Memory consumption at no point exceeded 6GB. Table 5.2 presents a list of elapsed time for each part of our algorithm.

	ENP75	**EPB75**
Index construction	49225ms	40794ms
All connections	4353ms	2105ms
Merging into scaffold c.	8751ms	5352ms
Tail connecions	7281ms	3107ms
Spectral clustering	183ms	5ms
Merging scaffold c.	1914ms	629ms
Enrichment connections	116ms	136ms
Merging into core c.	2634ms	2781ms

Table 5.2: Listing of run times for each part of our pipeline

Chapter 6

Implementation

6.1 Ensuring fast execution time

6.1.1 K-mer compression

For purposes of tracking occurrences we compress the SDK into a set of 32-bit k-mer IDs. It is reasonable to do so, as more than 4 billion SDKs would exceed the memory limitation of contemporary hardware anyway. In practice, we expect no more than several million of SDKs - for the EIL30 dataset, the exported number of k-mers was little over 2 million. Even if the SDK set cardinality was in the high millions, we recommend downsampling the set to keep the memory consumption of other parts of our algorithm within reasonable limits.

6.1.2 Representation of reads and components

Once we extracted the information about SDKs from a read, its sequence is no longer of any use to us and thus we can keep track of the read by a number identifier (such as its position in the input file)

A component is a structure that contains:

1. Its identifier. Initially, this identifier is identical to the identifier of a read used to create a component
2. List of reads (via their identifiers) contained in the component
3. Sorted list of SDKs (via their IDs) contained in the reads that the component contains

6.1.3 Fast connections via index

From the description of our methods it may not be clear how can a quadratic time complexity be avoided when calculating the connections between reads before running the first union-find. The answer lies in the use of a data structure we call *KmerComponent* index. The structure is a two-dimensional vector - indexing into the top level is done via k-mer IDs mentioned earlier, while the inner vectors contain for an indexed k-mer sorted identifiers of the components in which that k-mer occurs.

In order to calculate all the non-zero strength connections from one component X to all other components $\{Y_1, Y_2 ... Y_n\}$, we can simply iterate through all the inner vectors of k-mers contained in X, and count for each Y_i the number of k-mers it shares with X.

```
def get_connections(from_component: ComponentID):
    shared_kmer_counts : Dict[ComponentID, int]
    for kmer in components[from_component].discriminative:
        for component_id in kmer_component_index[kmer]:
            if component_id in shared_kmer_counts:
                shared_kmer_counts[component_id] += 1
            else:
                shared_kmer_counts[component_id] = 1

    del shared_kmer_counts[from_component]

    result = []
    for component_id, shared_count in shared_kmer_counts.items():
        result.append(from_component, component_id, shared_count)

    return result
```

Figure 6.1: Method for calculation of connections from a component

Runnin *get_connections* from every read/component and accumulating the results yields us all the non-zero connections between reads. In fact, every connection is going to be included twice - but this can easily be countered by having *get_connections* only calculate a connection if the tuple of identifiers for the two connected components is sorted in ascending/descending order.

6.1.4 Fast merging of components

In order to merge the components as dictated by the union-find and later spectral clustering, we need to perform two subtasks:

- Merge the components themselves
- Update the *KmerComponent* index to match the new layout of components

On the input of our merge procedure is a list of K component identifiers that are to be merged together. We choose the first component in the list to be the one absorbing all the others. Appending all of the read identifiers in the absorbed K - 1 components is straightforward, when it comes to identifiers of SDKs however, we need to make sure that the absorption process is both speedy and maintains the sorted order. Luckily, merging of K sorted lists of total size N can be done in $\mathcal{O}(N \cdot \log_2 k)$ as following:

```
def merge_sorted_arrays(arrays: List[List[Element]]):
    heap = PriorityQueue()
    next_position_for_array = [0 for _ in range(len(arrays))]
    for i, arr in enumerate(arrays):
        if next_position[arr_index] < len(arrays[arr_index]):
            heap.push((arr[next_position[i]], i))
            next_position[i] += 1

    previous_element = None
    result = []
    while not heap.empty():
        element, arr_index = heap.pop()

        if element != previous_element:
            result.append(element)
            previous_element = element

        if next_position[arr_index] < len(arrays[arr_index]):
            next_element = arrays[arr_index][next_position[arr_index]]
            heap.push((next_element, arr_index))
            next_position[arr_index] += 1

    return result
```

Figure 6.2: Method for merging of N sorted arrays

During the merging of components, we also accumulate key-value pairs of (k-mer id, list of component ids) that are to be removed from the *KmerComponent* index into a hash-map . Once the merging is complete, we iterate through this map and sort all of the value lists. Since for every k-mer (id) the list of components containing it is

sorted in our index, we can purge all of the entries specified in the hash-map value for this k-mer in linear time as following:

```
def get_filtered_list(original: list, for_removal: list):
    i, j = len(original), len(for_removal)
    filtered = []
    while i < len(original) and j < len(for_removal):
        if for_removal[j] < original[i]:
            j += 1
        elif original[i] < for_removal[j]:
            filtered.append(original[i])
            i += 1
        else:
            i += 1
            j += 1
    return filtered
```

Figure 6.3: Method for filtering out values out of a sorted list

One tid-bit to mention is that we need not insert identifiers of the absorbing component into the index. Since the components doing the absorption become scaffold components, the way we calculate connections is in a directed fashion (calculate from component to component) and the fact that in the future we will always be calculating connections only from the scaffold components, this just isn't necessary.

The part of our work where we determine connections between tails of vertices requires us to have another method for calculating connections. Once again, we make use of the method of merging N sorted lists to obtain aggregate lists X, Y of all the SDKs in the amplified tails T_x, T_y. The number of shared SDKs between the tails is equal to the length of intersection of X and Y.

```
def intersection_size(x: list, y: list):
    x_index, y_index = 0, 0
    Intersection_size = 0
    while x_index < len(x) and y_index < len(y):
        if x[x_index] < y[y_index]:
            x_index += 1
        elif y[y_index] < x[x_index]:
            y_index += 1
        else:
            Intersection_size += 1
            x_index += 1
            y_index += 1

    return result
```

Figure 6.4: Method for calculating the size of intersection of two sorted arrays

6.2 Complexity analysis

Variables used in this section:

- C - coverage of the used reads
- N - number of reads
- R - average read length
- K - number of SDKs
- O - average occurrence count of an SDK in the used reads
- S - number of created scaffold components

By far the two most computationally expensive parts of our algorithm are the construction of the *KmerComponent* index and calculation of connection for the first union-find step.

6.2.1 Construction of *KmerComponent* index

In order to build the *KmerComponent* index, we have to iterate all N reads and save all the occurrences of SDK as well as their positions in the reads. Since our index is made out of vectors, every addition to it is performed in constant time. In the entire run of our algorithm, we only ever remove items from the index, so the memory usage of

both the index, and the hash-map tracking k-mer positions in reads remains $\mathcal{O}(K \cdot O)$ .The dominating factor during index construction however is the number of bases in the input read files we have to iterate through - $\mathcal{O}(N \cdot R)$.

6.2.2 Calculating connections

In order to calculate all connections, we need to run the *get_connections* from all the vertices. Since every read contains on average $\frac{K \cdot O}{N}$ SDKs, and for every one of SDKs we need to iterate its O average occurrences, we end up with $\mathcal{O}(N \cdot \frac{K \cdot O}{N} \cdot O) = \mathcal{O}(K \cdot O^2)$ time complexity of calculating all the connections. As for memory complexity, each read overlaps on average $C - 1$ other reads, meaning that in case of uncontaminated SDK set it also connects with an average of $C - 1$ other reads, resulting in an $\mathcal{O}(N \cdot C)$ memory required to store all the connections. In our experiments however, small SDK contamination did not increase this amount noticeably.

6.2.3 Union-find and merging of components

Union-find yielding the scaffold components requires us to sort the connections in $\mathcal{O}(N \cdot C \cdot \log_2 N \cdot C)$ time, and subsequently perform $\mathcal{O}(N \cdot C)$ join operations in $\mathcal{O}(N \cdot C \cdot \alpha^*(N))$ where α^* is an inverse ackermann function.

If we assume that the union-find includes all N vertices in its join operations, the component merging step requires us to perform N merging operations on lists of average size $\frac{K/cdotO}{N}$. We already showed that merging together M components can be done in $\mathcal{O}(M \cdot \frac{K \cdot O}{N})$, and as such merging of N components will take $\mathcal{O}(K \cdot O)$.

Subsequent update of the *KmerComponent* index requires us to purge at worst O items from every one of K vectors of *KmerComponent* index. As we demonstrated, this can be done in $\mathcal{O}(K \cdot O)$.

6.2.4 Finding tails in scaffold components

For each component of size M, we run BFS 3 times and then iterate through the M computed distances to find tail vertices. This has both time and memory complexity of $\mathcal{O}(M)$ per component, totalling $\mathcal{O}(N)$O for all components combined.

To amplify the tails, we need to run *get_connections* from the tail vertices, of which there is a constant number for each of S components. In total, calculating connections takes $\mathcal{O}(S \cdot \frac{K \cdot O}{N} \cdot O)$ operations. Since every read overlaps with an average of $C - 1$ other reads, the total number of vertices in tails of all components amounts to $\mathcal{O}(S \cdot C)$ items.

To obtain SDKs from the amplified tails, we need to merge $\mathcal{O}(S \cdot C)$ lists of SDKs of average length $K \cdot \frac{O}{N}$, therefore requiring $\mathcal{O}(S \cdot C \cdot K \cdot \frac{O}{N})$ operations.

6.2.5 Spectral clustering

Lastly, to calculate connections between tails requires $\mathcal{O}(S^2 \cdot C \cdot K \cdot \frac{O}{N})$ operations, as for pair of tails we compute the intersection of two $\mathcal{O}(C \cdot K \cdot \frac{O}{N})$ sized arrays.

The input to the spectral clustering is a $\mathcal{O}(S^2)$ sized adjacency matrix of connections, and the computation of eigenvalues (which takes $\mathcal{O}(S^3)$ time) does not exceed this memory complexity. Subsequent merging of scaffold components takes at most $\mathcal{O}(S \cdot K)$ operations.

6.2.6 Core component enrichment

To compute connections from M core components, we perform at most $\mathcal{O}(M \cdot K\dot{S})$ operations (at worst every core component contains all of the SDKs) yielding at most $\mathcal{O}(M\dot{N})$ connections. Subsequent merging of components takes at worst $\mathcal{O}(M \cdot K)$ time.

Chapter 7

Conclusion

In this work we present a pipeline for categorization of mixed haplotype sequencing data by utilizing high-quality short-reads as means to extract haplotype-specific information usable as a guide for categorization of long reads. We have successfully demonstrated that for simulated datasets constructed from two E.coli strains in place of haplotype references it is possible to recover large, highly consistent sets of reads. Assembling the reads contained in mentioned sets results in very long contigs with excellent mapping to the reference genome.

Our methods however show limitations when it comes to a real dataset with unknown reference genome. Applying our algorithm to a novel yeast strain did not yield any usable information, as most of the reads were clustered together, demonstrating the fragility of some parts of our pipeline. Future research and testing needs to be done to determine ways of making it more robust against impurities on the input.

Our software is open-source and available on Github
`https://github.com/matuszelenak/Hybrid-Genome-Assembler/`.

Bibliography

[1] Cuda - high performance computing. https://developer.nvidia.com/cuda-zone. Accessed: 18.5.2020.

[2] cusparse, the cuda sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html. Accessed: 18.5.2020.

[3] Dna. https://en.wikipedia.org/wiki/DNA. Accessed: 15.5.2017.

[4] Libclustering. https://github.com/pthimon/clustering. Accessed: 18.5.2020.

[5] Megablast. https://blast.ncbi.nlm.nih.gov/Blast.cgi. Accessed: 18.5.2020.

[6] Nanosim-h. https://pypi.org/project/NanoSim-H/. Accessed: 18.5.2020.

[7] Snps. https://ghr.nlm.nih.gov/primer/genomicresearch/snp. Accessed: 18.5.2020.

[8] Tree diameter. https://www.geeksforgeeks.org/diameter-n-ary-tree-using-bfs/. Accessed: 18.5.2020.

[9] Daniel Aird, Michael G Ross, Wei-Sheng Chen, Maxwell Danielsson, Timothy Fennell, Carsten Russ, David B Jaffe, Chad Nusbaum, and Andreas Gnirke. Analyzing and minimizing pcr amplification bias in illumina sequencing libraries. *Genome biology*, 12(2):R18, 2011.

[10] Clinical Scientist. Population health research and disease management – the frontier, 2015. Accessed: 18.5.2020.

[11] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. 2007.

[12] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

[13] Mehdi Kchouk, Jean-François Gibrat, and Mourad Elloumi. Generations of sequencing technologies: from first to next generation. *Biology and Medicine*, 9(3), 2017.

[14] Sergey Koren, Arang Rhie, Brian P Walenz, Alexander T Dilthey, Derek M Bickhart, Sarah B Kingan, Stefan Hiendleder, John L Williams, Timothy PL Smith, and Adam Phillippy. Complete assembly of parental haplotypes with trio binning. *BioRxiv*, page 271486, 2018.

[15] Ksenija Lopandic. Saccharomyces interspecies hybrids as model organisms for studying yeast adaptation to stressful environments. *Yeast*, 35(1):21–38, 2018.

[16] Bin Luo, Richard C Wilson, and Edwin R Hancock. Spectral embedding of graphs. *Pattern recognition*, 36(10):2213–2230, 2003.

[17] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[18] Robert F Massung. Dna amplification: Current technologies and applications. *Emerging infectious diseases*, 11(2):357, 2005.

[19] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[20] Chen Yang, Justin Chu, René L Warren, and Inanç Birol. Nanosim: nanopore sequence read simulator based on statistical characterization. *GigaScience*, 6(4):gix010, 2017.

[21] Lihi Zelnik-Manor and Pietro Perona. Self-tuning spectral clustering. In *Advances in neural information processing systems*, pages 1601–1608, 2005.

[22] Wenmin Zhang, Ben Jia, and Chaochun Wei. Pass: a sequencing simulator for pacbio sequencing. *BMC bioinformatics*, 20(1):1–7, 2019.