Department of Computer Science
Series of Publications A
Report A-2017-3

Algorithms and Data Structures for Sequence Analysis in the Pan-Genomic Era

Daniel Valenzuela

To be presented with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium CK112, Exactum on June 9th, 2017 at 12 o'clock noon.

University of Helsinki
Finland

Supervisor
Veli Mäkinen, University of Helsinki, Finland

Pre-examiners
Szymon Grabowski, Lodz University of Technology, Poland
Alberto Policriti, University of Udine, Italy

Opponent
Gregory Kucherov, University Paris-Est Marne-la-Vallée, France

Custos
Veli Mäkinen, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/
Telephone: +358 2941 911, telefax: +358 9 876 4314

ISSN 1238-8645
ISBN 978-951-51-3230-7 (paperback)
ISBN 978-951-51-3231-4 (PDF)
Computing Reviews (1998) Classification: E.4, J.3
Helsinki 2017
Unigrafia

Algorithms and Data Structures for Sequence Analysis in the Pan-Genomic Era

Daniel Valenzuela

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
daniel.valenzuela@cs.helsinki.fi

PhD Thesis, Series of Publications A, Report A-2017-3
Helsinki, May 2017, 74+78 pages
ISSN 1238-8645
ISBN 978-951-51-3230-7 (paperback)
ISBN 978-951-51-3231-4 (PDF)

Abstract

The advent of Next-Generation Sequencing brought new challenges for biological sequence analysis: larger volumes of sequencing data, a proliferation of research projects relying on genomic analysis, and the materialization of rich genomic databases, to name a few. A recent example of the latter, gnomeAD, contains more than 15,000 whole human genomes from unrelated individuals. Today a pressing challenge is how to leverage the full potential of such pan-genomic collections.

Among the many biological sequencing processes that rely on computation methods, this thesis is motivated by variation calling and haplotyping. Variation calling is the process of characterizing an individual's genome by identifying how it differs from a reference genome. The standard approach is to first obtain a set of small DNA fragments – called *reads* – from a biological sample. Genetic variants in the individual's genome are detected by analyzing the alignment of these reads to the reference. A related procedure is haplotype phasing. Sexual organisms have their genome organized in two sets of chromosomes, with equivalent functions. Each set is inherited from the mother and the father respectively, and its elements are called *haplotypes*. The haplotype phasing problem is, once genetic variants are discovered, to attribute them to either of the haplotypes.

The first part of this thesis incrementally builds a novel pipeline for variant calling. We propose to replace the single reference model by a pan-genomic

one: a reference that comprises a large set of genomes from the same species.

The first challenge to realize this goal is to efficiently handle large collections of genomes. A useful tool for this task is the family of Lempel-Ziv compression algorithms. We focus on two of its exponents, namely the RLZ and LZ77 algorithms. We analyze the first, and propose some modifications to both. Using them we develop a scalable index that can process collections longer than 2TB.

With this pan-genomic read aligner we propose a novel variation calling pipeline to go from a single reference to thousands of them.We explore our variation calling pipeline on a mutation-rich subsequence of a Finnish population genome. Our approach consistently outperforms the single-reference approach to variation calling.

The second part of this thesis revolves around the haplotype phasing problem. First, we propose a general model for sequence alignment of diploid genomes that represents diploid genomes as a pair-wise alignment. Next we extend this model to offer a solution for the haplotype phasing problem in the family-trio setting (that is, when we know the variants present in an individual and in her parents). Finally, in the context of an existing read-based approach to haplotyping, we go back to basic algorithms: we observe that the aforementioned approach needs to prune a set of reads aligned to a reference. We model this problem as an interval scheduling problem and propose an algorithm that solves the problem in sub-quadratic time. Moreover, we give a 2-approximation algorithm that runs in linearithmic time.

Computing Reviews (1998) Categories and Subject Descriptors:
E.4 Coding and Information Theory: Data compaction and compression
J.3 Life and medical sciences: Biology and genetics

General Terms:
Algorithms, Design, Experimentation

Additional Key Words and Phrases:
Sequence alignment, Pan-genomics

Acknowledgements

I could not have made it this far without the support of many people to whom I am deeply thankful.

First and foremost, to my supervisor, Professor Veli Mäkinen: thank you for trusting me, for sharing your way of doing research, and for the opportunity to work in exciting problems. Thank you for the freedom and for the guidance you gave me throughout this four-year journey. I was also fortunate in my other mentors on this journey: Alexandru I. Tomescu, Travis Gagie and Simon J. Puglisi. Thank you for your time, patience and guidance. I would also like to express my sincere gratitude to the present and past members of the Genome-scale algorithmics for creating a fantastic working environment.

I would also like to thank the institutions that have supported this research: The Centre of Excellence in Cancer Genetics (CoECG), funded by the Academy of Finland, the Doctoral Programme in Computer Science (DoCS), the Department of Computer Science of the University of Helsinki, and the Helsinki Institute for Information and Technology (HIIT).

This manuscript is based on a series of papers, and I am thankful to all of my co-authors; it was a pleasure to work together with you. I would also like to thank my pre-examiners Szymon Grabowski and Alberto Policriti, for reading the entire thesis and providing helpful feedback. Thanks to Iona Italia and Pirjo Moen too, for their help with the manuscript.

It is thanks to Gonzalo Navarro that I ended up in Helsinki: thanks for encouraging me to take this leap, it was totally worth it!

After my M.Sc I was exploring possibilities for the future and I was lucky enough to encounter excellent mentors, whose advices and teaching were lasting and helped me to get through this PhD. For this I owe a special debt of gratitude to Professor Hannah Bast and to Andres Villavicencio.

Finally, but most importantly, I want to thank my friends and loved ones. My gratitude to you may have nothing to do with this thesis, but friendship is the most important thing in life and I am grateful for yours. I particularly like to thank my parents, Alejandro and Tili, for enthusiasti-

cally supporting me in every endeavour. Finally I would like to thank my wife Nadia, for bravely joining me in this great adventure of love. As we build a life together you have made my life richer than ever.

Helsinki, May, 2017
Daniel Valenzuela

Original Papers

This thesis is based on the following research articles, referred as Paper I to Paper VI in the thesis.

I Daniel Valenzuela. **CHICO: A Compressed Hybrid Index for Repetitive Collections**. In *Proc. 15th Symposium on Experimental Algorithms (SEA 2016)*.

II Travis Gagie, Simon J. Puglisi, and Daniel Valenzuela. **Analyzing Relative Lempel-Ziv**. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*.

III Daniel Valenzuela, Niko Välimäki, Esa Pitkänen and Veli Mäkinen. **Pan-Genome Read Alignment to Improve Variation Calling**. Submitted.

IV Veli Mäkinen and Daniel Valenzuela. **Recombination-aware alignment of diploid individuals**. In *BMC Genomics 2014 15(Suppl 6):S15*

V Veli Mäkinen and Daniel Valenzuela. **Diploid Alignments and Haplotyping**. In *Proc. 11th International Symposium on Bioinformatics Research and Applications (ISBRA 2015)*.

VI Veli Mäkinen, Valeria Staneva, Alexandru I. Tomescu, Daniel Valenzuela and Sebastian Wilzbach. **Interval scheduling maximizing minimum coverage**. In *Discrete Applied Mathematics*. Volume 225, 2017.

Contents

Chapter 1

Introduction

The first drafts of the Human Genome were published in 2001 [47, 94]. Obtaining them was a multi-billion dollar enterprise that involved over 20 institutions in 6 countries and hundreds of researchers [18].

In contrast, during the 2010s we have witnessed a growing market for *personalized genomes*: it is possible to send a sample of blood or saliva by mail and get your results on-line, for prices ranging between less than a hundred to the couple of thousand dollars.[1]

Despite the many technological advances since the completion of the Human Genome Project, two factors are crucial to explain such a drop in prices: first, the adoption of the so called Next-Generation Sequencing technologies made a huge impact on decreasing the sequencing costs; second, once the first human genome is obtained, the process to sequence a new individual's genome changes entirely. The first time, the sequence needs to be assembled "from scratch". From then onwards, the first sequence can be used as a *reference* for a much accessible process called variation calling (see Section 1.2).

Today the landscape keeps changing: there are thousands of high-quality genomes available [89] and more are being sequenced [90, 27], establishing a map of genetic diversity in the search for a better understanding of its implications. Similar processes have been ongoing also for non-human organisms [87, 84], to the extent that some authors talk about the dawn of the *pan-genomic era*[19].

All those achievements have been heavily reliant on computational methods [70, 41, 26]. There is a large repertoire of those methods that are critical

[1] The current diversity of prices is due to many factors, including how much of the total genome is actually sequenced. The startup 23andMe famously offered genomes from less than $100USD, mainly focusing on specific sites in the genome. On the other hand, Illumina offers whole-genome sequencing for $\$5,000$USD[92].

for different stages of the genome sequencing processes. As the sequencing technologies are advancing rapidly, the computational methods need to advance as well, evolving and adapting to new challenges. In this thesis I will present different algorithms and data-structures aiming to contribute to some of these processes needed for genome sequencing. In particular, some of the contributions consider the pan-genomic nature of the data we now have available.

The structure of this chapter is as follows: In Sections 1.1 and 1.2 I will introduce some basic concepts from bioinformatics. Then in Section 1.3 I will outline this thesis' contributions. Finally, in Section 1.4 I give a brief summary of the original publications, discussing my personal contribution to each of them.

1.1 DNA and genetic variation

Deoxyribonucleic acid (DNA) constitutes "the blueprints of life", for it is the molecule within the cell that contains the instructions for growing, functioning and reproducing for all known living organisms. A DNA molecule is the linear concatenation of nucleotides, which can be of four different kinds: adenine(A), cytosine(C), guanine(G) and thymine(T). Most of the DNA is packed in units called *chromosomes*. Different species have different numbers of chromosomes with different contents. Within species, different individuals normally have the same number of chromosomes and the content of each chromosome is highly similar throughout the population.

In most cells of sexual organisms, chromosomes are organized in pairs, such that each element of the pair – a "copy" – is inherited from a "father" and a "mother" respectively. Each of those copies are called *homologous chromosomes* or *haplotypes*. The cells that have this arrangement are called *diploid* cells. During reproduction, specialized cells *recombine* the pairs of chromosomes generating *gametes*: cells that have only one copy of each chromosome and that later can combine with a gamete from another individual to create a new individual. These replication and recombination processes are not exact, and sometimes an "error" occurs, generating genetic variations that can be later inherited by the offspring.

Because most of the genetic content is the same among individuals within the same species, it is possible to define a reference genome for a species. Then, we can characterize the *genetic variations* of each individual in terms of how they differ from the reference genome. A variant that occurs in a single nucleotide, where, for instance a specific T in the reference genome is replaced by a G, is called single nucleotide polymorphism (SNP),

or single nucleotides variants (SNV). Those account for a large amount of the known variations in human genomes, however there are also more complex variants such as (large) insertions, deletions, inversions, among others. When a variant is present in only one of the chromosome copies, it is said to be a *heterozygous variant*. A variant that is present in both copies is known as a *homozygous variant*.

1.2 NGS and variation calling

Next-Generation Sequencing (NGS) is an expression that refers to many technologies that have parallelized the sequencing process, producing vast amounts of sequences concurrently. What these have in common is that they cut DNA molecules from the donor into small pieces, known as *reads*, which are what is actually sequenced in great amounts. Reads length can range from about a hundred to a couple of thousands nucleotides. When reads are sequenced there is no information about their original position in the genome, so the output is essentially a large set of small pieces of the genome.[2] The problem of sequencing a genome starting from this set of reads is known as *de novo genome assembly*. This is known to be a hard problem, and its simplest mathematical formulation is NP-Hard [67].

When we already posses a reference genome of the same species as the donor, the enterprise to sequence the donor genome is much more accessible through what is known as *variation calling*. A simplified summary of this process is as follows: Starting from a biological sample from the donor, NGS technology is used to obtain a massive set of reads. A *read aligner* maps the reads to the reference genome, ideally to positions with a high similarity score with the read. The number of reads whose alignment covers a position is known as the *coverage*. As the number of NGS reads is large, the ideal situation is to have high coverage through the genome, so that any position on the reference genome will have many reads piled up. This pile-up is analyzed, typically using statistical methods [59, 36] that can discover variations in the donor genome. With this process SNPs are relatively easy to detect, but more complex variants pose a bigger challenge. Nowadays, variation calling is routinely performed to sequence genomes, using a wide variety of methods [4, 59, 85, 74]. However, their results are not always consistent [74]. This in itself is motivation enough to study possible improvements for variation-calling mechanisms.

[2] There is more information, for instance, each base is annotated with the error probability, and some technologies have *paired-end* reads, where there is a prior knowledge about their expected distance within the genome.

When variation-calling tools report a variant, they will indicate whether it is homozygous or heterozygous. However, this is not enough to completely reconstruct the donors genome: we still do not know to which of the two copies of each chromosome each variant belongs to. The problem of deciding whether two variants belong to the same copy of the chromosome or not is called *haplotype phasing*. There is a wide repertoire of methods to address this problem [13, 14, 10, 76]

1.3 Outline of the contributions

The main motivation of this thesis is found in variation calling and haplotype phasing. Our contributions are at different levels, from analysis of basic algorithms, to the design of novel pipelines for variation calling.

The first part of this thesis incrementally builds a novel framework for variant calling. We propose to replace the single reference model by a pan-genomic one: a reference that comprises a large set of genomes from the same species. The second part revolves around the haplotype phasing problem. In Chapter 2 we first review the basic algorithms that we will use throughout this thesis.

The first challenge in the route to the pan-genomic reference, is to efficiently handle large collection of genomes. A useful tool for this is the family of Lempel-Ziv compression algorithms, which have proven to be extremely effective to compress highly repetitive collections, such as collections of genomes from the same species. In Chapter 3 we introduce two exponents of this algorithms: the LZ77 and the RLZ algorithms. Then, we present the contributions of Paper I: a study of the functioning of the RLZ in a particular setting, namely, the construction of an artificial reference.

Then, in Chapter 4 we propose a slight generalization of the LZ77 algorithm so that we obtain a class that includes the LZ77 and the RLZ algorithms. Utilizing this we develop a scalable index that can process a 2.4TB synthetic collection of DNA in less than 12 hours.

To improve variation calling, in Chapter 5 we submit a novel pipeline that replaces the single linear reference by a pan-genomic one. To index the pan-genome we further develop the index of Chapter 4 to transform it into a read aligner for pan-genomic reference. This chapter is based on the work of Paper III. We tested our read aligner on real data, replacing the standard reference by sequences from the 1000 genomes project. The use of this pan-genomic reference increased the number of mapped reads, while the time required for the mapping remained in the same ballpark. We tested our variation calling pipeline on a mutation-rich subsequence of

a Finnish population genome, and observed that the results are better than a standard variation calling pipeline.

The first contribution of Chapter 6 is a general model for sequence alignment of diploid genomes, which represents diploid genomes as a pairwise alignment. This is based on Paper IV, where we originally proposed this model. Then, we extend this representation to model the haplotype phasing problem in the so called family-trio setting (that is, when we know the variants present in an individual and in her parents). This latter part of Chapter 6 is based on Paper V.

Finally, in the context of an existing read-based approach to haplotyping [76], in Chapter 7 we go back to basic algorithms. We observe that the aforementioned approach needs to prune a set of reads aligned to a reference, and in the original solution the reads were randomly pruned. We studied this pruning problem and its motivation, modeled it as an interval scheduling problem, and propose exact and approximate algorithms to solve it.

Finally, Chapter 8 summarizes the contributions, discussing open problems and possible directions for future research.

1.4 Original papers and individual contributions

Next I give a brief summary of the original papers that are the basis of this thesis, indicating my specific contribution in each of them.

Paper I. In this paper we presented a theoretical and empirical study on the compression achieved by the Relative-Lempel Ziv algorithm in the scenario where it needs to build an artificial reference.

My contributions to this work were in the experimental study: the experimental analysis was jointly designed by the three authors. I implemented and ran the experiments and contributed to the writing of that section.

Paper II. In this paper I presented CHICO, an improved version of the Hybrid Index of Ferrada, Gagie, Hirvola and Puglisi [29]. CHICO reduces the space usage of the original version while keeping similar query times. Compared with other indexes for repetitive collections, the size of the index and query times were competitive, while indexing times were better than all the alternatives. In this paper I also demonstrated the scalability of the approach, which indexed a 2.4TB collection in about 12 hours.

Paper III. In this paper we proposed a novel framework to perform variation calling using pan-genomic references and released an implementation called PanVC. We represent the pan-genome using a multiple sequence alignment, indexing the underlying sequence using CHIC aligner, a specialized version of the index of Paper II. We demonstrated that the approach can improve the effectiveness of variation calling.

The framework was jointly designed with Veli Mäkinen. I implemented PanVC and the CHIC aligner. I ran all the experiments for PanVC and wrote a significant part of the paper.

Paper IV. In this paper we proposed a generalization of edit distance/sequence alignment where the object to be aligned is itself a pairwise alignment. The proposal is to model diploid individuals as pair-wise alignments instead of a single sequence. This captures the possibility of known variants whose correct haplotype phase is not known or that has been wrongly phased.

The original idea is from Veli Mäkinen, the actual model and algorithms were jointly designed. I implemented the algorithms and ran the experiments, and the paper was written jointly.

Paper V. This is a follow-up of Paper IV, where we extended the diploid alignment model to solve the haplotyping problem in family trios, that is, when the genome has been sequenced but not phased for a father, a mother and a child.

The model and algorithm were done jointly. I implemented the algorithms and ran the experiments, and the paper was written jointly.

Paper VI. In this paper we addresses the problem of pruning a set of NGS reads aligned to a reference. This step is a required preprocessing step needed by previous approaches to the haplotyping problem [62, 76]. We modeled it as a job scheduling problem and present different algorithms to solve it.

The exact algorithms were designed together by Veli Mäkinen, Alexandru Tomescu and me. I also contributed with the analysis of the 2-approximation algorithm, and with the supervision of Sebastian Wilzbach, who implemented the exact algorithm and ran the experiments.

Chapter 2

Preliminaries

In this chapter we introduce some basic concepts in computer science that will be used through this thesis.

2.1 Strings

A *string* is a finite sequence of characters over an alphabet $\Sigma = \{1, 2, .., \sigma\}$. We denote $S[1..n]$ a string of length n, and we use $S[i]$ to signify the i-th character of S. We denote by $S[i..j]$ a *substring* of S starting at position i until position j, both inclusive. Any substring $S[1..i]$ (starting from the first character) is called a *prefix* of S, and any substring $S[i..n]$ (including the last character) is called a *suffix* of S. We say that T is a *subsequence* of S if we can obtain T by deleting any number of characters from S. Conversely, we say that S is a *supersequence* of T.

For example, $S = GATTACA$ is a string of length 7, and $S[3..5] = TTA$ is a substring of S and $T = GTTC$ is a subsequence of S obtained by eliminating the second, fifth and seventh characters. $S[5..7] = ACA$ is a suffix of S and $S[1..4] = GATT$ is a prefix of S.

2.2 Edit distance and sequence alignment

There are many definitions of distances between two strings. A simple, yet versatile definition is *edit distance*, defined as the minimum number of *edit operations* required to transform one string into the other. Different sets of allowed operations lead to different distances. For instance, the *Levenshtein distance* considers substitutions, deletions and insertions as the possible operations. It is possible to generalize the Levenshtein distance by

assigning different costs to each operation and then defining the distance as the minimum-cost required to transform one string into the other.

A closely related concept is *sequence alignment*. A *pair-wise alignment* (or simply an *alignment*, when it is clear from the context) of A and B is a pair of sequences $(\mathsf{S}^A, \mathsf{S}^B)$ such that S^A is a supersequence of A, S^B is a supersequence of B, $|\mathsf{S}^A| = |\mathsf{S}^B| = n$ is the length of the alignment, and all positions which are not part of the subsequence A (respectively B) in S^A (respectively S^B), contain the *gap* symbol $'-'$, which is not present in the original sequences. Given a cost function $\mathscr{C}(a, b)$ to transform the character a into b, the cost of a pair-wise alignment is defined as

$$\mathscr{S}(\mathsf{S}^A, \mathsf{S}^B) = \sum_{i=1}^{n} \mathscr{C}(\mathsf{S}^A[i], \mathsf{S}^B[i]).$$

The weighted edit distance is defined as

$$\mathscr{S}(A, B) = min\{\sum_{1=1}^{n} \mathscr{C}(\mathsf{S}^A[i], \mathsf{S}^B[i]) : (\mathsf{S}^A, \mathsf{S}^B) \text{ is an alignment of } A \text{ and } B\}.$$

An alignment that achieves that value is called an *optimal alignment*. It is possible to use a *similarity function* instead of a cost function, in which case the minimum is replaced by a maximum, and the resulting measure is the *similarity* between strings A and B. A rich variety of models exist, considering inversions, translocations, non-linear costs for gaps, etc [26].

2.3 Graphs and trees

A directed graph $G = (V, E)$ is a pair of sets comprising the set of *vertices* $V = \{v_1, v_2, \ldots, v_{|V|}\}$ (also called *nodes*) and the set of *edges* $E \subseteq V \times V$ (also called *arcs*). An edge $(u, v) \in E$ is called an arc from u to v and we say that u and v are *adjacent*. When we consider $(u, v) \equiv (v, u)$ we say that the graph is *undirected*, otherwise we call it a *directed graph*.

A *path* of length p is a sequence of nodes $u_1 u_2 \ldots u_p$ such that consecutive nodes are connected by an arc, i.e. for all $1 \leq i < p$ it holds that $(u_i, u_{i+1}) \in E$. We say that u_1 and u_p are *connected*. A *cycle* is a path of length at least two from a node to itself. A graph that contains no cycle is called *acyclic*.

A *tree* is a graph where any two vertices are connected by exactly one path. A *rooted tree* is a graph when a specific node is designated as root. Unless stated otherwise, we will assume all the trees are directed and rooted.

2.4 Flow networks

A flow network is a directed graph where the edges have a *capacity* $c(u, v)$ and can be assigned a *flow* value $f(u, v) \leq c(u, v)$ Some vertices can be designated as *sources* and some others can be designated as *sinks*.

A valid flow is characterized by the flow conservation property: the total amount of flow entering a node must be equal to the total amount of flow exiting the node, with the exception of sources, which only have outgoing flow, and sinks, which only have incoming flow. More precisely for all nodes $u \in V$ that are neither a source nor a sink, it holds that

$$\sum_{(x,u)\in E} f(x, u) = \sum_{(u,y)\in E} f(u, y)$$

The maximum flow problem is, given a flow network, find a valid flow that is maximum. A classical solution to the max-flow problem is the Ford-Fulkerson algorithm [35], which relies on the notion of *residual network*. Given a network (G, V), the residual network (G_r, V_r) is the network with the same nodes and edges, capacities $c_r(u, v) = c(u, v) - f(u, v)$ and $f_r(u, v) = 0$. The Ford-Fulkerson algorithm finds an augmenting path in the residual network and adjusts the flow network along the same path so as to increase the total flow. When there is no augmenting path left in the residual network, the flow found is maximum. Assuming integral capacities (so that the flow increases at least by one unit each augmentation step), the running time is $O(|E||\varphi^*|)$, where E is the set of arcs of the flow network and $|\varphi^*|$ is the value of the maximum flow.

2.5 String matching and text indexing

Given a text $T[1..n]$ and a pattern $P[1..m]$, typically with $m \ll n$, the string matching problem is to find all the occurrences of P in T. This is sometimes called a *locate* query, while a *count* query is only to find the number of such occurrences, and an *existential* query asks whether this number is zero or not.

When the text is available for preprocessing the scenario is known as *indexed string matching*. Sometimes these indexes are called *full-text indexes* to highlight their difference from the indexes used in natural language that assume a text made of words or other structures. In this section we will assume, without loss of generality, that the text $T[1..n]$ is terminated with a unique character $T[n] = \$$ which is lexicographically smaller than

any other character.[1] In the following we review the foundational full-text indexes.

2.5.1 Suffix tree

A trie is a tree used to represent a set of strings. Each edge is labeled with a character, and for every node no two out-going edges can have the same label. Then each node v uniquely represents the string spelled by the path from the root to v. It follows that if a string S is represented by a node v in the trie, then all the prefixes of S are also represented, precisely by the nodes in the path from the root to v.

A suffix trie is a trie where all the leaves represent the suffixes of the text. All the proper substrings of T are represented by an internal node. Given a text $T[1..n]$ and a query pattern $P[1..m]$, the suffix trie of T can be used to solve the string matching problem as follows: starting from the root, if there is no edge labeled with $P[1]$ we know that P does not occur in T. If such an edge exists, we descend through it and repeat the process for $P[2]$ and so on until $P[m]$. If at the end of this process we are in a node, all the leaves that are reachable from that node corresponds to the positions in T where P occurs. The suffix trie uses $O(n^2 \log n)$ bits, and it can solve the string matching in time $O(|P|)$.

The *suffix tree* [96] is a compressed representation of the suffix trie, where all the unary paths are compacted, and the edges are labeled with the corresponding strings, encoded as a pair of pointers to the text. The resulting data structure requires $O(n \log n)$ bits of space and it solves the counting problem in $O(|P|)$ time and the locate problem in $O(|P| + occ)$ time assuming a constant size alphabet, in which case both of the times are optimal.

2.5.2 Suffix array

The *suffix array* [65] of T, is the permutation of $[1..n]$ that corresponds to the lexicographical order of the suffixes of T. More formally, it holds that $T[SA[i]..n] < T[SA[i+1]..n]$. It is worth noting that the suffix array corresponds to the values in the leaves of the suffix tree from left to right.

The suffix array uses $n\lceil \log n \rceil$ bits and offers a functionality similar to that of the suffix tree, with a moderate slowdown: counting the occurrences

[1] Assuming that the strings terminate with the special symbol \$, lexicographical smaller than any other character, is commonly adopted to avoid the special case when a suffix is a prefix of another suffix. For instance, this is needed to guarantee that the suffix trie of $T[1..n]$ has exactly n leaves.

of P can be done in $O(|P| \log n)$, and reporting their *occ* locations takes $O(|P| \log n + occ)$ time.

Compared to the suffix tree, the suffix array significantly reduced the space usage, however, in certain domains the $n\lceil \log n \rceil$ bits are still prohibitive.

2.6 Compressed full-text indexes

The quest for reducing the space usage of the indexes took a big leap with the development of *compressed full-text indexes*. Here, the indexes take advantage of the compressibility of the text itself, and the goal is to obtain indexes whose space requirement is proportional to the size of the compressed representation of the text, while still providing the search capabilities of the indexes mentioned in the previous section. Another way of seeing this is as a compressed representation of the text that can be rapidly searched.

There are many approaches to design compressed full-text indexes. For instance, it has been noted that repetitions in the text translate into runs in the suffix array, which can be exploited to compress it. Many *compressed suffix arrays* have been developed based on this and similar ideas [39, 6, 72].

A related although different approach is based on the Burrows-Wheeler Transform (BWT), which was originally designed for text compression. The BWT is a reversible transformation of the text which usually is easier to compress than the text itself [12]. Later, Ferragina and Manzini discovered that the BWT can be used to solve the string matching problem [30]. It is possible to emulate the suffix array using the BWT to find the suffix array interval corresponding to the occurrences of the query pattern.

There is a large family of FM-Indexes, depending on the precise representation of the BWT and other supporting data structures. For instance, the alphabet-friendly FM-index [32] requires $nH_k(T) + o(n \log \sigma)$ bits[2] and it can count the occurrences of P in T in time $O(|P| \log \sigma)$. For detailed descriptions and different approaches to compressed text indexes we refer the reader to a comprehensive survey [72].

[2] $H_k(T)$ is the kth-order entropy of T, and $nH_k(T)$ is the space in bits used by a statistical compression model that uses contexts of length k. In the worst case (random text), $H_k(T) = \log \sigma$ and this term is equal to the plain encoding of the text.

Chapter 3

Analysis of Relative Lempel Ziv with Artificial References

Almost forty years have passed since Abraham Lempel and Jacob Ziv published the so-called LZ77 and LZ78 algorithms [100, 101]. Today, they remain central to data compression: Those algorithms and their derivatives are at the core of widely used compression tools like zip, gzip, 7zip, image formats like GIF and PNG, and also in modern libraries like Brotli and Zstandard used to speed-up web browsing.

These algorithms are still actively researched [2, 7, 34, 25], and new variants are still being proposed to address new challenges [51, 56]. Relative Lempel Ziv (RLZ) [56] is a recent variation of LZ77 that has been particularly successful in the compression of highly repetitive collections.

In this section we review the contributions of Paper I, where the compression achieved by using RLZ with artificial reference construction is analyzed both from a theoretical and from a practical perspective.

3.1 LZ77 and Relative Lempel Ziv

In the literature there are slightly different versions of the LZ77 parsing. We will adopt the following one: given a text $T[1..n]$, the LZ77 parsing of T is a partitioning $T = T^1T^2 \dots T^z$ such that for $1 \leq i \leq z$, with $p_i = \sum_{j=1}^{i-1} |T^j|$ it holds that either:

- T^i is a character that does not occur in $T[1..p_i]$ or
- T^i is the longest string that is simultaneously a prefix of $T[p_i + 1..n]$ and a substring $T[1..p_i]$

We will call the former *literal phrases* and the latter *copying phrases*. Both types of phrases can be represented as pair of integers (pos, len). For copying phrases, pos is the position in $T[1..p_i]$ in which T^i occurs and $len = |T^i|$ is its length. For literal phrases $pos = 0$ and len is the binary representation of the new character.

The above definition is sometimes referred to as *greedy LZ77 parsing* because, parsing from left to right, the next phrase is always the *longest* possible. The greedy parsing always produces the minimum number of phrases [78], however, when the output size is considered under different encoding schemes, more sophisticated non-greedy parsings are needed to achieve optimality [33].

Although several linear-time algorithms for computing the LZ77 parsing are known (c.f. [78]), there is ongoing research on how to improve the practical efficiency [50, 49]. An important challenge is the scenario where the text to be processed is too big to fit in memory. One strategy is to make efficient use of external memory [51]. Another is to introduce changes into the algorithm, such as a user-defined parameter to bound the distance between a phrase and its source. This approach, known as *fixed window LZ77*, has been used since the early days of Lempel-Ziv algorithms, and it remains present in popular tools such gzip. However, when the repeated substrings are too distant in the text, this model will achieve little compression.

In 2010, Kuruppu, Puglisi and Zobel introduced Relative Lempel Ziv (RLZ), a novel variation of LZ77 originally designed for compression of genomes. In RLZ, in addition to the text to be compressed, there is another text called *the reference* (or *the dictionary*), which is expected to be much shorter than the text. Similar to LZ77, the text is greedily parsed into phrases, but now the phrases correspond to occurrences in the dictionary instead of the text itself. More precisely, given a text $T[1..n]$ and a dictionary $D[1..d]$, and assuming that $T[1..i-1]$ has already been parsed, the next phrase corresponds to the largest $T[i..j]$ such that $T[i..j] = D[pos..pos+j-i+1]$ for some pos. The dictionary is also part of the output, and the phrases can be codified as pairs (pos, len) analogously to LZ77.

3.2 Artificial reference for RLZ

In collections where all the documents are highly similar to each other, taking any of the documents as a reference should produce a decent level of compression. Paradigmatic examples of this are genomic databases where all the stored sequences are individuals from the same species. However,

there are datasets that are still highly compressible but without such a homogeneous structure. In the latter setting, it is possible to build an *artificial reference* [44], by sampling substrings from the entire dataset and concatenating them. This approach has exhibited excellent performance in practice [44, 91, 61].

In the following sections we review the contributions of Paper I, where we analyzed the functioning of RLZ compression using artificial reference.

3.3 Theoretical analysis

We begin with an intuitive view about the artificial reference RLZ. Consider a somewhat short substring of the text: on the one hand, if it is frequent enough it is likely to be sampled and thus be part of the reference. Therefore, all its occurrences can be represented as a single pair (pos, len) and will use little space. On the other hand, if it is not so frequent all its occurrences can be encoded literally without consuming much space. The rest of this section presents a formal analysis of this reasoning.

Consider a string $T[1..n]$ over an alphabet of size σ whose LZ77 parse consists of z phrases. Given integers k and ℓ, let us sample k substrings of length ℓ from T and concatenate them, appending the first and last ℓ characters of T, to obtain the reference. This takes $\mathcal{O}(k\ell \log \sigma)$ bits.

For our analysis we use a partition of T into $\mathcal{O}(n/\ell)$ blocks of length at most $\ell/2$ such that $\mathcal{O}(z \log n)$ of them are distinct, for $2 \leq \ell \leq n$. A previous result by Gawrychowski [37] shows that such a partition exists when there is a grammar of size $z \log n$ that represents T; moreover, it is possible to obtain such a grammar from the LZ77 representation of T [83].

Let us consider the ith distinct block, whose frequency is f_i. The probability that one of its occurrences is completely included in some of the sampled substring is at least $1 - p_i$, where $p_i = \left(1 - \frac{f_i \ell}{2n}\right)^k$. In that case, all of its occurrences are stored using $\mathcal{O}(f_i \log(k\ell))$ bits in total. Otherwise, they are stored literally using $\mathcal{O}(f_i \ell \log \sigma)$ bits.

Let $b = \mathcal{O}(z \log n)$ be the number of distinct blocks. The expected size in bits of the RLZ parse is:

$$\mathcal{O}\left(\sum_{i=1}^{b} f_i(1-p_i)\log(k\ell) + \sum_{i=1}^{b} f_i p_i \ell \log \sigma\right) \leq \mathcal{O}\left(\frac{n \log(k\ell)}{\ell} + \ell \log \sigma \sum_{i=1}^{b} f_i p_i\right)$$

Since $1 - x \leq e^{-x}$, we have $p_i = \left(1 - \frac{f_i \ell}{2n}\right)^k \leq \frac{1}{e^{f_i k\ell/2n}}$ so $\sum_{i=1}^{b} f_i p_i \leq \sum_{i=1}^{b} \frac{f_i}{e^{f_i k\ell/2n}}$, which is concave and, thus, maximum when all the distinct blocks occur equally often. Therefore, calculation shows

$$\ell \log \sigma \sum_{i=1}^{b} f_i p_i = \frac{n \log \sigma}{e^{\Omega(k/z \log n)}} .$$

Summing up, we obtain the following theorem:

Theorem 3.1 (Paper I) *If we randomly sample and concatenate k blocks of length ℓ from the dataset to form an artificial reference, then the expected total size in bits of the RLZ encoding (i.e., the reference and the parse together) is bounded by*

$$\mathcal{O}\left(k\ell \log \sigma + \frac{n \log(k\ell)}{\ell}\right) + \frac{n \log \sigma}{e^{\Omega(k/z \log n)}} .$$

It follows that the following conditions guarantee good (expected) compression:

1. $k\ell \ll n$,
2. $\log(k\ell) \ll \ell \log \sigma$,
3. $k = \omega(z \log n)$.

The first condition states that the reference should be sufficiently smaller than T. The second condition states that a pointer to the reference should be sufficiently smaller than the literal encoding of the string that it is representing. The third condition is more interesting, as it reveals an asymmetry: choosing k, say, a tenth larger than optimal should not increase the size of the encoding by more than about a tenth, however, choosing k a tenth *smaller* than optimal could drastically worsen compression.

A careful selection of the sampling values k and ℓ leads to the following result:

Corollary 3.1 (Paper I) *If we randomly sample and concatenate $k = z \log^{2+\epsilon} n$ blocks of length $\ell = \sqrt{n/k}$, then the expected total size of the RLZ encoding is $\mathcal{O}\big((nz)^{1/2} \log^{2+\epsilon} n\big)$*

Which shows that if LZ77 compress well, then RLZ with an artificial reference can compress well too.

3.4 Experimental analysis

In our experimental study, we considered different $1GB$ collections containing web crawls, protein sequences or synthetic datasets. Their detailed description is given in Paper I. We built an artificial reference by picking values of k and ℓ and then concatenating k randomly sampled substrings of length ℓ from the dataset, storing the resulting reference in $k\ell \log \sigma$ bits. Then we ran the RLZ algorithm, and stored the phrases using $\log(k\ell)$ bits for pointers and $\log \ell$ bits for phrase lengths. We used two different regimes to choose the k and ℓ values:

- ***Fixed length*** used a fixed a value of ℓ and tried different values of k.
- ***Corollary 1*** used the values of k and ℓ prescribed by Corollary 3.1, using varying estimates of z.

The points representing the total size for each sampling scheme are connected by lines, and below each of them we plot an extra point representing the size used only by the reference. As a baseline, we included the size of the LZ77 encoding using $2 \log n$ bits for each phrase.

In Figures 3.1 and 3.2 we reproduce the results from Paper I. The curves we obtained are in accordance with Theorem 3.1: there is a sharp drop on the left, corresponding with the expected damage to compression by using too small k values. In this area increasing the number of samples significantly improves compression. At some point, the curves start to rise roughly parallel to the reference size, which increases linearly. This improves the understanding of why the reference pruning [91, 61] approach works: it is much safer to oversample in the beginning than to undersample.

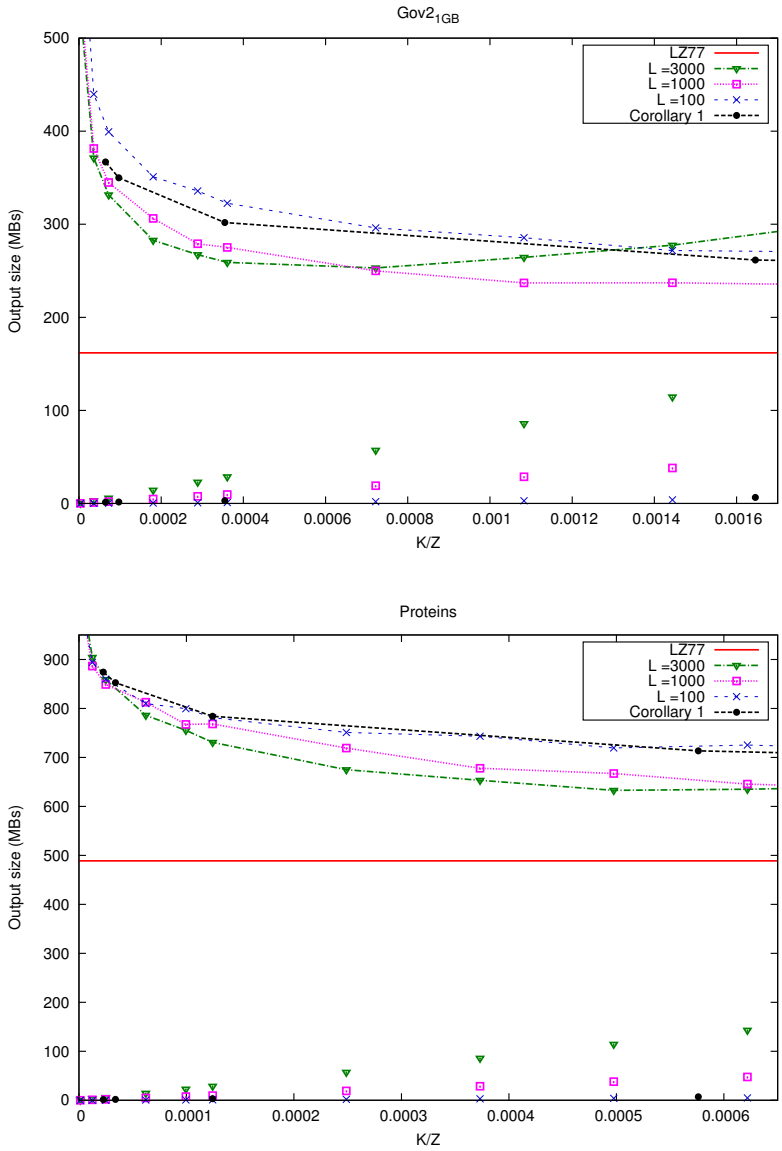


Figure 3.1: Size in bits of RLZ compression using different artificial references on two 1GB collections: Gov2$_{1GB}$ is part of a web crawl from the TREC collection, and Proteins is a set of protein sequences obtained from the Swissprot database. Different regimes for constructing the reference are shown for each collection: fixed length of samples of size 3000, 1000, and 100 characters each, and varying amounts (k value) for each of them. The strategy proposed by Corollary 1 is also used. The curves show the total size of the encoding as a function of k/z. The red horizontal line is the size of the LZ77 encoding as a baseline.

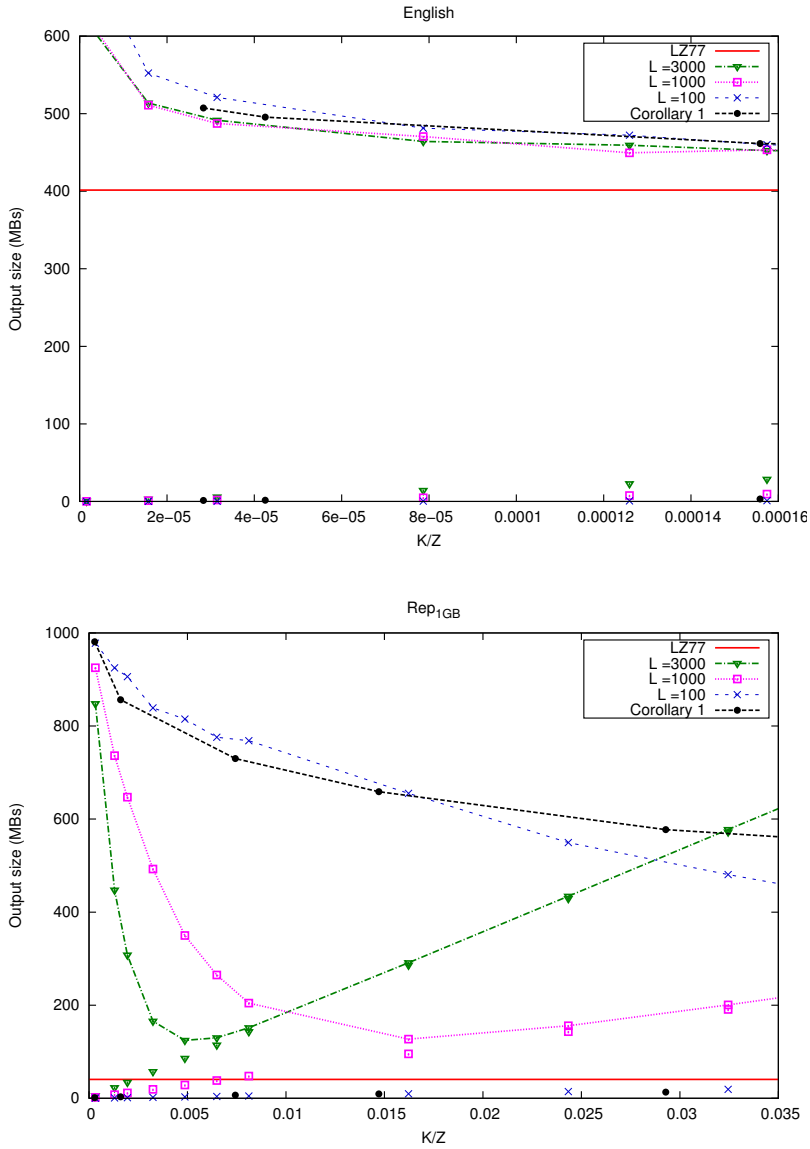


Figure 3.2: Size in bits of RLZ compression using different artificial references on two 1GB collections: English is a concatenation of English text from the Gutenberg Project, and Rep_{1GB} is a very repetitive synthetic file made of 400 copies of a random 25MB string. Different regimes for constructing the reference are shown for each collection: fixed length of samples of size 3000, 1000, and 100 characters each, and varying amounts (k value) for each of them. The strategy proposed by Corollary 1 is also used. The curves show the total size of the encoding as a function of k/z. The red horizontal line is the size of the LZ77 encoding as a baseline.

Chapter 4

CHICO: A Compressed Hybrid Index for Repetitive Collections

Given a text $T[1..n]$ and a query pattern $P[1..m]$, the string matching problem is to find all the occurrences of P in T. This is a central problem in computer science, and the variant where T is available to be preprocessed is called the *indexed pattern matching* problem.

Since it became evident, in the 80s [17], that indexed string matching is relevant for analysing biological sequences, bioinformatics has played an important role in pushing the development of the field. To increase applicability of index data structures there has been a continuous effort to reduce their size while retaining strong search capability. A classic example is the Suffix Array [65], which uses much less space than the Suffix Tree [96], and can provide the same functionality when augmented appropriately [1]. This trend has led to the development of a whole research area, *compressed indexing* [72], where the idea is to build an index that uses space proportional to a compressed representation of the text. Particularly important results, like the FM-Index [30, 31], are based on the Burrows-Wheeler transform (BWT) [12]. Those indexes are currently at the heart of widely-used *read aligners* such as BWA [60] and BowTie2 [57].

A recent trend in this context is indexing *repetitive collections*. The motivation again is nurtured from bioinformatics: projects like 1000 genomes and UK10K are examples of highly repetitive collections where providing pattern matching functionalities can be valuable.

In this chapter we present CHICO, the contribution of Paper II. CHICO is a compressed index for approximate string matching that combines Lempel-Ziv algorithms with the FM-Index to efficiently handle very large and repetitive collections.

4.1 Kernelization

CHICO is an improved version of the Hybrid Index of Ferrada, Gagie, Hirvola and Puglisi [29]. Their strategy is to find a *kernel string* that is, on a highly repetitive collection, much shorter than the input text and that it is also enough to solve the approximate pattern matching problem in it.

This index takes as an input the text, its LZ77 parsing, and also upper bounds M and K for the pattern lengths and edit distances respectively. The kernel text $\mathcal{K}_{M,K}$ is defined as the subsequence of T that retains only characters within distance $M+K-1$ from their nearest phrase boundaries. Figure 4.1 illustrates. Characters not contiguous in T are separated in $\mathcal{K}_{M,K}$ by $K+1$ copies of a special separator #.

To understand why the kernel string suffices to solve existential queries it is useful to adopt the following definitions: A *primary occurrence* is an (exact or approximate) occurrence of P in T that spans two or more LZ77 phrases. A *secondary occurrence* is an (exact or approximate) occurrence of P in T that is entirely contained in one LZ77 phrase. Farach and Thorup [28] noted that every secondary match is an exact copy of a previous (secondary or primary) match, and Kärkkäinen and Ukkonen [52] described how to find the secondary occurences from the primary occurrences, using the structure of the LZ77 parse.

From the definition of the kernel string it is easy to see that any substring of T with length at most $M+K$ that crosses a phrase boundary in the given parse of T has a corresponding and equal substring in $\mathcal{K}_{M,K}$. Therefore, all the primary occurrences of patterns will be found in $\mathcal{K}_{M,K}$, given the bounds M on the pattern length and K on the edit distance.

To locate all the occurrences in T, including secondary occurrences, additional data structures are needed. First, the positions of the phrases boundaries in the original text and their counterparts in the kernel string are stored. Using them it is possible to locate primary occurrences in T from their locations in $\mathcal{K}_{M,K}$. Then, phrases (pos, len) of the parsing can be represented as triplets $(x, y) \rightarrow w$ where $(x = pos, y = pos + len)$ is called *source*, and w is the position in the text where the phrase starts. We will not provide the details of the representation nor the exact process to report the secondary occurrences, but the general idea is to work recursively as follows: for each occurrence to be reported, the data structure is queried to find all the phrases whose sources entirely contain said occurrence. That implies that there is another occurrence to report, corresponding to a copy of the current occurrence, which is then also processed recursively.

4.2 LZ77-relaxed parsings

CHICO introduces the notion of *LZ77-relaxed parsings*[1]. This is a generalization of LZ77 achieved by dismissing two constraints. Firstly, the parsing does not need to be greedy. That is, a copying phrase does not need to be as long as possible – *any* previous factor is admissible. Secondly, literal phrases can be more than one character long. Observe that this defines a class of parsings that includes the greedy LZ77 parsing.

Paper II improves the Hybrid Index by allowing the use of any LZ77-relaxed parsing to build the kernel string. The main consequence is that now the content of the literal phrases always goes to the kernel string. The extra cost that this modification generates to the index is a bitmap of size z that indicates with a 1 the literal phrases and is 0 elsewhere. This is needed due to the introduction of literal phrases that represent strings and not merely single characters.

4.3 Reducing the number of phrases

From the definition of the kernel string it follows that all the phrases shorter than $2M$ will be added entirely to the kernel string. Using an LZ77-relaxed parsing we can get rid of those vacuous phrases.

To reduce the number of phrases we first transform all phrases whose length is smaller than $2M$ into literal phrases. Then, every time that two literal phrases are adjacent they are merged. In Figure 4.1 it is easy to see that this process will not alter the contents of the kernel string.

It would be possible to use a specialized Lempel-Ziv parser that is aware of M so it avoids copying phrases shorter than $2M$. Alternatively, if the parsing is obtained by a general-purpose parser, the phrase reduction can be quickly computed in a single pass.

4.4 Faster construction with RLZ

The use of LZ77-relaxed parsings not only allowed us to reduce the number of phrases but also brought the possibility of faster construction by using RLZ parsing. In its original definition the RLZ parsing does not fall into the LZ77-relaxed parsing category, as it introduces a new element, the dictionary. However, it is possible to modify the greedy LZ77 algorithm in a way that *almost* results in the RLZ algorithm. We will use an LZ77

[1] In Paper II this is called LZ77 valid parsing. Here we opt for *relaxed parsing* to avoid any confusion with LZ77

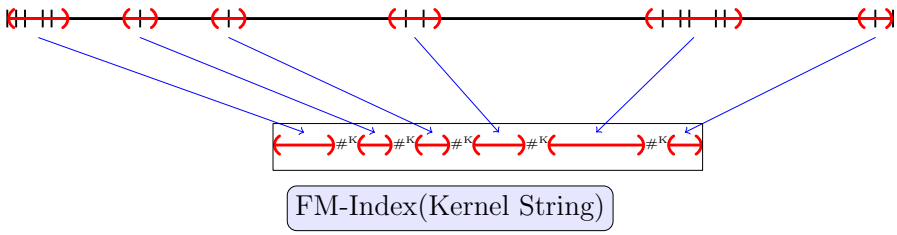


Figure 4.1: Schematic view of the construction of the kernel string. Vertical lines represent the phrase boundaries of an LZ77 parsing. All the characters within distance $M + K - 1$ to some phrase boundary are kept for the kernel string, while everything else is discarded. Note that the shorter phrases are going entirely to the kernel.

parsing, where each new phrase is the longest factor *within a prefix* of T. This is equivalent to computing the greedy LZ77 parsing of the given prefix, and then computing the RLZ parsing of the rest of the text using the said prefix as a dictionary.

It is clear that this approach will work well when the text is repetitive and homogeneous: if T corresponds to the concatenation of 1000 human genomes, then any prefix longer than one human genome will make a good reference to compress the rest of the collection.

4.5 In practice

We implemented the index in C++ making use of the Succinct Data Structure Library 2.0.3 (SDSL) [38] for most of the component succinct data structures.

We indexed the kernel text using an SDSL implementation of the FM-Index. As this FM-Index does not offer native support for approximate string matching, our experimental study used only exact queries.

To explore different trade-offs we studied different alternatives for the LZ77 factorization, all of them specialized in repetitive collections.

- **LZscan** [49], an in-memory algorithm that computes the LZ77 parsing.
- **EM-LZscan** [51] an external memory algorithm to compute the LZ77 parsing
- **RLZ** based on the RLZ of Hoobin et al. [44], which we modified according to Section 4.4, using KKP [50] to compute the LZ77 parsing of the reference.

- **PRLZ** Our parallel version of the previous one, implemented using OpenMP.

For each of them we ran the phrase reduction phase of Section 4.3. The first method is the preferred one when there is enough memory to construct the LZ77 parsing in memory, otherwise, any of the others three methods may be used.

In Paper II we compared CHICO against other indexes for repetitive collections using datasets from the pizzachili corpus, using LZscan to build the index. The results shows that it is competitive in terms of index size and query time, while clearly dominating the alternatives in terms of indexing time.

In larger collections we first compared the EM-LZscan and RLZ construction methods, observing that using RLZ the index can be built more than 50 times faster, with little impact on the size index. Table 4.1 reproduces the results in one of those collections. Then we compared the parallel version of RLZ against the sequential one. The results of one of those experiments is reproduced here in Table 4.2. Finally, we tested our index in a 2.4TB collection[2] that took about 12 hours to index and in which exact occurrences of query patterns were found in less than 100ms .

[2] The DNA collections used to test scalability were supposed to be "x versions of human chromosome y". However, due to a regrettable mistake in Paper II this is not the case. The misuse of an external tool to generate the data resulted on genetic variants being greedily applied to the first genome with no overlapping variants, generating a collection with some genomes containing most of the variants, and many identical genomes. As a consequence, although those experiments prove that the method can handle large collections, the resulting times and spaces do not reflect the behavior on a proper collection of genomes. In the next chapter this is handled properly, so e.g. Table 5.1 gives a better idea of the behavior of CHICO on real data.

	Size (bpc)	Build time (min)	Query Time (ms)	
			$\|P\| = 50$	$\|P\| = 70$
EM-LZ	0.00126	4647	18.16	14.90
$\text{RLZ}_{0.5GB}$	0.0060	143	55.28	46.67
RLZ_{1GB}	0.0047	65	50.30	40.72

Table 4.1: Different parsing algorithms to index collection $\texttt{CHR}_{21}$, a 90GB synthetic collection made of DNA. The first row shows the results for EM-LZ, which computes the LZ77 greedy parsing. The next rows show the results for the RLZ parsing using prefixes of size 500MB and 1GB. The size of the index is expressed in bytes per char, the building times are expressed in minutes, and the query times are expressed in milliseconds. Query times were computed as an average of 1000 query patterns randomly extracted from the collection.

	Size	Build time (min)		Query Time (ms)	
	(bpc)	LZ Parsing	Others	P=50	P=70
RLZ_{1GB}	0.00656	308	51	225.43	178.47
PRLZ_{1GB}	0.00658	22	55	224.04	181.21

Table 4.2: Results using RLZ and PRLZ to parse $\texttt{CHR}_{14}$, a 201 GB synthetic collection made of DNA. Query times were computed as an average of 1000 query patterns randomly extracted from the collection.

Chapter 5

Pan-Genome Read Alignment to Improve Variation Calling

An important motivation that we have discussed in previous chapters is *read alignment*, where short NGS reads are mapped to a (much larger) genomic reference. In essence, to do read alignment is to solve approximate pattern matching.

Among the many applications of read alignment [57], here we focus on *variation calling*, the process of characterizing an individual's genome by finding how it differs from a reference. The standard approach is to obtain a set of reads from the donor, to map them to a single reference genome using a read aligner, and analyzing the read pile-up to infer the variants that occur in the donor's genome.

However, our current knowledge about the human genome is pan-genomic [19]: after the first human genome was sequenced, the cost of sequencing has decreased dramatically, and today many projects are curating huge genomic databases.

In this chapter we present the contribution of Paper III, a new framework for variant calling with short-read data utilizing a pan-genomic reference. As an intermediate result we provide a fully scalable pan-genome read aligner based on the index of Chapter 4.

5.1 Pan-genome indexing

As DNA repositories keep growing the term *pan-genome* is being used more frequently [66, 87, 88, 19] – even getting media coverage [81, 73] – although not always with the same meaning. A recent attempt for a broad definition states that a *pan-genome* is any collection of genomic sequences to be ana-

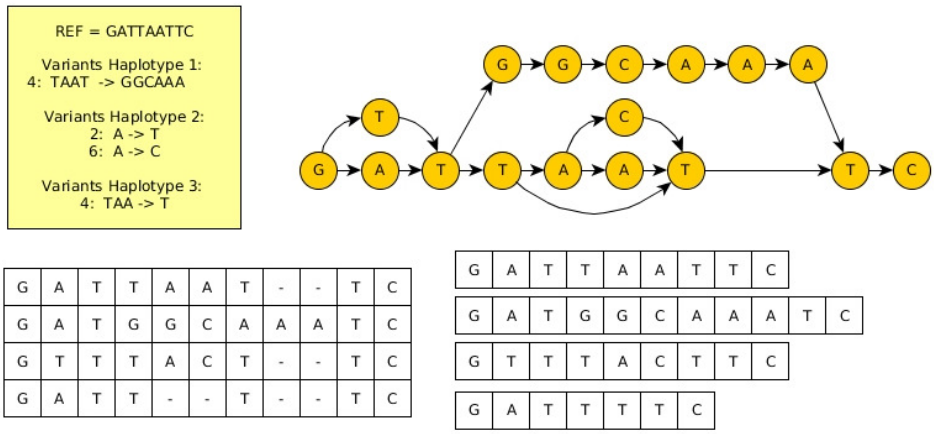


Figure 5.1: Four different representations of a pan-genome that corresponds to the same set of individuals. Top left: a reference sequence plus a set of variants to specify the other individuals. Top right: a (directed acyclic) graph representation. Bottom left: a multiple sequence alignment representation, Bottom right: a set of sequences representations.

lyzed jointly or to be used as a reference [19]. Once we adopt this general definition for pan-genomic reference, we are left with two closely related questions: how to represent a pan-genomic reference, and how to index it. Previous efforts can roughly be categorized into three classes: one can consider (i) a graph representing a reference and variations from it, (ii) a set of reference sequences, or (iii) a modified reference sequence. See Paper III for a review on these approaches [86, 84, 46, 21, 24, 71, 29, 23, 64].

The simple class (ii) model, a set of sequences, has been actively studied from a computer science perspective, achieving good results in terms of time and space efficiency using scalable methods. Unfortunately, unlike classes (i) and (iii), it has not been tested for enhancing variation calling. Here we aim to fill this gap. We do it by adopting a multiple sequence alignment representation of the pan-genome. On the one hand, it is easily transformed into a set of sequences and vice versa. On the other hand, similarly to class (i) and (iii), the multiple sequence alignment has enough structure to make it an attractive choice for a reference.

5.2 Pan-genomic references to improve variant calling

In this section we present our framework for variation calling using read alignment to a pan-genomic reference. First we give an overview, and then we dwell on the relevant details. Figure 5.2 illustrates our scheme.

Our approach represents the pan-genome reference as a multiple sequence alignment (Figure 5.1 bottom left). We index the underlying set of sequences in order to align the reads to the pan-genome. After aligning all the reads to the pan-genome we perform a read pileup on the multiple sequence alignment of reference genomes. The multiple sequence alignment representation of the pan-genome lets us extract a linear ad hoc reference easily. Such a linear ad hoc reference represents a possible recombination of the genomic sequences present in the pan-genome that is closer to the donor than a generic reference sequence. The ad hoc reference is then fed to any standard read alignment and variation detection workflow. Finally, we need to normalize our variants: after the previous step, the variants are expressed using the ad hoc reference instead of the standard one. The normalization step projects the variants back to the standard reference.

In the following sections we provide a detailed description of each component of our workflow.

5.2.1 CHIC aligner

We developed CHIC aligner, an extended version of the previous chapter's index. Using it we can perform read alignment to the set of sequences that makes the pan-genome. Recall that CHICO indexes a set of sequences using a recent data structure called hybrid index [29]. The hybrid index uses the Lempel-Ziv compression algorithm to factor out repetitions in the sequences and builds a sequence containing only the non-repetitive content. This sequence has been called kernel sequence, because to solve the pattern matching problem in the whole sequence, it is enough to solve it in the kernel sequence. In addition to the kernel string and its index, CHIC aligner also stores the phrase boundaries, and some auxiliary information. This is necessary to project the coordinates of the alignments from the kernel string to the original sequences. Different algorithms from the Lempel-Ziv algorithms can be used to construct the index, leading to different trade-offs. Among them we chose Relative Lempel-Ziv (RLZ) [56] because of its many virtues for the pan-genome scenario: it offers a very fast construction algorithm with moderate memory requirement and little compromise in the size of the index. Furthermore it can be easily run in parallel. Once the

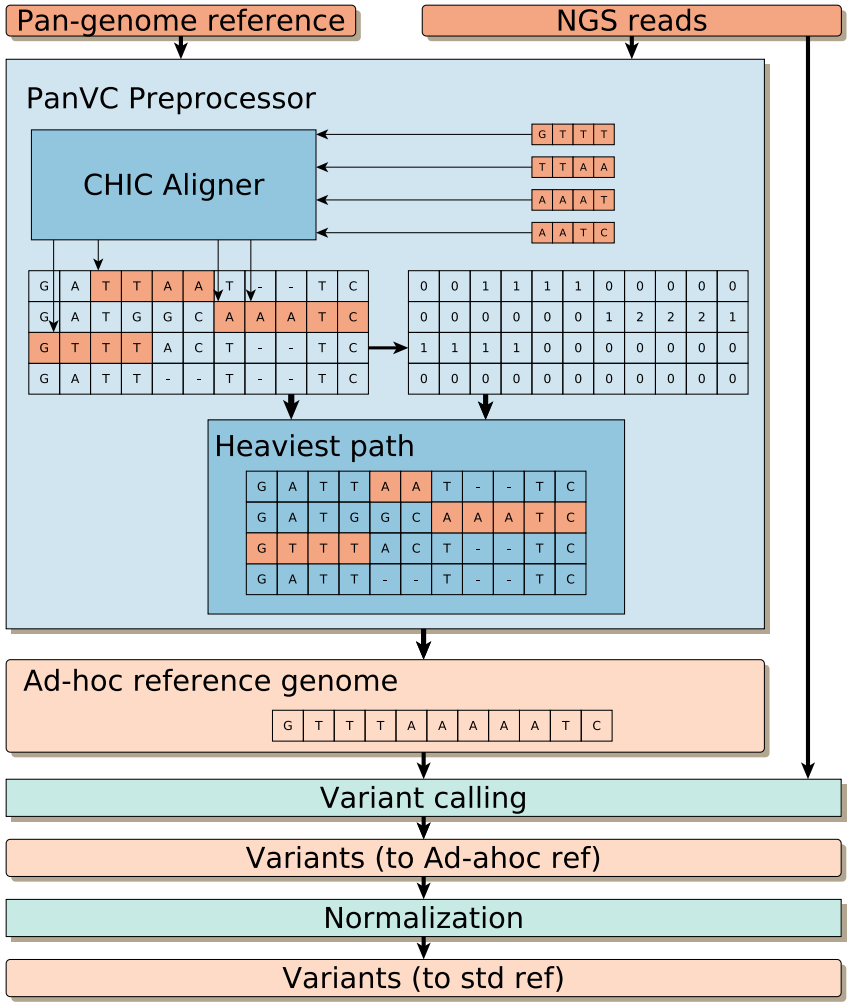


Figure 5.2: Schematic view of our PanVC workflow for variation calling, including a conceptual example. The pan-genomic reference comprises the sequences GATTATTC, GATGGCAAATC, GTTTACTTC and GATTTTC, represented as a multiple sequence alignment. The set of reads from the donor individual is GTTT, TTAA, AAAT and AATC. CHIC aligner is used to find the best alignment of each read. In the example, all the alignments are exact matches starting in the first base of the third sequence, the third base of the first sequence, the seventh base of the second sequence, and on the eight base of the second sequence. After all the reads are aligned, the score matrix is computed by incrementing the values of each position where a read aligns. With those values, the heaviest path algorithm extracts a recombination that takes those bases with the highest scores. This is the ad hoc genome which is then used as a reference for variant calling using GATK. Finally the variants are normalized so that they are using the standard reference instead of the ad hoc reference.

kernel sequence is built, CHIC aligner indexes it with a standard read aligner, such as Bowtie2 or BWA.

A feature that CHIC aligner has in common with some previous indexes for pan-genomes is that during indexing time, a context size M is given as a parameter. This sets an upper bound for the read length that can be aligned. In all our experiment we use $M = 120$.

CHIC aligner interface is compatible with other reads aligners: the input is a multi-fasta file with the input genomes and the alignments are output in the SAM format [59]. Our index offer two main reporting modes for the alignments: The first one looks for the best alignment in the kernel string and only its projection back to the original input is reported. The second one also looks only for the best match in the kernel string. Then its projection to the input is reported, and also all possible repetitions of such alignment are reported as well. Those can be combined with Bowtie2 (or BWA) reporting options, and instead of looking for the best match in the kernel string, it is possible to report all occurrences, or the best k ones, where k is a user-defined parameter. In our experiments we report only the best alignment, as it is expected that many repetitions occur in the pan-genomic context.

5.2.2 Pan-genome representation

As discussed in Section 5.1, we represent the pan-genome as a multiple sequence alignment and we index the underlying sequence using CHIC aligner. To transform one representation into the other and to be able to map coordinates we store bitmaps to indicate the positions where the gaps occur. Consider our running example of a multiple alignment

```
GATTAAT--TC
GATGGCAAATC
GTTTACT--TC
GATT--T--TC
```

We may encode the positions of the gaps by four bitvectors:

```
11111110011
11111111111
11111110011
11110010011
```

Let these bitvectors be B_1, B_2, B_3, and B_4. We extract the four sequences omitting the gaps, and preprocess the bitvectors for constant time `rank`

and `select` queries [48, 16, 68]: $\texttt{rank}_1(B_k, i) = j$ tells the number of 1s in $B_k[1..i]$ and $\texttt{select}_1(B_k, j) = i$ tells the position of the j-th 1 in B_k. Then, for $B_k[i] = 1$, $\texttt{rank}_1(B_k, i) = j$ maps a character in column i of row k in the multiple sequence alignment to its position j in the k-th sequence, and $\texttt{select}_1(B_k, j) = i$ does the reverse mapping, i.e. the one we need to map a occurrence position of a read to add the sum in the coverage matrix.

These bitvectors with rank and select support take $n+o(n)$ bits of space for a multiple alignment of total size n [48, 16, 68]. Moreover, since the bitvectors have long runs of 1s (and possibly 0s), they can be compressed efficiently while still supporting fast rank and select queries [79, 72].

5.2.3 Heaviest path extraction

After aligning all the reads to the multiple sequence alignment, we extract a recombined (virtual) genome favoring the positions where most reads were aligned. To do so we propose a generic approach to extract such a heaviest path on a multiple sequence alignment. We define a score matrix S that has the same dimensions as the multiple sequence alignment representation of the pan-genome. All the values of the score matrix are initially set to 0.

We use CHIC aligner to find the best alignment for each donor's read. Then we process the output as follows. For each alignment of length m that starts at position j in the genome i of the pan-genome, we increment the scores in $S[i][j], S[i][j+1] \dots S[i][j+m-1]$. When all the reads have been processed we have recorded in S that the areas with highest scores are those where more reads were aligned.

Then we construct the ad hoc reference as follows: we traverse the score matrix column wise, and for each column we look for the element with the highest score. Then, we take the nucleotide that is in the same position in the multiple sequence alignment and append it to the ad hoc reference. This procedure can be interpreted as a heaviest path in a graph: each cell (i, j) of the matrix represents a node, and for each node (i, j) there are N outgoing edges to nodes $(i+1, k)$, $k \in \{1, \dots, N\}$. We add an extra node A with N outgoing edges to the nodes $(1, k)$, and another node B with N ingoing edges from nodes (L, k). Then the ad hoc reference is the sequence spelled by the heaviest path from A to B. The underlying idea of this procedure is to model structural recombinations among the indexed sequences.

A valid concern is that the resulting path might contain too many alternations between sequences in order to maximize the weight. To address this issue we propose an algorithm to extract the heaviest path, constrained to have a limited number of jumps between sequences (see Paper III, Supple-

ment B). However, in our experiments we noticed that the unconstrained version presented in this section performs well.

It is worth noting that as opposed to a graph representation of the pan-genome where the possible recombinations are limited to be those pre-existing in the pan-genome, our multiple sequence alignment representation can also generate novel recombinations by switching sequences in the middle of a pre-existing variant. This happens in our example in Figure 5.2, where the ad hoc reference could not be predicted using the graph representation of the same pan-genome shown in Figure 5.1.

5.2.4 Variant calling

Variant calling can be in itself a complex workflow, and it might be tailored for specific type of variants (SNVs, Structural Variants), etc. We aim for a modular and flexible workflow, so any workflow can be plugged in it. The only difference is that we will feed it the ad hoc reference instead of the standard one. In our experiments, we used GATK [4] version 3.3, following the Best Practices: first we aligned the reads to the reference using BWA, and next we used Picard to sort the reads and remove duplicates. Then we performed indel realignment using GATK RealignerTargetCreator and IndelRealigner, and finally we called variants using GATK HaplotypeCaller using parameters `genotyping mode = DISCOVERY`, `standemit conf = 10` and textttstandcall conf = 30.

5.2.5 Normalizer

Finally we need to normalize our set of variants. To do so we apply the variants to the ad hoc reference, so that we obtain an alignment between the ad hoc reference and the predicted sequence. The metadata generated in the preprocessor stage – while extracting the heaviest path – includes an alignment between the standard reference and the ad hoc reference. Using those, we can run a linear-time algorithm to obtain an alignment between the standard reference and the predicted sequence. From this alignment, we can generate a vcf file that expresses the predicted sequence as a set of variants from the standard reference.

5.3 Experiments

5.3.1 Read alignment

The experiments on CHIC aligner have two purposes. The first one is to study to what extent including more sequences in the reference pan-genome increases the number of mapped reads. The second one is to measure the cost of having such a large pan-genome reference. We built pan-genome references that consist of 21 and 201 different human genomes. These were built using only the consensus GRCh37, or the consensus plus 10, or plus 100 individuals from the 1000 genomes project [89]. We generated them using two haplotypes from each individual.

To carry out our experiment in a realistic set we aligned the reads from a new donor who is not part of the pan-genome reference. Thus, we took a sample of 10^6 paired-end reads from a recent study on the genome of a Mongolian individual [5]. We measured the indexing time for each pan-genome reference, the size of the resulting index and the total time to align the reads to the reference. Also we report the number of mapped reads. As a baseline, we compare against Bowtie2 using a single human genome as a reference. The results are shown in Table 5.1.

		Indexing		Alignment	
Aligner	Input size	Time	Index Size	Time	Mapped reads
BOWTIE2_1	2.7GB	1h	3.708GB	5.32s	782883
CHIC_{21}	57GB	5h	25GB	2m	785552
CHIC_{201}	540GB	35h	180GB	24m	786026

Table 5.1: Scalability of CHIC aligner. Time required to build the index, to align a set of 10^6 paired-end reads, and the number of unaligned reads. BOWTIE2_1 indexes the human reference, CHIC_x indexes x different versions of the human genome (only numbered chromosomes).

The results show that the time needed to align the reads to the pan-genomic reference are larger than in the single reference scenario, however they are good enough to be used by practitioners. Increasing the number of references has the desired effect of increasing the number of reads that are successfully mapped.[1] In the next section we explore the impact of this for variant detection.

[1] The results in Table 5.1 differ from the results in Paper III because the tool used to generate the multiple sequence alignment in Paper III applied the variants greedily to the first genomes without overlapping variants. For this reason the index size and the times are larger here, but also the number of reads mapped is higher.

5.3.2 Variation calling

Our experimental setup for variation calling consists of a hidden donor genome, out of which a set of sequencing reads is given as input to the variation calling prediction workflows. We considered the following approaches for variation calling:

- MSA_{chic}. This is our PanVC framework, using CHIC for read alignment.
- MSA_{base}. Here we implemented a PanVC version that does not rely on CHIC aligner, but instead, uses BWA to align the reads against each sequence in the pan-genomic reference.
- GRAPH. We considered a previous approach [86] that uses a DAG to represents the pan-genomic reference, and modified it to extract an ad hoc reference, so that we can use it within PanVC.
- GATK. As a baseline, we considered GATK workflow [4] that aligns the reads against a reference genome using BWA and analyses the resulting read pileup.

The first three methods use reference sets of 20, 50 and 100 genomes. The GATK baseline method is limited to use only one reference.

Our experiments focus on variation calling on complex regions with larger indels and/or densely located simpler variants, where significant improvements are still possible. The reason for that is that graph-based pan-genome indexing has been already thoroughly evaluated [86] for mapping accuracy on human genome data. From those results one can infer that on areas with isolated short indels and SNVs, a regular single-reference based indexing approach with a highly engineered alignment algorithm might be already sufficient.

Therefore, we based our experimental setup on the analysis of highly-polymorphic regions of the human genome [45, 53] that was created in a previous study [86]. This test setup consists of variation-rich regions from 93 genotyped Finnish individuals (1000 genomes project, phase 1 data). The 93 diploid genomes gave us a multiple alignment of 186 strains plus the GRCh37 consensus reference. This multiple alignment is obtained by applying the variants greedily to the first genome without other overlapping variants. We chose variation-rich regions that had 10 SNVs within 200 bases or less. The total length of these regions was 2.2 MB. To produce the ground-truth data for our experimental setup, we generated 221559 100bp

single-end reads from each of the Finnish individuals giving an average coverage of $10x$.

All the evaluated methods output variation calling results that are projected with respect to the standard reference genome. Our hidden donor genome can also be represented as a set of variants with respect to the standard reference genome. This means that we can calculate the standard prediction success measures such as precision and recall. For this, we chose to define the prediction events per base, rather than per variant, to tolerate better invariances of variant locations as have been found to be critical in a recent study [98].

In addition to precision and recall, we also compute the unit cost edit distance of the true donor and the predicted donor. This is defined as the minimum amount of single base substitutions, insertions, or deletions required to convert the predicted donor into the true donor. Here the sequence content of the true donor is constructed by applying its set of variants to the standard reference and the sequence content of the predicted donor is constructed by applying the predicted variants to the standard reference. See Paper III for more details on the evaluation metrics.

As our experiments are on human data, where genomes are diploids, the heterozygous variants may overlap, which causes some changes to the evaluation measures above. That is, when applying the variants to the reference, we omit variants that overlap already processed ones, and the result is thus a single sequence consisting of all compatible variants. We follow this approach also when computing the precision and recall measures to make the "per base" prediction events well-defined. The results are illustrated in Tables 5.2 and 5.3. Row GATK of Table 5.2 stands for the GATK workflow. Row MSA + GATK of Table 5.2 stands for the multiple sequence alignment -based pan-genome indexing scheme specified in the Methods section to produce the ad hoc reference. Row Graph +GATK of Table 5.2 is using the graph-based indexing of [86], where the ad hoc reference component is specified in the Methods section.

The columns in both tables refer to predictions after using GATK with the standard reference, and after using ad hoc references produced using different size sets of reference genomes in pan-genome indexing. The results are averages over 10 donors.

	Pan-genome reference size			
	1	20	50	100
GATK	74695.9	-	-	-
MSA_{base} + GATK	-	2885.5	1956.9	1204.7
MSA_{chic} + GATK	-	1349.3	1117.4	1099.3
Graph +GATK	-	3230.4	3336.8	2706.9

Table 5.2: Edit distance from the predicted donor sequence to the true donor. The average distance between the true donors and the reference is 95193.9.

Measure	GATK	20	50	100
SNV Precision	0.992161	0.997355	0.996913	0.996348
SNV Recall	0.904897	0.996721	0.997528	0.997554
Indel Precision	0.364853	0.994608	0.994906	0.994804
Indel Recall	0.0624981	0.961927	0.973369	0.982674

Table 5.3: Precision and recall of our method MSA_{base} compared to GATK.

Measure	GATK	20	50	100
SNV Precision	0.992161	0.998585	0.998863	0.998773
SNV Recall	0.904897	0.997098	0.998695	0.999072
Indel Precision	0.364853	0.996514	0.99731	0.997778
Indel Recall	0.0624981	0.982659	0.985723	0.985958

Table 5.4: Precision and recall of our method MSA_{chic} compared to GATK.

Chapter 6

Diploid Alignments

Genomes are frequently abstracted as single linear sequences. A basic problem in comparative genomics is to compute the edit distance between genomes. That is, to find the minimum number of editions, insertions or deletions needed to transform one genome into another. To take into account more convoluted biological processes, different distances have been proposed, such as the inversion distance [42], the inversion-indel distance [97], and DCJ-indel distance [99, 9], to name a few.

However, human genomes are diploid, as we inherit two independent versions of our genome from each parent. Unfortunately it is quite difficult to obtain the exact content of each of those two sequences. In practice, the genotyping process identifies variants that are present in an individual's genome, but it does not necessarily identify to which of the two sequences each variant belongs. The process to assign each variant to each of those sequences is known as *haplotype phasing*.

In this chapter we first review the proposal of Paper IV that models diploid genomes as pair-wise alignments. This representation leads to a generalization of edit distance (and related concepts such as sequence alignment and sequence similarity) that accounts for unknown haplotype phases. Furthermore, we review the proposal of Paper V that extends this representation to model the haplotype phasing problem.

6.1 Sequence alignment and recombinations

Recall that a *pair-wise alignment* (or simply an *alignment*, when it is clear from the context) of sequences A and B is a pair of sequences $(\mathsf{S}^A, \mathsf{S}^B)$ such that S^A is a supersequence of A, S^B is a supersequence of B, $|\mathsf{S}^A| = |\mathsf{S}^B| = n$ is the length of the alignment, and all positions which are not part of the

subsequence A (respectively B) in S^A (respectively S^B), contain the *gap* symbol $'-'$, which is not present in the original sequences.

Given a similarity function $\mathscr{C}(a,b)$ that assets the similarity between two characters, the similarity of a pair-wise alignment is defined as

$$\mathscr{S}(\mathsf{S}^A,\mathsf{S}^B)=\sum_{i=1}^{n}\mathscr{C}(\mathsf{S}^A[i],\mathsf{S}^B[i]).$$

The similarity of two sequences is then defined as

$$\mathscr{S}(A,B)=max\{\sum_{1=1}^{n}\mathscr{C}(\mathsf{S}^A[i],\mathsf{S}^B[i]):(\mathsf{S}^A,\mathsf{S}^B)\text{ is an alignment of }A\text{ and }B\}.$$

An alignment that achieves that value is called an *optimal alignment*. It is possible to use a *cost function* instead of a similarity function, in which case the maximum is replaced by a minimum, and the measure is a *distance* between A and B. In particular, the unit cost function is defined as 0 when the arguments are two identical characters, and 1 otherwise. When unit cost is used the result is the *edit distance* between A and B, which stands for the minimum number of insertions, deletions or substitutions to transform one sequence into the other.

We say that $(\mathsf{S}^{A'},\mathsf{S}^{B'})$ is a *recombination* of an alignment $(\mathsf{S}^A,\mathsf{S}^B)$ if both alignments have the same length n and there exists a binary string P (for *phase*) such that $\mathsf{S}^{A'}[i]=\mathsf{S}^A[i]$ and $\mathsf{S}^{B'}[i]=\mathsf{S}^B[i]$ if $P[i]=0$, and $\mathsf{S}^{A'}[i]=\mathsf{S}^B[i]$ and $\mathsf{S}^{B'}[i]=\mathsf{S}^A[i]$ if $P[i]=1$. We say that the characters are *swapped* in the positions in which $P[i]=1$. We denote this *recombination relation* by $(\mathsf{S}^{A'},\mathsf{S}^{B'})\Re(\mathsf{S}^A,\mathsf{S}^B)$.

6.2 Diploid to diploid similarity

6.2.1 A general model

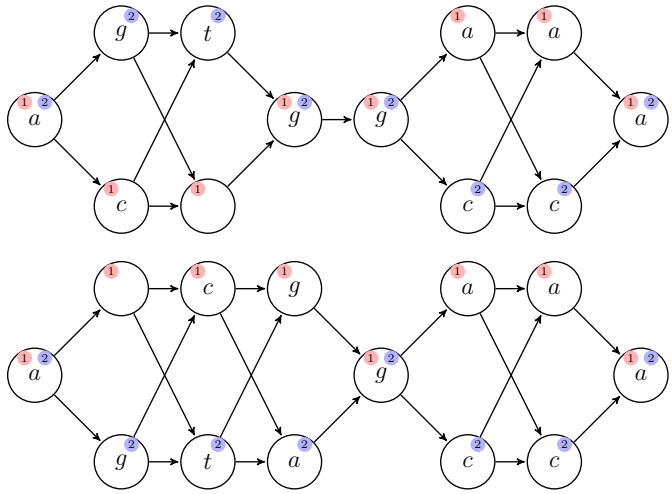
Given two pair-wise alignments $(\mathsf{S}^A,\mathsf{S}^B)$ and $(\mathsf{S}^X,\mathsf{S}^Y)$ that represent two diploid genomes, we define the *diploid to diploid similarity* as the sum of the optimal similarity scores by components, given by the best possible recombination of both diploids. More formally:

$$S_{d-d}((\mathsf{S}^A,\mathsf{S}^B),(\mathsf{S}^X,\mathsf{S}^Y))=\max\{\mathscr{S}(A',X')+\mathscr{S}(B',Y'):\\(\mathsf{S}^{A'},\mathsf{S}^{B'})\Re(\mathsf{S}^A,\mathsf{S}^B)\wedge(\mathsf{S}^{X'},\mathsf{S}^{Y'})\Re(\mathsf{S}^X,\mathsf{S}^Y))\}$$

Now we propose an alternative formulation of the problem that can be seen as an extension of the DAG path-alignment considered for progressive

S^A :	a	g	t	g	g	a	a	a
S^B :	a	c	-	g	g	c	c	a
$S^{A'}$:	a	c	-	g	g	a	a	a
$S^{B'}$:	a	g	t	g	g	c	c	a
S^X :	a	-	c	g	g	a	a	a
S^Y :	a	g	t	a	g	c	c	a

(a)



(b)

Figure 6.1: (a) Pair-wise alignment (S^A, S^B) for sequences $A = agtggaaa$ and $B = acggcca$. A recombination $S^{A'}$ and $S^{B'}$ is obtained by interchanging the shaded areas. The corresponding bitvector is $I = 01100000$. A pair-wise alignment (S^X, S^Y) is shown for sequences $X = acggaaa$ and $Y = agtagcca$. The diploid to diploid distance between (S^A, S^B) and (S^X, S^Y) is $0+1 = 1$, and is obtained using the recombination $(S^{A'}, S^{B'})$. The plain unit cost edit distance between (A, B) and (X, Y) is $2 + 3 = 5$. The plain unit cost edit distance between (A, B) and (Y, X) is $3+2 = 5$. The synchronized diploid to diploid distance between (S^A, S^B) and (S^X, S^Y) is 3. (b) DAG representation for the same input. From each DAG two covering paths are extracted, meaning that every node is visited by at least one path. The paths corresponding to the optimal solution are labeled with numbers 1 and 2 in each graph.

multiple sequence alignment [58, 63]. The main difference is that in our case the input is a pair of pair-wise alignments instead of multiple sequence alignments.

First we consider a labeled DAG representation of a diploid alignment that can be constructed by levels. In the 0th level of the DAG there is a single source node s_0, and then we create levels 1 to n adding edges from every node in level i towards every node in level $i+1$ as follows. If $\mathsf{S}^A[i] = \mathsf{S}^B[i]$ we put a single node in level i, and if $\mathsf{S}^A[i] \neq \mathsf{S}^B[i]$ we put two nodes in level i. Figure 6.1 shows an example.

Let G^1 and G^2 be two labeled DAGs each representing a pair-wise alignment. The **covering alignment problem** is to find two paths A and B that cover all nodes of G^1, and two paths X and Y that cover all nodes of G^2, such that $\mathscr{S}(A,X) + \mathscr{S}(B,Y)$ is maximum over all path-covers of G^1 and G^2. It is easy to see that this is equivalent to the diploid to diploid alignment problem.

In Paper IV we did not provide any algorithms nor complexity bounds for this problem, although the brute force solution that considers all possible recombinations yields to an exponential time algorithm. Very recent work has proved that this problem is NP-Hard [82].

In the following sections we explore alternative formulations that have polynomial time solutions.

6.2.2 Diploid to pair of haploids similarity

Given a pair-wise alignment $(\mathsf{S}^A, \mathsf{S}^B)$ and two sequences X and Y, the *diploid to pair of haploids similarity* is defined as the sum of the optimal similarity scores by components given by the best possible recombination of the diploid genome. More formally:

$$S_{d-hh}((\mathsf{S}^A,\mathsf{S}^B),(X,Y)) = \max\{\mathscr{S}(A',X) + \mathscr{S}(B',Y) : (\mathsf{S}^{A'},\mathsf{S}^{B'})\Re(\mathsf{S}^A,\mathsf{S}^B)\}$$

To compute the diploid to pair of haploids distance, we proceed as follows. We store values $V_{i,j,k}$ for $1 \leq i \leq |X|$, $1 \leq j \leq |Y|$, and $1 \leq k \leq n$, such that $V_{i,j,k}$ stands for the diploid to pair of haploids distance between the alignment $(\mathsf{S}^A_{1..k}, \mathsf{S}^B_{1..k})$ and the sequences $X_{1..i}$, $Y_{1..j}$. For the sake of simplicity we first show the recurrence when $\mathsf{S}^A[k] = \mathsf{S}^B[k] = c \in \Sigma$:

$$
\begin{array}{rllllll}
V_{i,j,k} = min\{ & \mathscr{C}(X_i, c) & + & \mathscr{C}(Y_j, c) & + & V_{i-1,j-1,k-1} & , \quad (6.1.1) \\
& \mathscr{C}('-', c) & + & \mathscr{C}(Y_j, c) & + & V_{i,j-1,k-1} & , \quad (6.1.2) \\
& \mathscr{C}(X_i, c) & + & \mathscr{C}('-', c) & + & V_{i-1,j,k-1} & , \quad (6.1.3) \\
& \mathscr{C}('-', c) & + & \mathscr{C}('-', c) & + & V_{i,j,k-1} & , \quad (6.1.4) \\
& \mathscr{C}(X_i, '-') & & & + & V_{i-1,j,k} & , \quad (6.1.5) \\
& & & \mathscr{C}(Y_j, '-') & + & V_{i,j-1,k} & \} \quad (6.1.6)
\end{array}
$$

Expression (6.1.1) stands for a match or substitution; expressions (6.1.2) to (6.1.4) stand for insertions either into X, Y, or both; and expressions (6.1.5) and (6.1.6) stand for deletion from X or Y. We do not include the case for simultaneous deletion from X and Y because it can be obtained by first deleting a character from X, and then from Y.

When $a = \mathsf{S}^A[k] \neq \mathsf{S}^B[k] = b$ we have to consider symmetric cases:

$$
\begin{array}{rllllll}
V_{i,j,k} = min\{ & \mathscr{C}(X_i, a) & + & \mathscr{C}(Y_j, b) & + & V_{i-1,j-1,k-1} & , \quad (6.2.1) \\
& \mathscr{C}(X_i, b) & + & \mathscr{C}(Y_j, a) & + & V_{i-1,j-1,k-1} & , \quad (6.2.2) \\
& \mathscr{C}('-', a) & + & \mathscr{C}(Y_j, b) & + & V_{i,j-1,k-1} & , \quad (6.2.3) \\
& \mathscr{C}(X_i, a) & + & \mathscr{C}('-', b) & + & V_{i-1,j,k-1} & , \quad (6.2.4) \\
& \mathscr{C}('-', a) & + & \mathscr{C}('-', b) & + & V_{i,j,k-1} & , \quad (6.2.5) \\
& \mathscr{C}('-', b) & + & \mathscr{C}(Y_j, a) & + & V_{i,j-1,k-1} & , \quad (6.2.6) \\
& \mathscr{C}(X_i, b) & + & \mathscr{C}('-', a) & + & V_{i-1,j,k-1} & , \quad (6.2.7) \\
& \mathscr{C}('-', b) & + & \mathscr{C}('-', a) & + & V_{i,j,k-1} & , \quad (6.2.8) \\
& \mathscr{C}(X_i, '-') & & & + & V_{i-1,j,k} & , \quad (6.2.9) \\
& & & \mathscr{C}(Y_j, '-') & + & V_{i,j-1,k} & \} \quad (6.2.10)
\end{array}
$$

Now we have twice as many expressions for matches/substitutions (6.2.1 and 6.2.2), twice as many conditions for insertions (6.2.3 to 6.2.8), and the same number of conditions for deletions (6.2.9 and 6.2.10). When $a = b$ those duplicated equations become redundant and we recover the previous formulation. Also it would be possible to formulate simplified recursions when there is a gap in the alignment (that is, either a or b equals $'-'$), however those are also particular cases of the general formulation, given that $\mathscr{C}('-', '-') = 0$.

To compute the S_{d-hh} distance using dynamic programming we fill a table with $V_{i,j,k}$ values in time $O(|X||Y|n)$. To find the actual alignment, we need to carry a bitvector P while doing the traceback process. P is initially empty. Every time we take a transition that decreases k we prepend a 0 (respectively a 1) to P, if we choose an expression from 6.2.1, 6.2.3, 6.2.4, or 6.2.5 (respectively 6.2.2, 6.2.6, 6.2.7, or 6.2.8). With this bitvector P we identify the recombination $(\mathsf{S}^{A'}, \mathsf{S}^{B'})$ that generated the optimal alignment, signaling with a 1 the positions where a recombination is done.

Because the model assumes that there is no prior knowledge about the correct phasing of the variants, the recombinations do not have any penalty. It is easy to modify the recurrences to account for a penalty for each recombination, should we apply the model under different assumptions.

6.2.3 Synchronized diploid to diploid alignment

In this section we present a different way to simplify the diploid to diploid alignment. First we will introduce some concepts. We say that $(\mathsf{S}^{\hat{A}}, \mathsf{S}^{\hat{B}})$ is a *recombined superalignment* of an alignment $(\mathsf{S}^{A}, \mathsf{S}^{B})$ if there exist sequences $\mathsf{S}^{A'}$ and $\mathsf{S}^{B'}$ that are obtained by removing gaps from the same positions in $\mathsf{S}^{\hat{A}}$ and $\mathsf{S}^{\hat{B}}$, and it holds that $(\mathsf{S}^{A'}, \mathsf{S}^{B'})\Re(\mathsf{S}^{A}, \mathsf{S}^{B})$. We denote this relation by $(\mathsf{S}^{\hat{A}}, \mathsf{S}^{\hat{B}})\Re_S(\mathsf{S}^{A}, \mathsf{S}^{B})$. In the following we resort to distance modeling to describe a speed-up.

Synchronized diploid to diploid distance:[1]

Given two pair-wise alignments $(\mathsf{S}^{X}, \mathsf{S}^{Y})$ and $(\mathsf{S}^{A}, \mathsf{S}^{B})$ of length n_1 and n_2 respectively, the synchronized diploid to diploid distance is defined as

$$
\begin{aligned}
S_{sync}((\mathsf{S}^{A}, \mathsf{S}^{B}), (\mathsf{S}^{X}, \mathsf{S}^{Y})) = \min\{ & \mathscr{S}(\mathsf{S}^{\hat{A}}, \mathsf{S}^{\hat{X}}) + \mathscr{S}(\mathsf{S}^{\hat{B}}, \mathsf{S}^{\hat{Y}}) \quad | \\
& \forall l \in \{1, \ldots, |\mathsf{S}^{\hat{A}}|\} \quad \exists \quad w, z \quad \text{such that:} \\
& (\mathsf{S}^{\hat{A}}_{1..l}, \mathsf{S}^{\hat{B}}_{1..l})\Re_S(\mathsf{S}^{A}_{1..w}, \mathsf{S}^{B}_{1..w}) \wedge (\mathsf{S}^{\hat{X}}_{1..l}, \mathsf{S}^{\hat{Y}}_{1..l})\Re_S(\mathsf{S}^{X}_{1..z}, \mathsf{S}^{Y}_{1..z}))\}
\end{aligned}
$$

6.2.3.1 A quadratic time algorithm

We compute values $D_{w,z}$ for $w \in \{1, \ldots, n_1\}$, $z \in \{1, \ldots n_2\}$, where $D_{w,z}$ stands for the synchronized diploid to diploid distance between alignments $(\mathsf{S}^{X}_{1..w}, \mathsf{S}^{Y}_{1..w})$ and $(\mathsf{S}^{A}_{1..z}, \mathsf{S}^{B}_{1..z})$. The recurrence is a direct generalization of the standard edit distance computation, with the additional possibility of recombination.

$$
\begin{aligned}
D_{w,z} = min\{ \; & D_{w-1,z-1} + \mathscr{C}(\mathsf{S}^{X}_{w}, \mathsf{S}^{A}_{z}) + \mathscr{C}(\mathsf{S}^{Y}_{w}, \mathsf{S}^{B}_{z}) \quad , & (6.3.1) \\
& D_{w-1,z-1} + \mathscr{C}(\mathsf{S}^{X}_{w}, \mathsf{S}^{B}_{z}) + \mathscr{C}(\mathsf{S}^{Y}_{w}, \mathsf{S}^{A}_{z}) \quad , & (6.3.2) \\
& D_{w,z-1} + \mathscr{C}('-', \mathsf{S}^{A}_{z}) + \mathscr{C}('-', \mathsf{S}^{B}_{z}) \quad , & (6.3.3) \\
& D_{w-1,z} + \mathscr{C}(\mathsf{S}^{X}_{i}, '-') + \mathscr{C}(\mathsf{S}^{Y}_{j}, '-') \quad \} & (6.3.4)
\end{aligned}
$$

[1] In Paper IV the formalization of synchronized diploid to diploid alignment and the algorithms to compute it are built using the concept of guiding function. Here we formalize it in a different way, but the resulting algorithms are equivalent.

		Diploid to Pair of Haploids		Synchronized Diploid To Diploid			
		Cubic Algorithm		Matrix		Diagonals	
Length	Mutations	Distance	Time	Distance	Time	Distance	Time
1000	21	21	19.190	21	0.040	21	0.001
2000	47	43	154.030	47	0.110	47	0.010
4000	85	79	1219.060	85	0.650	85	0.020
8000	174	-	-	172	1.790	172	0.070
10000	218	-	-	216	2.820	216	0.180
20000	408	-	-	403	11.080	403	0.580
40000	777	-	-	750	44.290	750	1.170
80000	1578	-	-	1531	177.200	1531	4.630
100000	1994	-	-	1935	276.960	1935	11.500

Table 6.1: The distance between pairs of diploids computed by our three algorithms, and the time (in seconds) used for the computation. The cubic algorithm is our implementation of the diploid to pair of haploids algorithm, Synchronized - Matrix is the straightforward implementation of the synchronized diploid to diploid distance, and Synchronized - Diagonals is the $O(ND)$ time implementation of the same algorithm using the doubling diagonals technique.

			Synchronized - Diagonals		Levenshtein	
Length	Variations	Mutations	Distance	Time	Distance	Time
10000	100	7	7	0.006	52	0.010
20000	227	18	18	0.020	139	0.052
30000	338	33	33	0.064	241	0.138
40000	434	36	36	0.096	272	0.214
50000	538	46	46	0.140	367	0.362
60000	637	50	50	0.230	448	0.528
70000	742	67	67	0.274	529	0.850
80000	839	73	73	0.370	562	1.066
90000	960	91	91	0.550	697	1.570
100000	1089	109	109	0.688	762	2.198
1000000	10127	990	990	58.978	7676	376.320

Table 6.2: A number of variations were applied to the reference genome, and blocks of size 200 were recombined freely. After that, random mutations were introduced with probability 0.001. We show the distances and times (in seconds) obtained by our synchronized diploid to diploid distance algorithm and by the Levenshtein distance.

6.2.3.2 An $O(ND)$ time algorithm for unit costs

The same technique [93, 69] used for the computation of the Levenshtein distance of strings to achieve $O(ND)$ running time, where D is the final distance can be applied here. The key observation is:

Lemma 6.1 *Let (S^A, S^B) and (S^X, S^Y) be two alignments, and let $D_{w,z}$ be the synchronized diploid to distance costs computed as before. Then $\forall 0 \leq w \leq n_1, 0 \leq z \leq n_2$ it holds $D_{w,z} > D_{w-1,z}$ and $D_{w,z} > D_{w,z-1}$.*

Proof. It is enough to observe the second and third elements in expressions 6.3.3 and 6.3.4. The only situation that could contradict the lemma is if there is an i such that $S_i^A = S_i^B =' -'$, or analogously, a j such that $S_j^X = S_j^Y =' -'$. As such a situation does not add any information to an alignment we can easily remove all such positions before computing the distance without altering either the alignments or the resulting distance. □

Assume for simplicity of exposition that $|n_1| = |n_2| = n$ and we store values $D_{w,z}$ in a table. We also assume all the edit operations have unitary cost. We want to test if the distance between $(\mathsf{S}^A, \mathsf{S}^B)$ and $(\mathsf{S}^X, \mathsf{S}^Y)$ is smaller than a threshold t or not. That is, we want to test if $D_{N,N} \leq t$ for some threshold t. Diagonals $j - i \in \{-t/2, -t/2 + 1, \ldots, 0, 1, \ldots, t/2\}$ are sufficient to consider in the computation, as any path using a diagonal outside this zone must use more than t operations that have cost 1, leading to an alignment with a cost higher than t. Starting with $t = 1$ and doubling this value until $D_{N,N}$ does not decrease any more gives the optimal answer, and the final area where the computation is done is of order $O(ND)$; the previous zone sizes form a geometric series, so the computation done inside them is of the same order as the computation inside the final zone.

6.2.4 In practice

In Paper IV we presented an implementation of the algorithms of Section 6.2.2 and Section 6.2.3. They were implemented using unit cost, so they measure distance between alignments. Here we reproduce the main results. In Table 6.1 the three algorithms are compared using inputs of size up to 10^4, and in Table 6.2 we compared the $O(ND)$-time algorithm against an implementation of the Levenshtein distance as a baseline in inputs of size up to 10^6.

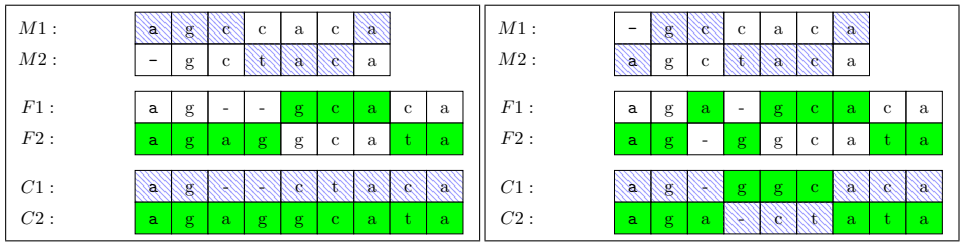


Figure 6.2: On the left we show how the pair-wise alignments would be if we knew the correct phasing of the three individuals. On the right, the same individuals are presented, but the haplotype phasing is not known a priori. We assume a similarity function that scores 1 for equal characters, and -1 for indels and mismatches. The colored recombinations show the sequences M_1' (hatched blue), F_1' (solid green) from the recombination that gives the optimal alignment. The haplotyping similarity of the trio is $S(\texttt{AGCTACA}, \texttt{AGCTACA}) + S(\texttt{AGAGGCATA}, \texttt{AGAGGCATA}) = 7+9 = 16$. The binary strings associated to the recombinations are $P_M = \texttt{1001110}$, $P_F = \texttt{110100011}$ and $P_C = \texttt{000111000}$. Note that the latter corresponds to the predicted phase for the child genome. It is also important to note that none of the binary strings signal evolutionary recombinations, as they account for phasing errors in the input data.

6.3 Haplotyping through diploid alignment

All the previous models are tools to measure similarity between genomes. Those are modeled either as a diploid or as pair of haploids, and the models account for different extents of recombination. Their aim is to compare individuals where heterozygous and homozygous variations are known, but there is no knowledge about the correct phasing of the variations.

In this section we extend one of them, the diploid to pair of haploids, to propose a formulation for the haplotype phasing problem in a setting where the parents' genotypes are also known.

6.3.1 The similarity model

Let $(\mathsf{S}^{M_1}, \mathsf{S}^{M_2})$,$(\mathsf{S}^{F_1}, \mathsf{S}^{F_2})$ and $(\mathsf{S}^{C_1}, \mathsf{S}^{C_2})$ be three pair-wise alignments, of length L_M, L_F and L_C respectively. Those represent the diploid sequences of the mother, father and child, and we call the three of them a *mother-father-child trio* (m-f-c trio).

We define the *haplotyping similarity of an m-f-c trio*, H(m-f-c), as the maximum pair-wise similarity between one of the sequences of the mother and one of the sequences of the child, plus the pair-wise similarity between the other child sequence and one of the father sequences, assuming that haplotype phasing has not been performed. This means that we need to al-

low free recombination in each pair-wise alignment to let the model discover the real phase. More formally:

$$\mathsf{H}(\text{m-f-c}) = \max\{S(M_1', C_1') + S(F_1', C_2') : (\mathsf{S}^{M_1'}, \mathsf{S}^{M_2'})\Re(\mathsf{S}^{M_1}, \mathsf{S}^{M_2}) \\ \wedge (\mathsf{S}^{F_1'}, \mathsf{S}^{F_2'})\Re(\mathsf{S}^{F_1}, \mathsf{S}^{F_2}) \wedge (\mathsf{S}^{C_1'}, \mathsf{S}^{C_2'})\Re(\mathsf{S}^{C_1}, \mathsf{S}^{C_2})\}$$

Figure 6.2 shows an example optimal alignment of a m-f-c trio.

6.3.2 Dynamic programming algorithm

In this section we present our algorithm to compute the m-f-c similarity. We propose a dynamic programming formulation that computes values $\mathsf{H}_{i,j,k,m,f,c}$ with $i \in \{1,\ldots,L_M\}$, $j \in \{1,\ldots,L_F\}$, $k \in \{1,\ldots,L_C\}$, $m \in \{1,2\}$, $f \in \{1,2\}$ and $c \in \{1,2\}$. The value in $\mathsf{H}_{i,j,k,m,f,c}$ stands for the similarity score between $(\mathsf{S}^{M_1}[1..i], \mathsf{S}^{M_2}[1..i])$, $(\mathsf{S}^{F_1}[1..j], \mathsf{S}^{F_2}[1..j])$, and $(\mathsf{S}^{C_1}[1..k], \mathsf{S}^{C_2}[1..k])$, with the additional constraint that the last character of the mother alignment is swapped if and only if $m = 1$, the last character of the father alignment is swapped if and only if $f = 1$ and the last character of the child alignment is swapped if and only if $c = 1$.

We first consider the particular case when the input alignments contain no gaps. That is, $\mathsf{S}^{C_1} = C1$, $\mathsf{S}^{C_2} = C_2$, $\mathsf{S}^{F_1} = F_1$, etc. It is possible to compute those values recursively as follows:

$$\mathsf{H}_{i,j,k,m,f,c} = \max \begin{cases} \mathsf{H}_{i-1,j,k,*,f,c} & + & s('-', M_m[i]) & & & & \text{if } i > 1 \\ \mathsf{H}_{i-1,j,k-1,*,f,*} & + & s(C_c[k], M_m[i]) & + & s(C_{c\oplus1}[k], '-') & & \text{if } i,k > 1 \\ \mathsf{H}_{i,j-1,k,m,*,c} & & & + & s('-', F_f[j]) & & \text{if } j > 1 \\ \mathsf{H}_{i,j-1,k-1,m,*,*} & + & s(C_c[k], '-') & + & s(C_{c\oplus1}[k], F_f[j]) & & \text{if } j,k > 1 \\ \mathsf{H}_{i-1,j-1,k-1,*,*,*} & + & s(C_c[k], M_m[i]) & + & s(C_{c\oplus1}[k], F_f[j]) & & \text{if } i,j,k > 1 \end{cases}$$

Where $\mathsf{H}_{i,j,k,*,*,*} = \max_{\{m,f,c\}\in\{1,2\}^3}\{\mathsf{H}_{i,j,k,m,f,c}\}$ considers all the 8 valid subproblems where the previous last characters could have been swapped or not, and similarly when only one or two $*$ symbols are present. With $c \oplus 1$ we mean 2 if $c = 1$ and 1 otherwise.

The first and third cases correspond to the scenarios where the last character of the mother (respectively, of the father) is not aligned with any character of the child, and therefore a gap symbol is inserted. The second and fourth cases corresponds to the scenarios where one of the last characters of the child is aligned with one of the last characters of the mother (respectively, of the father), and the other character of the child is not aligned, therefore, a gap is inserted. The fifth case is the scenario where the last character of one of the child sequences is aligned with one of the last sequences of the mother, and the last character of the other

child sequence is aligned with the last character of one of the sequences of the father. In Paper IV we presented the pseudo code detailing how to handle the cases where there is a gap in the input. Here we summarize it in Algorithm 1.

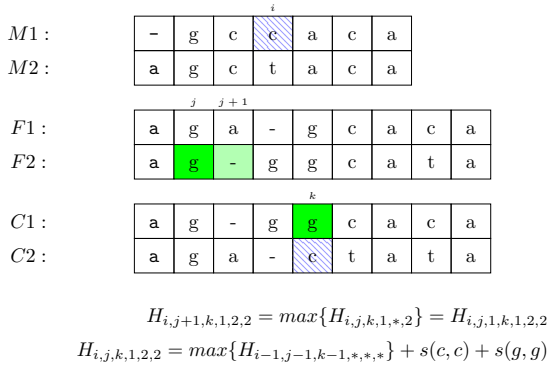


Figure 6.3: Example showing two steps of the dynamic programming algorithm. First for the computation of $H_{i,j+1,k,1,2,2}$ we highlight the characters that need to be considered. As the $j+1$ character of the father sequence that is being considered is a gap, the recursion returns the previous value of j, keeping all the parameters constant, except for the sequence of the father that can be considered (line 10 of Algorithm 1.) For the computation of $H_{i,j,k,1,2,2}$ the previous values indicated by lines 6,7,11,12, and 14 need to be considered. Those correspond to all possible combinations of alignments between the highlighted characters (allowing some of them to be ignored, but not all of them).

6.3.3 From m-f-c similarity to haplotype phasing

Once all the values $\mathsf{H}_{i,j,k,m,f,c}$ are computed, it is enough to trace back the path that originated the optimal score to obtain the optimal alignment. We notice that if we collect the three last indexes m, f and c from the path we will obtain the recombination binary strings for the mother, father, and child. In particular, the latter gives us the phasing of the child diploid that maximizes the similarity of the m-f-c trio. In the example of Figure 6.2 the binary strings are P_M, P_F and P_C; the last one being the phasing of the child diploid.

6.3.4 In practice

A straightforward implementation would require $O(n^3)$ time and $O(n^3)$ memory as well, because we need to traceback the table to retrieve the

Algorithm 1 Haplotyping similarity of an m-f-c trio. The algorithm corresponds to the dynamic programming implementation of the recurrence presented in Section 6.3.2, modified to handle the gaps in the input sequences properly.

```
1:  function HaploidSimilarity(M_1, M_2, F_1, F_2, C_1, C_2)
2:
3:      H[0,0,0,*,*,*] ← 0
4:      SetGlobal(H,M1,M2,F1,F2,C1,C2)
5:      for i ← 0 to L_M do
6:          for j ← 0 to L_M do
7:              for k ← 0 to L_M do
8:                  for m ← 1 to 2 do
9:                      for f ← 1 to 2 do
10:                         for c ← 1 to 2 do
11:                             H[i,j,k,m,f,c] ← HValue(i,k,k,m,f,c)
12:     return max H[L_m, L_f, L_c, *, *, *]
```

```
1:  function HValue(i, j, k, m, f, c)
2:      value ← −inf
3:      if i > 0 then
4:          if M_m[i] =' −' then
5:              return max{H[i−1, j, k, *, f, c]}
6:          value ← max{value, H[i−1, j, k, *, f, c] + s('−', M_m[i])}
7:          value ← max{value, H[i−1, j, k−1, *, f, *] + s(C_c[k], M_m[i]) +
    s(C_{c⊕1}[k], '−')}
8:      if j > 1 then
9:          if F_f[i] =' −' then
10:             return max{H[i, j−1, k, m, *, c]}
11:         value ← max{value, H[i, j−1, k, m, *, c] + s('−', F_f[j])}
12:         value ← max{value, H[i, j−1, k−1, m, *, c] + s(C_c[k], '−') +
    s(C_{c⊕1}[k], F_f[j])}
13:     if j > 1 & > 1 then
14:         value ← max{value, H[i−1, j−1, k−1, m, *, c] +
    s(C_c[k], M_m[i]) + s(C_{c⊕1}[k], F_f[j])}
15:     return value
```

phasing. In Paper V we decided to implement the checkpoint method [77], a flexible variant of Hirschberg's algorithm [43] that allows our algorithm to run in $O(n^3)$ time using $O(n^2)$ memory.

The experiments presented in Paper V used simulated data for m-f-c trios. The mother genome started from a 1000bp long sequence randomly extracted from human chromosome 21. Different types of variations were inserted at random positions, and then we simulated a recombination of that pair-wise alignment to obtain the child chromosome that is inherited from the mother. For the recombination process we choose recombination points at random. We did this analogously to simulate the father, and the chromosome that the child inherits from the father.

The variations planted on the parents were as follow: *point mutations* consisted of SNPs and single nucleotide deletions. Then we introduced *long indels* (larger than 50 base pairs). In the case of insertions, those consisted of random base pairs. We also inserted *short tandem repeats* [95] consisting of insertions of lengths between 20 and 60 repeating a sequence between 2 and 6 base pairs. After the recombination has been simulated to generate the child sequences, we introduced *de novo point mutations*.

In addition to all the previous parameters, we also introduced *random errors* over all the sequences at the end, to take into account errors during the sequencing of the sequences and during variant calling to discover the genotype patterns that constitute the inputs of our algorithm.

We studied the quality of our phasing algorithm by measuring how sensible it was with respect to each of the parameters. We measured the percentage of positions that were incorrectly phased by our method, For each different type of variation, the ratio ranged between 0 and 15%. We considered the following three scenarios with regard to the error ratios: a very optimistic one, were the genotyping was done without errors (0%), a moderated scenario where error ratio is 5%, and a pessimistic scenario where the error ratio is 15%. The time used for haplotyping each simulated trio was less than a hour. The results are shown in Figure 6.4.

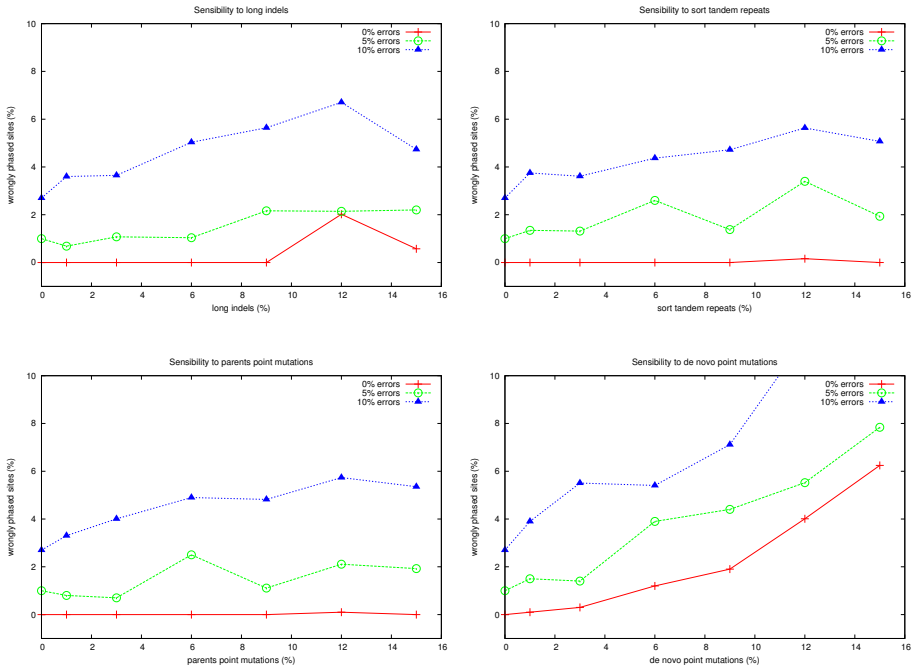


Figure 6.4: The percentage of incorrectly phased positions versus the mutation ratio. For every graphic we include three levels of random noise. The two first plots (above) show the behavior against long indels and short tandem repeats. The third plot (below left) shows the behavior against point mutations in the parents and the fourth plot (below right) shows de novo variants.

Chapter 7

Read Pruning as Interval Scheduling

The work in this chapter is motivated by a read-based approach to haplotype phasing called the *minimum error correction problem* (MEC) [62]. This formulation analyzes a set of NGS reads aligned to a reference genome and it aims for a consistent separation of them into two sets, each one associated with one of the haplotypes. Naturally, such separation does not always exist, because the sequencing and alignment processes are vulnerable to errors. Therefore, the MEC formulation looks for the minimum number of corrections to the reads to achieve the desired separation.

Although the MEC problem is known to be NP-hard [15], there is a recent tool, released under the name of WhatsHap, that implements a practical algorithm [76] to solve MEC in $O(2^{k-1}N)$ time. This is particularly useful because the running time is exponential only in k, the maximum number of reads covering any position of the genome, while the dependency on N,the length of the genome, is only linear and there is no dependence on the read length.

Unfortunately this time complexity makes the algorithm feasible only for small values of k. In practice, WhatsHap addresses this issue by randomly pruning reads from positions with a too high coverage. This may lead to some positions having a too low coverage for accurate results. The ideal situation for MEC to achieve a meaningful solution would be to have a high number of reads evenly distributed through the genome.

In this chapter we present the contribution of Paper VI : we formulate the problem of read pruning such that the maximum read coverage is less than a given parameter k, while keeping the minimum coverage across all genomic positions as high as possible. We point out that this read pruning problem can be formulated as a job scheduling problem, however

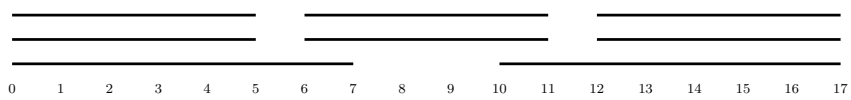


Figure 7.1: An instance of interval scheduling maximizing minimum coverage in which 8 intervals are given and $k = 2$ machines are available to execute them. Observe that in all solutions to this problem three disjoint intervals of length 5 need to be removed, leading to a solution that executes 5 jobs and the number of idle machines is never greater than 1. In the solutions to the classical interval scheduling with given machines problem, the two intervals of length 7 are removed both in the case when all intervals have the same profit, and in the case when the profit of an interval equals its length.

in a different fashion than the most commonly studied variants of this problem. We propose two exact solutions to this problem that run in time $O(n^2 \log k / \log n)$ and $O(nk \log k)$, respectively, where n is the number of reads. Additionally, we give a 2-approximation algorithm that runs in $O(n \log n)$ time.

7.1 Interval scheduling

In the interval scheduling problem the input is a set of jobs, such that each of them has a specific time interval in which it needs to be carried out by any of the available machines. In the simplest variant of this problem each machine is always available, it can process at most one job at a time and the jobs are processed without interruptions. The task is to find the minimum number of machines needed to process all the jobs [54]. It is possible to solve this problem in $O(n \log n)$ time [40].

A well-studied variant is *interval scheduling with given machines*, also known as the *k-track assignment problem* or *interval scheduling on identical machines*. In this setting the number of available machines is k and there is a profit associated with the execution of each job. The task here is to find a set of jobs that maximize the total profit. There are solutions to this variant that run in polynomial time [3, 8]. Other variants are known to be NP-hard: for example when each job can be executed only by a given subset of machines [3], or when each machine is available during a specified time interval [11]. We refer the reader to comprehensive surveys [54, 55] for further references.

Most previous work has focused on maximizing a profit obtained by executing jobs, or minimizing the resources used. On the other hand, our formulation can be seen as one that minimizes a notion of idleness.

7.2 Problem formulation

First we introduce some useful definitions. We say that an interval $[s_i, f_i)$ *covers* a point p if $p \in [s_i, f_i)$. Given a set of intervals $S = \{[s_i, f_i) \ : \ i \in \{1, \ldots, n\} \mid 0 \leq s_i < f_i \leq N\}$ and a point p we define the *coverage* of p as $\mathsf{cov}_S(p) = |\{[s_i, f_i) \in S | [s_i, f_i) \text{ covers } p\}|$. The *maximum coverage of* S is defined as $\mathsf{maxcov}(S) = max_{p \in [0,N)} \mathsf{cov}_S(p)$ and the *minimum coverage of* S as $\mathsf{mincov}(S) = min_{p \in [0,N)} \mathsf{cov}_S(p)$. Now we can formalize the problem:

Problem 7.1 (Paper VI)
Interval scheduling maximizing minimum coverage
INPUT. *A set* $S = \{[s_i, f_i) \ : \ i \in \{1, \ldots, n\} \mid 0 \leq s_i < f_i \leq N\}$ *of intervals and an integer* k.
TASK. *Find an* $S' \subseteq S$ *that maximizes* $\mathsf{mincov}(S')$, *subject to* $\mathsf{maxcov}(S') \leq k$ *and*

Figures 7.1 and 7.2 illustrate the problem. In the read pruning context S represents the set of n NGS reads, each one mapped to positions s_i to f_i, and N is the length of the genome.

7.3 Exact solution

In this section we show how to reduce the problem to a maximum flow problem. We first show that Problem 7.1 can be reduced to its decision version with a logarithmic slowdown. Then we reduce the decision version to a max-flow problem. Finally we show an improved running time by using an ad-hoc max-flow algorithm.

7.3.1 Reduction to the decision version

The decision version of Problem 7.1 is as follows:

Problem 7.2 (Paper VI)
Interval scheduling with bounded coverage
INPUT. *A set* $S = \{[s_i, f_i) \ : \ i \in \{1, \ldots, n\} \mid s_i < f_i\}$ *of intervals, and integers* k, t.

TASK. *Decide if there exists an* $S' \subseteq S$ *such that* $\mathsf{maxcov}(S') \leq k$ *and* $\mathsf{mincov}(S') \geq t$*, and if yes, output such* S'*.*

If we have an algorithm A that solves Problem 7.2 we can use it to solve Problem 7.1 using exponential search as follows. We first observe that if Problem 7.2 admits a solution for a given $t = t'$, it also admits a solution for all $t \in \{0, \ldots, t'\}$. We use the algorithm A for $t = 1$, $t = 2$, $t = 4$, ..., until we find a t^l where for $t^r := 2t^l$ the decision problem no longer have a solution. Then by binary searching on decision problem instances with $t \in [t^l..t^r]$ we obtain the minimum coverage $t = \texttt{OPT}$ of the optimal solution to the maximization problem. This leads us to the following result.

Lemma 7.1 *Given an algorithm that solves Problem 7.2 in time* $f(n)$*, it is possible to solve Problem 7.1 in time* $O(f(n) \log \texttt{OPT})$*, where* OPT *is the minimum coverage of an optimal solution to Problem 7.1.*

7.3.2 Reduction to max-flow

Now we reduce Problem 7.2 to a max-flow problem. Figure 7.2 shows an example. Let $s = \min_{i \in \{1,\ldots,n\}} s_i$ and $f = \max_{i \in \{1,\ldots,n\}} f_i$. We build a graph $G_{S,k,t}$, possibly having parallel arcs, using the set of vertices $V(G) = \{s-1, f+1\} \cup \{s_i, f_i \ : \ i \in \{1, \ldots, n\}\}$. The only source of the graph is the vertex $s-1$, and the only sink is $f+1$. For every vertex i in $V(G)$ other than $f+1$, we add the arc (i, j) to $E(G)$ where j is the successor of i in $V(G)$. We call these *backbone* arcs. The backbone arcs $(s-1, s)$ and $(f, f+1)$ receive capacity k, and the other backbone arcs have capacity $k - t$. Additionally, for every interval $[s_i, f_i) \in S$, we add the arc (s_i, f_i) with a capacity of 1 to $E(G)$. We call these *interval* arcs.

Theorem 7.1 shows that computing the max-flow on $G_{S,k,t}$ is equivalent to solving Problem 7.2.

Theorem 7.1 (Paper VI) *Problem 7.2 admits a solution on an instance* (S, k, t) *if and only if the max-flow in* $G_{S,k,t}$ *has value* k*, and this solution can be retrieved from any integral max-flow on* $G_{S,k,t}$*.*

The full proof can be found in Paper VI. The idea is to show that the flow passing through an interval arc is equivalent to the selection of that arc in the solution S'. The capacity $k - t$ imposed on the backbone arcs not incident to $s-1$ or to $f+1$ implies that for any $p \in [s, f)$ we have at most $k - t$ intervals not covering p, thus at least t intervals covering p.

Because $G_{S,k,t}$ has $O(n)$ vertices and arcs, it is possible to use a specialized max-flow algorithm [75] that leads to the following corollary.

Corollary 7.1 *Problem 7.2 is solvable in time $O(n^2/\log n)$ by solving the max-flow problem on $G_{S,k,t}$.*

Combining Corollary 7.1 and Lemma 7.1 we obtain the following result:

Corollary 7.2 *Problem 7.1 is solvable in time $O(n^2 \log \texttt{OPT}/\log n)$, where $\texttt{OPT}$, $\texttt{OPT} \leq k$, is the minimum coverage of an optimal solution.*

7.3.3 Improving the running time

Now we show an alternative solution using an ad-hoc max-flow algorithm that improves the running time when $k = o(n/\log n)$.

The Ford-Fulkerson algorithm is a textbook method to find max-flows (see e.g. [20, Section 26.2]). It works by finding an augmenting path in the residual network of the graph and then adjusting the flow network along the same path to increase the total flow. When there is no augmenting path left in the residual network, the flow found is a maximum. Assuming integral capacities, the flow increases at least by one unit on each augmentation step, thus the running time is $O(|E||\varphi^*|)$, where E is the set of arcs of the flow network and $|\varphi^*|$ is the value of the maximum flow.

Let us consider what occurs when Ford-Fulkerson is run on $G_{S,k,t}$, the resulting network from the reduction of Sec. 7.3.2. We observe that a flow of value $k - t$ can be sent through the backbone arcs from source to sink. That is, we can initialize the network with a flow of value $k - t$, and run Ford-Fulkerson from that initial feasible flow. We need at most $t \leq k$ augmentation steps, each requiring $O(n)$ time. Therefore we obtain the following result.

Theorem 7.2 *Problem 7.2 is solvable in time $O(nk)$.*

Again, combining this theorem and Lemma 7.1 we obtain the following result:

Corollary 7.3 *Problem 7.1 is solvable in time $O(nk \log \texttt{OPT})$, where $\texttt{OPT}$, $\texttt{OPT} \leq k$, is the minimum coverage of an optimal solution.*

7.4 An $O(n \log n)$-time approximation algorithm

In this section we propose an approximation algorithm that runs in time $O(n \log n)$. The approximation ratio is $\frac{k}{\lfloor k/2 \rfloor}$, which for k even is a 2-approximation algorithm. First we present some concepts that are used in this section.

We generalize the definition of minimum coverage to intervals as $\mathsf{mincov}_S([s_i, f_i)) = min_{p \in [s_i, f_i)} \mathsf{cov}_S(p)$. When an interval has minimum coverage smaller or equal to $\lfloor k/2 \rfloor$ we say that such an interval is *crucial*; otherwise, we call it *expendable*. The following result is the key idea behind the approximation algorithm.

Lemma 7.2 *Given an input (S, k) to the interval scheduling maximizing minimum coverage problem (Problem 7.1), and a point p, if $\mathsf{cov}_S(p) = k' > k$, there at least $k' - k$ intervals covering p that are expendable.*

The full proof can be found in Paper VI. The idea is to reason by contradiction: to claim that there are less than $k' - k$ expendable intervals covering p is equivalent to claim that there are more than k crucial intervals covering p. If we assume this, we can prove that at least one of those intervals cannot really fulfill the definition of a crucial interval.

To solve the problem, the algorithm maintains a set of candidate intervals S^* which is initialized as S. For every interval its maximum and minimum coverage is computed, detecting the crucial intervals, which will never be removed from S^*.

Then, the intervals' delimiters are traversed from left to right. Whenever a delimiter p is found to have coverage k' greater than k, $k' - k$ intervals covering p are removed from S^*. By Lemma 7.2, we know that among the k' intervals covering p, there are at least $k' - k$ intervals that are expendable, so we can delete those safely. After each deletion, the coverages are updated and if needed new intervals are marked as crucial. Because the algorithm never removes crucial intervals, (and the optimal solution to Problem 7.1 is bounded by k), the approximation ratio is $\rho = \frac{k}{\lfloor k/2 \rfloor}$.

In Paper VI we further describe the data structures for the algorithm to run in $O(n \log n)$ time. The main idea is to use a perfect binary search tree as a one-dimensional range search tree (as explained, e.g. in [22, Section 5.1]). The interval delimiters are stored in the leaves, and the tree is also annotated with information about the coverage of the delimiters and the intervals. All the above leads us to the following result.

Theorem 7.3 (Paper VI) *There is an $O(n \log n)$ time approximation algorithm to Problem 7.1 that finds a solution with minimum coverage at least $\frac{\lfloor k/2 \rfloor}{k}\texttt{OPT}$, where $\texttt{OPT}$ is the minimum coverage of an optimal solution.*

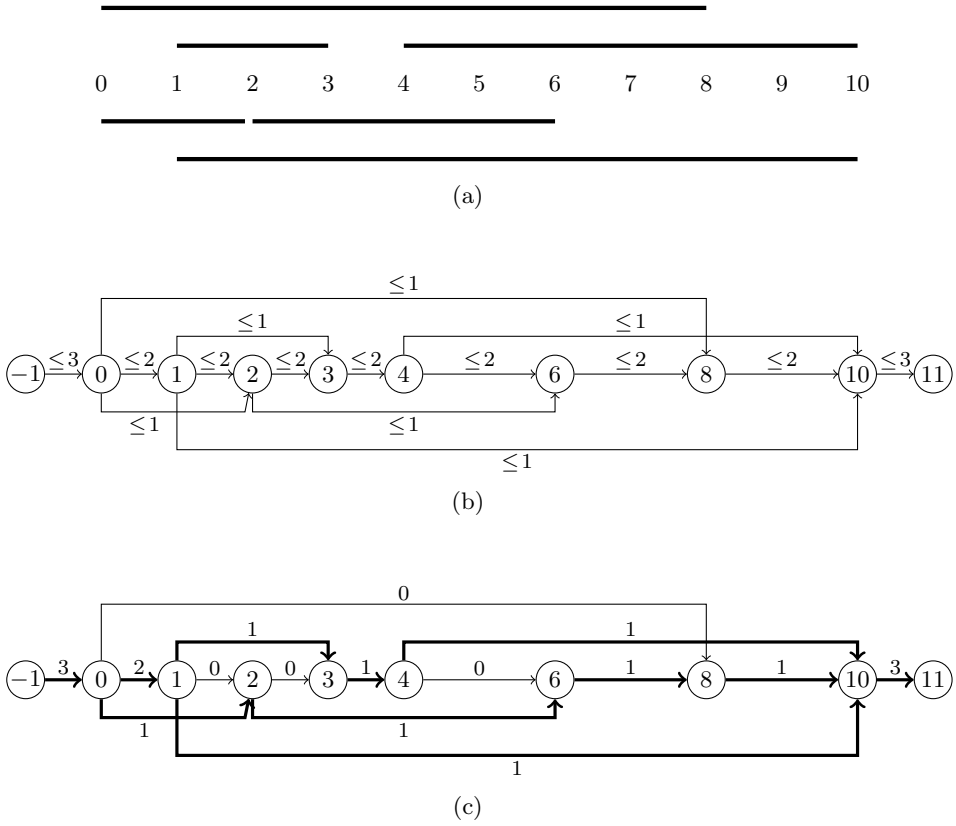


Figure 7.2: An example for the reduction of Problem 7.2 to a max-flow problem. In Figure 7.2(a) an instance (S, k, t) of Problem 7.2 consisting of 5 intervals, where we assume $k = 3$ and $t = 1$. One solution is obtained by removing the interval $[0, 8)$. In Figure 7.2(b) the graph $G_{S,k,t}$ whose arcs are labeled by capacities. In Figure 7.2(c) a maximum flow of value 3 in $G_{S,k,t}$; the arc labels now indicate their flow values. Notice that the arc $(0, 8)$ has a flow value of 0.

Chapter 8

Discussion

In this thesis we have presented different contributions to sequence analysis: improved algorithms, data structures and models for different kinds of sequence alignment, and different models for genotyping and haplotyping problems.

For each subject that we have studied, there are still many open questions leading to interesting routes for new research. Here I attempt to systematically discuss the contributions of this thesis with an emphasis of directions for future work

Chapter 3. We presented a theoretical analysis – supported by an experimental study – of the Relative Lempel-Ziv compression algorithm when it uses an artificial reference.

Although our analysis shed some light on this, there are still open questions. For instance, we still do not know how tight the analysis is. Another concern is that the sampling parameters prescribed by our analysis are given in terms of z, which depends on the behavior of LZ77, and if we could run LZ77 then we would not need RLZ in the first place. However, it is possible to estimate z more efficiently than by computing the LZ77 parse [80]. Lastly, although our experimental results qualitatively match the theoretical predictions, the constant within the big-O expressions are still unknown. Whether those can be estimated efficiently, or by a different analysis is an interesting way to further develop the results from this chapter.

Chapter 4. In this chapter we presented an improved, scalable version of the Hybrid Index [29]. This index is specially promising for genomic data because of how smoothly it handles approximate pattern matching in repetitive collections. Our modifications allowed this method to use different Lempel-Ziv factorizations to build the index, including RLZ. We introduced a parallel version of RLZ that permitted the building of an index

for a 2.4TB collection in about 12 hours.

An open challenge is to test CHICO in large collections lacking an obvious reference. This could be the case if we want to index a large collection of genomes from different species; or in information retrieval when indexing large web crawls. CHICO can be actually built for those collections using external memory to build the greedy LZ77 parsing, however this approach would damage the construction speed. The research question here would be how to integrate RLZ with an artificial reference into the hybrid index. A different direction is to explore the relaxed LZ77 parsing further, for instance, designing a specialized parser for the hybrid index.

Chapter 5. In this chapter we proposed a framework for variant calling using a pan-genomic reference, which was represented as a multiple sequence alignment. We adapted the index of Chapter 4 to develop a read aligner for pan-genomic references.

Although we show that our read alignment scales to pan-genomes made of thousands of human genomes, we still have not tested the full variation pipeline in such a setting. Another problem that we have not explored deeply enough is the construction of the pan-genome itself; there is no unique way to generate the multiple sequence alignment for the set of references, moreover, to obtain an optimal multiple sequence alignment is known to be a difficult problem. Finally, when we extract the ad-hoc reference we do it by extracting a heaviest path from our pan-genome. It is possible to select not one, but the two heaviest paths, so as to extract a diploid reference.

Chapter 6. The contribution of this chapter is more general: we pose a model of sequence alignment where the objects to align are pair-wise alignments themselves. We represent diploid individuals as pair-wise alignments to account for recombinations and phasing errors. We also extended this model to family trios, were we use it to solve a haplotyping problem.

As for the application to solve the haplotyping problem, the natural follow-up is to test our approach with real data. Regarding the more general diploid alignment model, there may be applications in different domains. For instance, we have not explored yet how this generalization of alignment could be integrated into short read alignment.

Chapter 7 The work in this chapter is motivated by a read-based approach to haplotype phasing, which require a preprocessing of the input to prune the reads. We modeled this read pruning problem as an interval scheduling problem, where instead of maximizing a profit, we keep the minimum coverage as high as possible.

Directions for future work in this topic comprise different optimizations models within our interval scheduling model. For instance, it would be possible to keep our current target function, and add a secondary criterion to optimize, such as the average coverage.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylova, W. F. Smyth, G. Tischler, and M. Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Computing Surveys*, 45(1):5, 2012.

[3] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987.

[4] G. A. Auwera et al. From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline. *Current Protocols in Bioinformatics*, 2013.

[5] H. Bai, Guo, et al. The genome of a Mongolian individual reveals the genetic imprints of Mongolians on modern human populations. *Genome biology and evolution*, 6(12):3122–3136, 2014.

[6] D. Belazzougui, V. Mäkinen, and D. Valenzuela. Compressed suffix array. In *Encyclopedia of Algorithms*, pages 386–390. 2016.

[7] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071. SIAM, 2016.

[8] K. I. Bouzina and H. Emmons. Interval scheduling on identical machines. *Journal of Global Optimization*, 9(3-4):379–393, 1996.

[9] M. D. V. Braga, E. Willing, and J. Stoye. Genomic distance with DCJ and indels. In *Proc. 10th Workshop on Algorithms for Bioinformatics (WABI)*, pages 90–101, 2010.

[10] S. R. Browning and B. L. Browning. Haplotype phasing: existing methods and new developments. *Nature Reviews Genetics*, 12(10):703–714, 2011.

[11] P. Brucker and L. Nordmann. The k-track assignment problem. *Computing*, 52(2):97–122, 1994.

[12] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[13] W. Chen et al. Genotype calling and haplotyping in parent-offspring trios. *Genome research*, 23(1):142–151, 2013.

[14] Z.-Z. Chen, F. Deng, and L. Wang. Exact algorithms for haplotype assembly from whole-genome sequence data. *Bioinformatics*, 29(16):1938–1945, 2013.

[15] R. Cilibrasi, L. Van Iersel, S. Kelk, and J. Tromp. On the complexity of several haplotyping problems. In *Proc. 5th Workshop on Algorithms for Bioinformatics (WABI)*, pages 128–139. Springer, 2005.

[16] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[17] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo. Sequence landscapes. *Nucleic Acids Research*, 14(1):141–158, 1986.

[18] F. S. Collins, M. Morgan, and A. Patrinos. The Human Genome Project: lessons from large-scale biology. *Science*, 300(5617), 2003.

[19] Computational Pan-Genomics Consortium. Computational pangenomics: status, promises and challenges. *Briefings in Bioinformatics*, 2016.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[21] A. Danek, S. Deorowicz, and S. Grabowski. Indexing large genome collections on a PC. *PLoS ONE*, 9(10), 2014.

[22] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1998. Second edition.

[23] S. Deorowicz, A. Danek, and S. Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

[24] A. Dilthey et al. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47:682–688, 2015.

[25] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel–Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[26] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[27] Exome Aggregation Consortium. Analysis of protein-coding genetic variation in 60,706 humans. *Nature*, 536(7616):285–291, Aug. 2016.

[28] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[29] H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A*, 372, 2014.

[30] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE, 2000.

[31] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[32] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 150–160. Springer, 2004.

[33] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 768–777. SIAM, 2009.

[34] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, pages 533–544. Springer, 2015.

[35] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[36] E. Garrison et al. FreeBayes. https://github.com/ekg/freebayes, 2016.

[37] P. Gawrychowski. Faster algorithm for computing the edit distance between SLP-compressed strings. In *Proc. of the 19th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 229–236, 2012.

[38] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms, (SEA)*, pages 326–337, 2014.

[39] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. SIAM, 2003.

[40] U. I. Gupta, D.-T. Lee, and J.-T. Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, 11(C-28):807–810, 1979.

[41] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[42] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999.

[43] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[44] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment*, 5(3):265–273, 2011.

[45] R. Horton et al. Variation analysis and gene annotation of eight MHC haplotypes: The MHC Haplotype Project. *Immunogenetics*, 60(1), 2007.

[46] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.

[47] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[48] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[49] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 139–150, 2013.

[50] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 189–200. Springer, 2013.

[51] J. Karkkainen, D. Kempa, and S. J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th Data Compression Conference (DCC)*, pages 153–162. IEEE, 2014.

[52] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*. Citeseer, 1996.

[53] E. Khurana et al. Integrative annotation of variants from 1092 humans: Application to cancer genomics. *Science*, 342(6154), 2013.

[54] A. W. Kolen, J. K. Lenstra, C. H. Papadimitriou, and F. C. Spieksma. Interval scheduling: A survey. *Naval Research Logistics*, 54(5):530–543, 2007.

[55] M. Y. Kovalyov, C. Ng, and T. E. Cheng. Fixed interval scheduling: Models, applications, computational complexity and algorithms. *European Journal of Operational Research*, 178(2):331–342, 2007.

[56] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206. Springer, 2010.

[57] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[58] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–64, Mar. 2002.

[59] H. Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27(21):2987–2993, 2011.

[60] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[61] K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of relative Lempel-Ziv dictionaries. In *Proc. of the 25th International Conference on World Wide Web (WWW)*, pages 807–816. International World Wide Web Conferences Steering Committee, 2016.

[62] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail. Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem. *Briefings in bioinformatics*, 3(1):23–31, 2002.

[63] A. Löytynoja, A. J. Vilella, and N. Goldman. Accurate extension of multiple sequence alignments using a phylogeny-aware graph algorithm. *Bioinformatics*, 28(13):1684–1691, 2012.

[64] S. Maciuca, C. del Ojo Elias, G. McVean, and Z. Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *Proc. 16th Workshop on Algorithms for Bioinformatics (WABI)*, volume 9838, pages 222–233. Springer, 2016.

[65] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[66] D. Medini, C. Donati, H. Tettelin, V. Masignani, and R. Rappuoli. The microbial pan-genome. *Current opinion in genetics & development*, 15(6):589–594, 2005.

[67] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. In *Proc. 7th Workshop on Algorithms for Bioinformatics (WABI)*, pages 289–301. Springer, 2007.

[68] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTC)*, pages 37–42, 1996.

[69] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[70] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.

[71] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[72] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[73] C. Offord. The pangenome: Are single reference genomes dead? `http://www.the-scientist.com/?articles.view/articleNo/47510/title/The-Pangenome--Are-Single-Reference-Genomes-Dead-/`. Accessed: 2016-01-17.

[74] J. O'Rawe et al. Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. *Genome medicine*, 5(3):1, 2013.

[75] J. B. Orlin. Max Flows in $O(nm)$ Time, or Better. In *Proc. of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 765–774, New York, NY, USA, 2013. ACM.

[76] M. Patterson et al. WhatsHap: Haplotype assembly for future-generation sequencing reads. In *Proc. 18th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, volume 8394, pages 237–249, 2014.

[77] D. R. Powell, L. Allison, and T. I. Dix. A versatile divide and conquer technique for optimal string alignment. *Information Processing Letters*, 70(3):127–139, 1999.

[78] S. J. Puglisi. Lempel-Ziv compression. In *Encyclopedia of Algorithms*, pages 1095–1100. Springer, 2016.

[79] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[80] S. Raskhodnikova, D. Ron, R. Rubinfeld, and A. D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013.

[81] A. Regalado. Rebooting the human genome. `https://www.technologyreview.com/s/537916/rebooting-the-human-genome/`. Accessed: 2016-01-17.

[82] R. Rizzi, M. Cairo, V. Mäkinen, and D. Valenzuela. Diploid alignment is NP-hard. *arXiv preprint arXiv:1611.05086*, 2016.

[83] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[84] K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, and D. Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10:R98, 2009.

[85] J. Schröder, S. Girirajan, A. T. Papenfuss, and P. Medvedev. Improving the power of structural variation detection by augmenting the reference. *PLoS ONE*, 10(8), 2015.

[86] J. Sirén, N. Välimäki, and V. Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[87] H. Tettelin et al. Genome analysis of multiple pathogenic isolates of Streptococcus agalactiae: implications for the microbial pan-genome. *Proceedings of the National Academy of Sciences of the United States of America*, 102(39):13950–13955, 2005.

[88] H. Tettelin, D. Riley, C. Cattuto, and D. Medini. Comparative genomics: the bacterial pan-genome. *Current opinion in microbiology*, 11(5):472–477, 2008.

[89] The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, Sept. 2015.

[90] The UK10K Consortium. The UK10K project identifies rare variants in health and disease. *Nature*, 526(7571):82–90, Sept. 2015.

[91] J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. 37th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 283–292, 2014.

[92] TrueSeq. `http://web.archive.org/web/20170108115446/http://www.illumina.com/clinical/illumina_clinical_laboratory/trugenome-clinical-sequencing-services.html`. Accessed: 2016-01-17.

[93] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[94] J. C. Venter et al. The sequence of the human genome. *Science*, 291(5507), 2001.

[95] J. L. Weber and C. Wong. Mutation of human short tandem repeats. *Human molecular genetics*, 2(8):1123–1128, 1993.

[96] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 1–11. IEEE, 1973.

[97] E. Willing, S. Zaccaria, M. D. V. Braga, and J. Stoye. On the inversion-indel distance. *BMC Bioinformatics*, 14(S-15), 2013.

[98] R. Wittler, T. Marschall, A. Schönhuth, and V. Mäkinen. Repeat- and error-aware comparison of deletions. *Bioinformatics*, 31(18):2947–2954, 2015.

[99] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

[100] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[101] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on information theory*, 24(5):530–536, 1978.