Universitat Politècnica de Catalunya

Efficient Approximate String Matching Techniques for Sequence Alignment

Santiago Marco-Sola

Advisor **Paolo Ribeca**
Co-advisor **Roderic Guigó**

Thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computing
of the Polytechnic University of Catalonia, December 2016

Abstract

One of the outstanding milestones achieved in recent years in the field of biotechnology research has been the development of high-throughput sequencing (HTS). Due to the fact that at the moment it is technically impossible to decode the genome as a whole, HTS technologies read billions of relatively short chunks of a genome at random locations. Such reads then need to be located within a reference for the species being studied (that is *aligned* or *mapped* to the genome): for each read one identifies in the reference regions that share a large sequence similarity with it, therefore indicating what the read's point or points of origin may be. HTS technologies are able to re-sequence a human individual (i.e. to establish the differences between his/her individual genome and the reference genome for the human species) in a very short period of time. They have also paved the way for the development of a number of new protocols and methods, leading to novel insights in genomics and biology in general.

However, HTS technologies also pose a challenge to traditional data analysis methods; this is due to the sheer amount of data to be processed and the need for improved alignment algorithms that can generate accurate results quickly.

This thesis tackles the problem of sequence alignment as a step within the analysis of HTS data. Its contributions focus on both the methodological aspects and the algorithmic challenges towards efficient, scalable, and accurate HTS mapping. From a methodological standpoint, this thesis strives to establish a comprehensive framework able to assess the quality of HTS mapping results. In order to be able to do so one has to understand the source and nature of mapping conflicts, and explore the accuracy limits inherent in how sequence alignment is performed for current HTS technologies.

From an algorithmic standpoint, this work introduces state-of-the-art index structures and approximate string matching algorithms. They contribute novel insights that can be used in practical applications towards efficient and accurate read mapping. More in detail, first we present methods able to reduce the storage space taken by indexes for genome-scale references, while still providing fast query access in order to support effective search algorithms. Second, we describe novel filtering techniques that vastly reduce the computational requirements of sequence mapping, but are nonetheless capable of giving strict algorithmic guarantees on the completeness of the results. Finally, this thesis presents new incremental algorithmic techniques able to combine several approximate string matching algorithms; this leads to efficient and flexible search algorithms allowing the user to reach arbitrary search depths.

All algorithms and methodological contributions of this thesis have been implemented as components of a production aligner, the GEM-mapper, which is publicly available, widely used worldwide and cited by a sizeable body of literature. It offers flexible and accurate sequence mapping while outperforming other HTS mappers both as to running time and to the quality of the results it produces.

Acknowledgements

In order to succeed in any endeavour in life, we must remember to be thankful for those close to us we can lean upon. No man is an island, and no project like this can be accomplished without the help and support of many others. Here, I want to thank all those who have stood close to me during these years, giving me their support, knowledge, and trust.

First, I want to thank all my friends and colleagues from the CNAG for their support, all those productive discussions, and, most of all, the good times we spent together. At the same time, I want to express my gratitude for all the support I have had over the years from the direction of the CNAG; especially from Ivo Gut and Simon Heath who always believed in this project, strongly supporting it through thick and thin. In the same way, I want to thank Mercè Juan Badia and Maria Serna for all the help they have given me with administrative matters throughout the whole thesis.

I would also like to thank Juan Carlos Moure and Toni Espinosa. I consider myself very lucky to have found people so genuinely kind and valuable, who have helped me so much in my career. It goes without saying that, had it not been for you, I would not have met Alejandro Chacón. No doubt, he is one of the most hard-working and quick witted people I have ever known. Thank you Alejandro, this project would not have been the same without all your contributions, support, and time.

At the same time, I want to thank Roderic Guigó for his ever correct advice. Thanks for helping me widen my focus when I needed it the most, and realise what really matters. I am in debt for the opportunities you have given me and the trust you have always had in the project and in me.

Somebody once told me that beyond hard work, intelligence, and talent, the key to success is having someone to trust you and give you the opportunity to prove yourself. That person and I have now known each other long enough to see past our differences and agree that we have both learnt a lot during this endeavour. For that, Paolo, I have only gratitude, for all your support, the time you have spent, and the wonderful opportunity you have given to me. No doubt, it has been a tough journey, but if I had to, I would do it all again.

To the three women in my life, and the man who stood by my side when I needed him the most.
There is no point to this life without the certainty of your love and support.

You want to know how I did it? This is how I did it: I never saved anything for the swim back.

Vincent Freeman

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the most outstanding research milestones in biotechnology achieved in recent years has been the development of the so-called high-throughput sequencing (HTS) technologies. These technologies are able to sequence an individual humans genome in very short period of time (compared to Illumina sequencers which can take up to 10 days to finish the process) and produce millions of short reads of the genome in the form of short nucleotide sequences (coming from different parts of the genome). This has established the framework for new analysis protocols where all this information is processed together towards the reconstruction of the individual genome and generation of further results. As a result of the computational challenge, a great deal of research and development of new methods and algorithms has been carried out in the last decades [Goodwin et al., 2016]. Current research in bioinformatics is strongly driven by the need to catch up with sequencing technologies and develop new protocols to give new insights into the genomic understanding of biology [Eisenstein, 2015].

This thesis tackles one of the fundamental problems in data analysis of HTS data: sequence alignment. Once the data (short lectures of DNA) is output by the sequencers (random chunks of DNA from the genome) this data must be joined together so as to reconstruct the original sequence genome of the individual. With the help of a reference genome, every re-sequencing project needs the help of a sequence aligner (a.k.a mapping tool or mapper) to accurately locate the genome position related to the read data.

Despite being easy to formulate, this problem leads to many ambiguities and degrees of freedom in practice. On the top of that, the performance of sequence aligners is critical to coping with the huge amounts of data generated by modern sequencers. But not only that, the quality of the result becomes vital as the downstream analysis relies entirely on the accuracy of this step. A poor methodology or misplaced choice of protocols can lead to erroneous results in the whole analysis.

Many current analysis protocols (e.g. re-sequencing, bisulfite sequencing or de-novo assembly) strongly depend on the results produced by the alignment step. However, each experimental protocol requires different results and thus requires the aligner to adjust the results to better fit the needs of each protocol. Similarly, sequencing data may depict different properties depending not only on the protocol but also on the underlying sequencing technology [Trivedi et al., 2014].

Because of this, modern mappers are compelled to adjust their internal parameters (e.g. error morel and tolerance, read-length, scoring scheme, ...) so as to better suit the needs of each case without sacrificing quality or performance. In this context, versatility and scalability of aligners has become paramount. To evidence this, in the past decade there has been a huge amount of research and development in many alignment tools serving different purposes to fulfill the needs of research making use of these new sequencing biotechnologies [Fonseca et al., 2012].

Until the establishment of HTS technologies, theoretical algorithms from classical literature dealing with sequence alignment (a.k.a approximate string matching or ASM in the field of computer science) were considered with very different scenarios in mind i.e. alignment of common words against texts [Navarro, 2001]. Sequencing technologies (long-read sequencing most of all) produce inputs on a different range of marked parameters (i.e longer reads, higher error, longer text-references, etc). For this reason, traditional algorithms had to be rethought and adapted, but also, development of new algorithms with a strong emphasis on practical optimisations and heuristics has become of critical relevance. Like in other experimental fields with a need for massive computations, an algorithm depicting good theoretical bounds can end up to be impracticable if practical aspects of the implementation and tuning are neglected.

This is not only on the algorithmic level. Data analysis challenges brought by high-throughput technologies have also influenced the design of new data structures for primary and secondary storage [Hayden, 2015]. Many classical index structures for ASM (i.e. Suffix Trees or suffix arrays) have been proved not to be suited for the scale of data managed and, therefore, several proposals have been made to overcome the memory-wall and save disk storage. Among other succinct indexes, the FM-Index (based on the same Burrows Wheeler transform used for data compression) has outstandingly increased in popularity among sequence aligners [Ferragina and Manzini, 2000]. Providing a full-text index in a very small space, this index structure has enabled algorithms to operate in RAM memory with genomes of human size [Li and Durbin, 2010]. This has guided the development of aligners towards its use and research over its inefficiencies and possible optimisations.

On a higher and more operational-oriented level, high-throughput technologies have lead to the development of large and complex analysis pipelines. It is important to note that the requirements of these pipelines have changed progressively throughout the years. Because of the novelty of the data produced and its constant improvement, pipeline development has been forced to keep up with the pace of the sequencing technologies [Marx, 2013]. For that, many frameworks have been proposed to link together small analysis tools and compose flexible and robust pipelines [Bardet et al., 2012; Guo et al., 2015]. Even though, huge advances have been made towards accurate pipelines [Hwang et al., 2015], many stages of common analysis pipelines are still being researched to gain better performance, accuracy and quality-control mechanisms. Thus, standardisation and flexibility has become paramount in the field for every developer and researcher [Gargis et al., 2015].

Such is the scale of the problem and the necessity for scalable solutions that a progressive disconnection from formal methods, using ad-hoc heuristics has occurred [Slater and Birney, 2005]. This is having a serious impact in the field of sequence mapping, metrics for assessing accuracy are often found to be ill-defined and therefore many widely used tools don't produce the expected results (i.e. algorithmic artifacts or inaccurate results [Ribeca and Valiente, 2011]). One of the fundamental aims of this thesis is to emphasis that accuracy and quality-assurance are paramount. This situation requires comprehensive methods and solid tools that can guarantee the quality of the analysis results.

One common misconception is that an algorithm that performs well with a specific protocol, organism and range of input-data parameters can consequently be used in other cases retaining its benefits. Even if good results are achieved for a certain dataset and experiment, the smallest change in the input data can yield poor results if the analysis method is not robust enough and based upon solid algorithmic foundations. The always present trade-off between performance and accuracy often leads to hazardous algorithmic heuristics that cannot lightly be used in similar experiments. Hence, it is essential that algorithmic ideas have to be formally supported by solid analysis than can attain performance and quality guarantees. Good algorithmic ideas often stand by themselves and are proven by good experimental results. However, the other way round, taking specific experimental measurements as reference to build algorithms without proper analysis constitutes a methodological mistake. In this way, algorithmic guarantees on quality and performance is one of the main focus of this thesis.

This thesis does not intend to be an introductory guide to either bioinformatics or sequence alignment algorithms. References are provided throughout the document to give support to many concepts and help the reader understand what the original contributions of the thesis are. However the ultimate goal is to explain, analyse, and benchmark novel algorithmic approaches in the field.

1.1 Context

From the start this thesis project has been strongly coupled to the development of a production-ready mapper, the GEM-mapper (Genomic Multi-tool Mapper). The project, lead by Paolo Ribeca, was launched in 2008 at the CRG within Roderic Guigó's group as a research initiative towards designing an accurate and scalable aligner for high-throughput sequencing data. Because of that, all the ideas and algorithms proposed within this thesis have been subsequently implemented within the GEM-mapper to test their performance in a practical scenario with real data.

Since its adoption by the CNAG (Centro Nacional de Análisis Genómico, the Spanish national sequencing center) in 2010 when the center was created, the GEM-mapper has been used daily. As of 2010 the CNAG was equipped with twelve Illumina GA IIx machines, for a peak sequencing capacity of 150 Gbases/day (equivalent to the resequencing of a human genome). In seven years of operation, its sequencing machines have constantly been upgraded to the newest HiSeq 2000-4000 series, leading to an almost 10 fold increase in peak production capacity. Such breathtaking yields have driven the development of the GEM-mapper towards increasing performance, with the goal of keeping up with the amount of sequencing data produced by the centre. Since 2008 three major versions of the GEM-mapper have been developed and released. They all present significant differences in their architectural design, due to the constant incorporation of improvements and new algorithms to keep up with the pace of sequence production.

Over the years, the CNAG has acquired other sequencing platforms such as the Oxford Nanopore minION. Targeting new protocols and applications, these technologies produce sequencing data with markedly different profiles –longer reads, and different error rates and error models– requiring new analysis tools and, of course, an adapted mapper. In fact, the entire development of the GEM-mapper can be divided into two periods according to the sequencing technology it was focusing on. During the first period, the focus was on relatively short reads (i.e. ¡100nt) with special emphasis on completeness (that is on retrieving all possible alignments) and resolving mapping conflicts (mainly due to inherent repetitions present in the genome). During the second period, the focus was placed on obtaining performance scalability with longer reads, tolerance to higher error rates and generation of meaningful local alignments.

As evidence of its usefulness and suitability to address big genomic projects, the GEM-mapper and several analysis pipelines based on it have been adopted by other institutions and projects, such as the Encyclopedia of DNA Elements (ENCODE) consortium, the Chronic Lymphatic Leukaemia (CLL) Genome Consortium, and the Genetic European Variation in Health and Disease (GEUVADIS) consortium to name a few.

The development of the GEM-mapper has always been strongly influenced by the requirements of the analysis pipelines it is embedded into. For that reason, many features and tools had been implemented to offer compatibility and support to pipeline development. GEM-tools, in particular, is a software suite which implements several tools for quality-control, format conversion, accuracy analysis and alignment manipulation.

Underlying GEM-tools, a library is offered for C and Python developers to benefit from the its components and use its building blocks to quickly develop pipeline solutions. In this way, the GEM-tools library has become the software glue to bring together many tools involved in the mapping stage of production-ready pipelines for genomic data analysis.

Independently and with the boom of massive parallel devices (i.e. GPGPUs), many integrated core architectures (i.e. Intel Phi) and powerful coprocessor units (i.e. FPGAs) have developed. Ribeca's research group established two collaborative studies to assess the suitability and benefits of porting the GEM-mapper to these hardware devices. A first attempt was done in collaboration with Maxeler Technologies and their multiscale dataflow technology based on FPGAs. The second longer term study was done with the Universidad Autónoma de Barcelona (UAB) and was focused on the use of NVDIA's GPUs. These two collaborations highlighted the need to re-design the mapper architecture so as to isolate and group together algorithmic bottlenecks before offloading to these devices (GPU GEM-mapper).

The GEM-mapper continues to be developed, maintained and used by several sequencing centres. Despite the main lines of research still being focused on accuracy and performance, its development also aims to enlarge the scope of its use by finding new protocols and applications (like support for bisulfite protocols or its adoption within assembly workflows). At the same time, newly developed algorithmic techniques aim to deal with upcoming sequencing technologies and handle the ever increasing yields produced.

1.2 Motivation

Although the GEM-mapper project has been strongly conditioned by the constant development of new sequencing technologies and the requirements of increasingly large sequencing analysis projects, its leading primary investigator –Paolo Ribeca– and this thesis' author have tried their best to remain faithful to its initial formulation and purpose. Described below are the main tenets on which this thesis project and, by extension, the GEM-mapper's design, are based.

1. HTS alignment algorithms have to guarantee well defined, quantifiable and most importantly, reproducible accuracy (which is paramount).

   Mapping is a crucial stage in most resequencing pipelines and many other HTS analysis workflows. Good quality results generated by a mapper will heavily influence downstream analysis, ultimately leading to meaningful and accurate scientific findings in biological research and medical diagnostics. Conversely, incorrect or incomplete alignments may lead to misleading or incorrect scientific findings.

   For that reason, accuracy in the results produced by sequence alignment algorithms is paramount. This thesis aims to define a robust methodological framework to assess the accuracy of mapping results. In this spirit, it aspires to establish a common ground allowing different mapping tools to be evaluated and compared using quantifiable metrics. Moreover, all the algorithmic techniques presented in this thesis are analysed from the standpoint of their accuracy, leading to a meaningful measurement of the quality of the results.

2. HTS calls for high performance algorithms that can cope with increasing yields in the production of genomic data.

   Even if a mapping tool delivers high quality results, they could be of no use if the time spent computing them is impractical. So sequence alignment algorithms have to keep up with the pace of HTS technologies. This thesis intends to contribute high performance algorithms for sequence alignment that can be used in a real-life sequencing production environments.

   Nowadays, the price per sequenced base continues to decrease faster than the costs per base related to computational infrastructure (such as expenditure for disk space or storage on the cloud, RAM, or CPU time). Quite soon we may well see an unbalanced scenario where the bulk of the sequence analysis budget goes into computing resources rather than sequencing. We support the idea that mapping should not take the dominant share of the computing resources of a sequencing center. In particular, sufficient computational resources should be left available to subsequent analysis stages.

   In addition, we would like to debunk the common misconception that genome-scale mapping is still strictly limited to high-performance computing (HPC) facilities. We would like to point out that with suitable algorithms and data structures it is feasible for any scientist to run a mapping tool on their laptop (this ability is limited however, with regards to storage, as most experiments nowadays produce more data than the average laptop can contain and process)

3. Developments in sequencing technologies require alignment algorithms that are able to scale well with increasing read length, error rate and size of reference genome.

   Constant developments in sequencing technologies have imposed new requirements on mapping tools. As sequenced reads get longer and exhibit higher degrees of error, many tools and algorithms have become obsolete. Such is the difference in the range of parameters among HTS technologies that, in less than a decade, we have gone from using relatively short reads

(<100nt) with high homology rates (1-5% sequencing error) to very long reads (of the order of thousand of nucleotides) with much higher error rates (with an error rate ranging from 5 to 30% depending on the technology). Similarly, as more complex genomes of new organisms are assembled, we have gone from aligning to references that fit quite easily in memory (for instance *Saccaromices cerevisiae* with its 12Mbp) to genomes that challenge the memory capacity of current computers (such as the *Paris japonica* genome with its 150Gbp). In consequence, such wide shifts in the range of parameters force researchers to develop new scalable, succinct data structures and algorithmic methods.

One of the central focuses of this thesis is to tackle the technically challenging problem of how algorithms scale with longer read-lengths, higher error rates and larger genome references. Throughout this project, we seek suitable algorithmic approaches and present comprehensive benchmarks considering a wide variety of input parameter ranges. Our intention is to clearly state the limitations and tradeoffs of each technique with respect to scalability.

4. New protocols and applications in bioinformatics which use HTS data require flexible mapping tools suitable for a wide variety of diverse experiments.

   One of the purposes of this thesis is to make users aware that different sequencing experiments require different analysis setups. In this spirit, we want to point out that a single mapper configuration is usually unable to suit the different needs of data produced by different technologies, protocols or experiments. A practical mapping tool must be flexible enough to adapt to different scenarios and clearly state its limitations regarding different applications. Likewise, a mapper should be designed considering its place in common analysis pipelines, and favouring interconnectivity among components (for instance following standard I/O formats, data exchange conventions, etc...).

1.3 Methodological tenets. Algorithm Engineering

Traditional algorithm theory has always worked with simple models of problems and machines to search for solutions that can guarantee performance for all possible inputs [Donald, 1999]. Classical approaches have led to mathematically sound, highly reliable, and even elegant algorithms suitable to be applied to a wide range of applications [Cormen, 2009]. Ultimately, these algorithms are supposed to become part of real applications. Unfortunately, this development process is not always straightforward. Transferring algorithms to real applications can be a slow process; sometimes developments don't produce the desired results and others – due to their complexity – are not even worth trying. Working in an applied area – trying to develop practical solutions to real problems – every developer becomes progressively more aware of the gap between algorithm theory and practical algorithms. Realistic modern hardware architecture grows more and more complex (e.g. Out-of-Order pipelines, SIMD instructions, convoluted cache hierarchies, etc) diverging from simplistic computational models proposed by the traditional literature. At the same time, new applications grow in requirements, which forces research onto more elaborate algorithms. Often, these algorithms may neglect promising ideas due to the impracticality of implementing them or just because their mathematical analysis would be too difficult. In the same way, formal algorithm analysis can derive bounds that turn out to be unlikely to appear in practice. Or worse, asymptotic analysis might ignore critical constants that strongly affect experimental runnings – making some algorithms only interesting in the theoretical field. Not to mention the inherent skewness of real input datasets that can make bounds meaningless. Hence, some algorithms are not amenable for formal analysis and in most situations we are unable to suggest useful formal bounds on their performance.

Algorithm engineering (AE) methodology [Sanders, 2009] and its guidelines are adopted for algorithmic research in the context of this thesis. With the aim of providing practical algorithmic solutions for real input-datasets, this methodology aspires to bridge the obvious gap between algorithm theory and experimental software engineering.

The rational behind this methodology holds that isolating the analysis and design of algorithms from implementation and experiment is a very limited approach. Instead, it proposes to join together these activities into a feedback loop of design, analysis, implementation and experimentation. In this way, AE proposes a development cycle focused on algorithm development performing multiple iterations over its substeps (Figure 1.1). This cycle is driven by falsifiable hypotheses validated by experimentation. Results from experiments cannot prove a hypothesis true but they can support it. However, the outcome of the experiments can prove a hypothesis wrong. As a result, hypotheses can come from creatively proposed ideas (as in classical algorithm theory) or from inductive reasoning based on the results from previous experiments – introducing valuable feedback from experimental activities. Because of this, experimental reproducibility is a must. Not only to make the process extendable from any stage of the process, but also to enable other researchers to repeat the experimentation and draw the same conclusion – or spot mistakes in previous iterations.

Below, each of the substeps involved in the AE cycle are briefly explained. We would like to put each in the context of the thesis by presenting actual examples.

- Application, realistic models, and real inputs.

  First of all, following AE principles, the application focus of our research must be defined. For this, realistic models need to be specified to meet the precise requirements of our problem. Selected cases-of-use must be representative of the real applications targeted and must be detailed enough so as to guarantee the success of the process (i.e. applicability in real production setups).

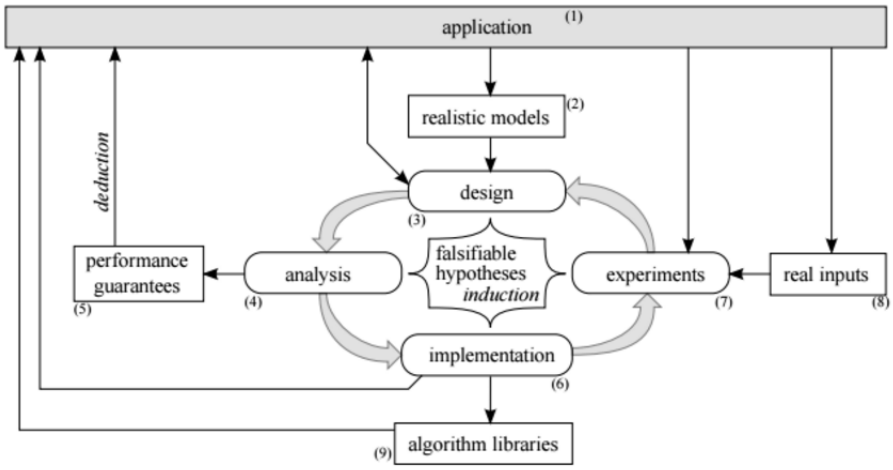


Figure 1.1: AE development cycle [Chimani and Klein, 2010]

Accordingly, we have to select representative input datasets to run the experimentation – which in turn will derive fundamental feedback for our design. Input datasets should come preferably from inputs used in real cases-of-use. However, we will also consider simulated datasets in case this can lead to better grounds to profile experiments and evaluate results (i.e. adding important information not found in real datasets). In any case, models supporting simulated datasets have to be as accurate as possible modelling real inputs – as any factitious bias can lead to dramatically poor assumptions and poor design decisions. In anycase, it is highly recommended to endorse results from simulated data with experiments using real inputs.

In the case of sequence alignment – and within the context of this thesis – we mainly focus on resequencing experiments against the human genome (mainly using GRCh37; Genome Reference Consortium Human genome build 37). The principal reasons supporting this application choice are based on the relevance and wide applicability of this case-of-use (i.e. human biology research, medical research and diagnosis, etc.). At the same time, there are many other relevant applications, mapping to a relatively large mammal reference genome – whose assembly is considered complete and of good quality – which is representative of many other widely used setups. Furthermore, the human genome is large enough to stress many challenging aspects of the problem that go unnoticed using smaller mammal genomes (e.g. Felis Catus, Mus Musculus, Saccharomyces Cerevisiae etc.).

In the same vein, we have compiled a representative ensemble of sequence datasets to be used during experimentation. These datasets come from different technologies and aim to represent the most important technologies and their input variants (i.e protocol, read length, and error degree and model). Below, in Table 1.1 we present a table with the input datasets and their main characteristics.

Independently, we have generated synthetic datasets mimicking the properties of the previous real datasets and other ranges related to testing scalability. Table 1.2 shows a summary of the simulated datasets and their corresponding technology.

Accession	Sequencing technology	Total reads (Millions)	Read length (nt)	Coverage
ERR409730	Illumina HiSeq (SE)	150	100	5x
ERR409730	Illumina HiSeq (PE)	2 x 75	2x100	5x
ERR1122087	Illumina MiSeq (SE)	50	300	5x
ERR1122087	Illumina MiSeq (PE)	2 x 25	2x300	5x
ERR039480	Ion Torrent	80	min=4 avg=188 max=2716	5x
1000genomes	Moleculo	3.8	min=30 avg=3941 max=19553	5x

Table 1.1: Input real datasets

Dataset	Sequencing technology	Total reads (Millions)	Read length (nt)	Coverage
Illumina.l100.se	Illumina SE	150	100	5x
Illumina.l100.pe	Illumina PE	2 x 75	2x100	5x
Sim.Illumina.l300.se	Illumina SE	50	300	5x
Sim.Illumina.l300.pe	Illumina PE	2 x 25	2x300	5x
Sim.Illumina.l500.se	Illumina SE	30	500	5x
Sim.Illumina.l1000.se	Illumina SE	15	1000	5x

Table 1.2: Input simulated datasets

- Algorithmic design step. Design in AE aims to deliver practical solutions. Compared to classical algorithm design, constant factors are not overlooked and play a fundamental role in choosing among different algorithm and data structure designs. In the same way, hidden factors and constraints related to the actual computing architecture are taken into account. Also, asymptotic bounds and amortized costs are carefully considered to avoid neglecting their actual cost in a real scenario.

  This mainly practical approach might favour the choice of simple algorithms that, despite having theoretically intractable bounds, can lead to sufficiently good performance. In practice, and using real inputs, these algorithm might never run into worst case scenarios and can perform adequately (as opposed to other convoluted solutions with theoretical guarantees).

  At this step we also account for a commonly overlooked factor in algorithm design: the implementation complexity. Ideally we would like to derive simple algorithms, easy to be implemented and tested. Experience shows that simple algorithms often lead to fewer bugs – which in turn results in shorter development times – and better performance compared to convoluted algorithmic solutions.

- Analysis step.

  Following the design step, AE indicates the analysis of the previously designed algorithms and data structures from different resource usage perspectives. To that end, we must acquire awareness of realistic models to properly analyse expected running time bounds, memory usage (e.g. overall memory usage , memory hierarchy impact, data pattern access, etc), I/O requirements (most importantly bandwidth required), etc.

This modelling is not only important to highlight the critical parts of our algorithms but to help check initial analysis bounds with experimental results. Most often we may find that experimental results with real input datasets lead to better bounds or hints that can lead to better designs.

- Implementation step.

  Implementation represents one of the most important steps of AE. It aims for the most precise specification of the algorithm and highlights corner cases and details oblivious to previous steps. During implementation certain flaws and omissions are often spotted not obvious during analysis and design steps. While implementing, algorithmic guarantees derived from previous steps can be checked(avoiding bad implementations or regressions in the code). For this reason, it's highly advisable to take advantage of modern development tools (e.g full-fledged IDEs like eclipse, unitary test framework, etc.) and rely on solid software engineering practices. One should never underestimate the usefulness of this often time-consuming process and all the insights from the algorithm that can be derived during the process.

- Experimentation step.

  Experimentation is a fundamental step in AE where previous hypothesis can be disproved and new ones can be derived – to initiate another iteration of the loop. It's often the case that theoretical analysis oversight in estimating the complexity of the algorithm and experimentation shows that the runtime analysis is either wrong, not accurate enough or, in certain cases, behaves counter intuitively. The results of experimentation can help to improve the current algorithm, proposing new design directions. At this point, simulated datasets can become very useful as they can be used to complement real input results. In situations where there is no ground truth available – like in sequence alignment – these synthetic datasets can help to estimate the accuracy of the algorithms proposed. Also, the role of profilers (e.g. Intel VTune Amplifier, Valgrind, Perf, etc.) become very useful. They can be used not only to check all algorithmic guarantees produced previously but also to help the discovery of new bottlenecks.

- Production of some algorithm library or application.

  Last but not least, at the end of potentially multiple iterations over the AE cycle some type of modularised implementation (i.e. library, application or similar) is expected to be delivered. This has no other purpose than the profusion of the algorithm for others to make use of it, make corrections or expand it, applying further iterations.

AE methodology often leads to remarkable improvements in real-case scenarios due to its focus on experimental algorithms and realistic datasets. Ideas and designs that may have been discarded using traditional algorithm theory are considered by AE as long as experimental evidence support their performance. The fact that experimental results play a fundamental role in the AE cycle enables this methodology to reach practical solutions to real-case problems.

1.4 Thesis outline

This thesis is conceptually structured into three main parts. In the first part (Chapters 2 and 3), we present an introduction to relevant topics such as bioinformatics, high-throughput sequencing and sequence alignment. This first block of chapters aims to put this thesis in context, providing background and insights into the challenges and accuracy limitations of sequence alignment. The second block (Chapters 4 and 5) presents the contributions of this thesis to the problem of approximate string matching (ASM) from the standpoint of algorithmic design. Finally, the third block (Chapters 6 and 7) gives a more comprehensive view of the different contributions of this thesis, integrated together into a practical sequence aligner implementation (GEM-mapper).

In more detail, Chapter 2 (Bioinformatics and high-throughput sequencing) serves as a general introduction to bioinformatics and HTS, focusing in particular on protocols and applications relevant to the topics of the thesis. A brief background on a few essential methods and problems is provided.

Chapter 3 (Sequence alignment) analyses in depth the general problem of mapping sequencing reads to a genome. It presents the problem in its duality – first as a practical tool in everyday bioinformatics, then as a theoretical challenge in computer science. Also, it proposes meaningful methods to assess the quality of the results of the alignment process.

Chapter 4 (Index structures) first presents a brief overview of the major index classes that have been traditionally used in order to perform indexed approximate string matching. Focusing on the FM-Index family, we analyse in depth several key design choices and their impact on index performance. This chapter strongly emphasises practical aspects of FM-index design, and how to implement functions that are key to some of the algorithms proposed later on.

Chapter 5 (Filtering algorithms) aims to give a unified vision of classical filtering algorithms towards search-space reduction. It presents the main filtering algorithms this work is based upon and the principles used to combine them towards more efficient and effective filters.

Chapter 6 (Neighbourhood search) describes neighbourhood search algorithms from the brute-force approaches to the most refined versions. Here we present the bidirectional search algorithms and propose an efficient generalisation to edit distance. Additionally, hybrid techniques are presented as an effective method to combine filtering and neighbourhood search algorithms.

Finally, chapter 7 (High-throughput sequencing mappers) addresses the more applied side of this project, describing in detail the internal design of the GEM-mapper. This chapter aims to describe the practical implementation of a production ready genomic sequence aligner. We also aim to link the algorithmic sections of this thesis with their practical implementations, and quantify their performance in a real setup using realistic input datasets. Furthermore, this chapter focuses on highlighting the most critical parts of the implementation, bridging the gap between theoretical and practical performance in a real hardware setup.

Chapter 2

Bioinformatics and High Throughput Sequencing

Having an education in pure computer science, one of the main challenges I was confronted with was to acquire fundamental knowledge of such a multidisciplinary field as bioinformatics. As it is a relatively new field and many methods and approaches are still in their infancy, it can be difficult to keep up with the latest findings and truly grasp their wider biological implications. Not only that, but exposure to all the methodologies and techniques used in sequencing – within such a big sequencing centre as the CNAG – can be overwhelming. Besides the developments in the algorithmic topics of this thesis, I consider that one of the most enriching parts of it was to acquire knowledge in the many fields involved in high-throughput sequencing and bioinformatics in general.

This chapter is intended to give a brief overview of the biological background, technologies and applications required to understand high-throughput sequencing technologies, its results and applications. It is not intended to be a self-contained dissertation on biological or biochemical foundations, but rather a short introduction, with special emphasis on the points relevant to the

thesis and its main points of discussion (i.e. fast and accurate sequence alignment). References are given throughout the chapter to external supporting material.

Section 2.1 (Biological background) aims to give a brief introduction to the molecular biology principles necessary to understand the purposes and challenges of sequencing. Section 2.2 (Sequencing technologies) gives a description of the most relevant modern sequencing approaches and the current technologies available. This section aims to emphasise the main advantages and disadvantages of these technologies and the actual characteristics of the data produced (rather than the chemistry behind the processes). Similarly, in Section 2.3 (Sequencing protocols), the most relevant sequencing protocols are described so as to put them into context and present their main applications. Additionally, Section 2.4 (Bioinformatics applications) describes the most notable applications of sequencing within the context of modern bioinformatics. This section aims to present and justify the critical role of sequencing alignment within these applications and how its performance determines the final quality of the results. Finally, Section 2.5 (High-throughput data analysis) aims to give a glimpse into the complexity of modern sequencing data analysis pipelines and the requirements and impact of the sequence alignment algorithms within.

2.1 Biological background

2.1.1 DNA

Deoxyribonucleic acid (DNA), discovered by Watson and Crick in 1953, is a double helix structured molecule that carries the genetic instructions used by all known living organism and many viruses. Each DNA molecule is made of a chain of nucleotides, also called bases. Each base can be one of four nitrogen-containing nucleobases, either adenine, cytosine, guanine or thymine (denoted A,C,G and T respectively). Nucleotides are chained by covalent bounds between the sugar of one nucleotide and the phosphate of the next, forming an alternated sugar-phosphate backbone.

DNA molecules consist of two strands of nucleotides that bind around each other to form a double helix (the DNA *s*econdary structure). Sequences of nucleotides between the two strands are complementary, with each A on one strand binding to a T on the other one and vice versa, and each C on one strand binding to a G on the other one and vice versa. This constitutes an important redundancy mechanism of DNA. In figure 2.1 we can see a representation of a DNA molecule and its components.

Each DNA strand has an unbound phosphate group attached at the 5' end and a carbon hydroxyl group at the 3' end; where 5' and 3' are the ends of the molecule. Due to this chemical asymmetry of the bounds between nucleotides in each strand, both strands have a direction (i.e. forward or reverse). These long sequences of nucleotides form chromosomes within eukaryotic cells. For instance, human chromosome 1 contains 249 million nucleotide base pairs – or 9% of the complete human genome.

2.1.2 Central dogma of molecular biology

Some subsequences within a chromosome, known as *genes*, encode for proteins. Roughly speaking, there are molecular mechanisms that read these genes and interpret them as instructions to build up proteins from amino acids. Ultimately, proteins constitute the basic building blocks of life, performing a vast number of functions within living organisms. For instance, they are required for the structure, function, and regulation of all tissues and organs of the body.

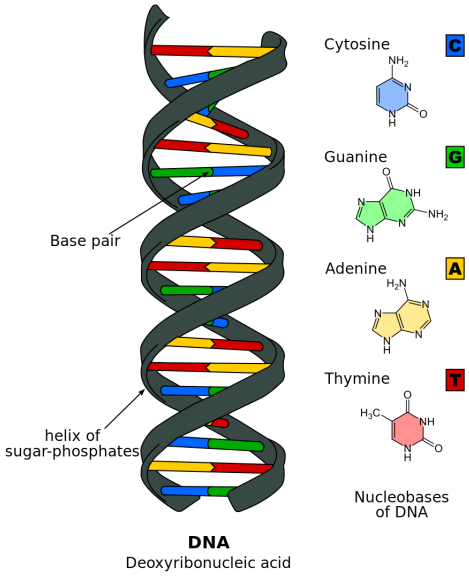


Figure 2.1: Deoxyribonucleic acid

To give more detail, the framework explaining the transfer of sequence information between DNA, RNA and proteins is known as the *central dogma* of molecular biology. First stated by Francis Crick in 1958 [Crick, 1958], it classifies such flows of information into three types: general, special and unknown. The most common are general transfers, describing the flow of genetic information within a biological system [Crick et al., 1970]. That is, a DNA molecule can be duplicated into another DNA molecule (*replication*), this molecule can then be transcribed into mRNA (*transcription*), and finally mRNA can be synthesized into proteins (*translation*).

2.1.3 Replication

In order to make biological inheritance possible, the DNA molecule has to be replicated. During DNA replication, bonds between the nucleotides on opposite strands of the helix are broken and thus both strands of the DNA are separated. In this way, each strand can serve as a template for the generation of its counterpart (semiconservative replication). Each new counterpart being synthesized by the DNA enzyme polymeraze, reading in the direction 3' to 5' and adding the complementary nucleotides to each template. Additionally, cellular proofreading and error-checking mechanisms take place to ensure almost perfect replication [Berg et al., 2006]. Nevertheless, errors can occur and go undetected (see section 2.1.4).

The polymerase chain reaction (PCR) is a common laboratory experiment technique that replicates DNA molecules in-vitro. It cyclically duplicates all DNA fragments bracketed between two known short sequences (*primers*) thus amplifying the original number of molecules by several orders of magnitude in a short time. PCR is now a common technique in research and clinical laboratories, used for a wide range of different applications. Many sequencing technologies rely upon PCR to amplify initial samples before actual sequencing.

It is important to note that PCR is prone to error. Nonspecific binding of the primers involved in the process can lead to errors during replication, causing PCR-amplified molecules to be different from the original ones. This strongly affects some sequencing protocols and is the primary

reason why several service providers employ PCR-free protocols whenever possible (i.e. when the original concentration of DNA molecules in the sample is sufficient to be detected without amplification). In addition, a lot of GC bases contained in DNA fragments pose difficulties during amplification and sequencing [Aird et al., 2011]. Specifically, GC-rich DNA sequences (typically when guanines and cytosines are ¿60%) are more stable than sequences with low GC-content. PCR amplification of these templates is often hampered by the formation of secondary structures, which do not melt at the temperatures usually used when the protocol is executed [Frey et al., 2008]. This leads to systematic underrepresentation of GC-low sequences in sequenced data – or to an overrepresentation of sequences with balanced AT vs. GC content.

2.1.4 Genetic Variations

In any species genomic information is passed from one individual to its offspring in the next generation. In general, genomes of different individuals within a population accumulate small differences at the sequence level. Differences can come from random mutations due to DNA exposure to chemicals or small errors in DNA replication mechanisms. Thus, when referring to the *reference genome* of a particular species we denote a consensus genome drawn from multiple individuals. Variations with respect to the reference genome are what distinguish an individual from the rest of the population.

Most of these variations take place at a small scale. Within these, we define *single nucleotide polymorphism* (SNP) a single base being turned into another one, and *single nucleotide variation* (SNV) a single base being inserted or deleted. Although these happen to be the most frequent types of variation, we can also observe insertions or deletions (*indels*) of several bases (typically $<$ 1Kbp). Variations on a larger scale ($>$ 1Kbp) are called *large structural variations*. To name just a few of these events, one can observe the translocation of subsequences to different locations, inversion of entire subsequences, fusions/fissions of chromosomes, copy number variations, and more [Feuk et al., 2006]. These variations can happen to both coding and non-coding segments of the genome.

It is important to note that a cell does not need to be *haploid*, that is to say, it can have more than just one set of chromosomes. In fact *diploid* cells like those of humans contain two copies of each chromosome, of which one copy is inherited from the father, and another from the mother. These copies are known as *homologous chromosomes* or *haplotypes*. In eukaryotes, the productions of offspring involves the genetic recombination of pairs of haplotypes via the random combination of fragments from both chromosomes. Variants can occur in these haplotypes; we denote as alleles those variants in the form of a gene. Variants occurring in just a single haplotype are called heterozygous and variants found at both haplotypes are called homozygous. If both alleles at a gene (i.e. locus) are the same for both haplotypes, we called it homozygous (with respect to that gene or locus). Otherwise, if the alleles are different, we called it heterozygous. The specific mutations of a specific individual make up its genotype.

Imperfect replication is not the only source of variations. Another good example are transposons and retroviruses, that can copy their genetic material and integrate it into the host genome at multiple locations. The human genome contains several transposable elements, such as those in the Alu family. Alu elements are the most abundant transposable elements, having about a million copies throughout the human genome [Szmulewicz et al., 1998]. Likewise, many endogenous retroviral sequences can also be found in the human genome [Griffiths, 2001].

2.2 Sequencing technologies

Broadly speaking, we understand sequencing to be the process of obtaining the sequence of bases (or formula) of DNA molecules. The ability to sequence genomes is one of the major sources of today's biological information, enabling scientists to investigate cellular processes and leading to invaluable biological insights. Since the advent of the first experimental techniques allowing it, DNA sequencing has become a key technology not only for general biological research but also for many applied fields such as biotechnology, evolutionary studies, metagenomics, medical diagnostics and personalised medicine.

Starting with the original Sanger sequencing [Sanger et al., 1977] which allowed the decoding of the first bacteriophage genome in 1977, sequencing technologies have exponentially progressed to the point that several thousand genomes from different species are now known. A key landmark in the development of sequencing was the completion of the first Human Genome draft in the year 2001. Published simultaneously by the Human Genome Project (HGP) [Lander et al., 2001] and Celera Genomics [Venter et al., 2001], the first reference genome for the human species is considered a fundamental milestone in sequencing. Not only did these first initiatives require a conspicuous investment; they also emphasised the need for building computing infrastructures and writing advanced software in order to support new areas of biological research.

The Human Genome Project was funded with three billion US dollars, and took approximately a decade to complete. It is estimated that by 2003 the cost per sequencing base was $1. A decade later, by 2012, so-called high-throughput sequencing technologies had brought the cost down to an estimated 4 cents per million bases. In 2016, Illumina claimed that a set of ten HiSeq X ultra-high-throughput sequencers could *re-sequence* (i.e. establish the differences between the genome of an individual and the reference) over 18,000 human genomes per year at the price of $1000 per genome. Upcoming improvements in sequencing technologies promise further drops in costs in the near future. This descending trend is super-exponential and much faster than the one observed in computers with the so-called Moore's Law (Figure 2.2).

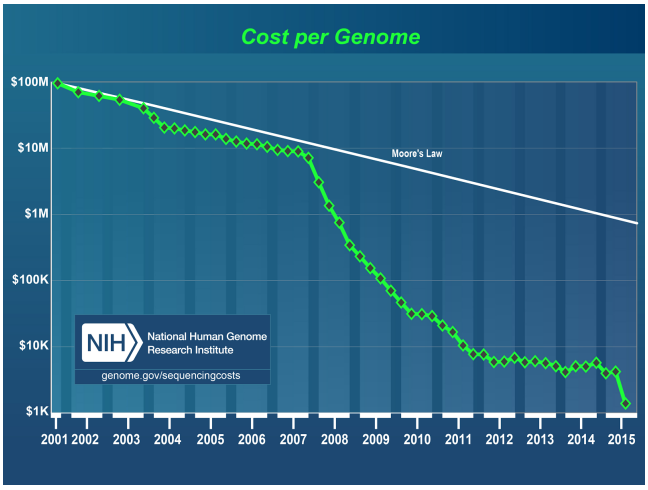


Figure 2.2: Cost per re-sequenced genome (Source NIH)

The future of sequencing technologies is difficult to predict. Nevertheless, it is safe to assume that improvements provided by new technologies will mostly target the production of longer reads with better quality. The need for long reads which are able to sequence complex regions of the

genome (highly repetitive, high GC-content, etc.) is becoming the priority for many companies. We also expect to see a continued increases in the production yields as more and more facilities (e.g hospitals, research centres, pharma companies, etc) acquire sequencers for standard laboratory procedures. The proliferation of sequencing procedures and massive production of genomic data for multiple day to day purposes is deemed to be the next milestone for sequencing technologies. This will emphasise even more the need for high quality techniques and standardised procedures that can run efficiently in a short period of time.

At the moment, there are a wide variety of sequencing technologies available. Each technology produces reads with different characteristics and which have their own advantages and limitations. Hence, it is very important to understand these characteristics and their importance in applications. Depending on the targeted application, the budget, and the required properties of the results one may choose among several different technological options [Mardis, 2011]. In table 2.1, we present a summary of the most relevant technologies in the context of this thesis. In the following sections, these sequencing technologies are briefly presented. This section is not meant to be a profound dissertation on sequencing technologies but a background section with emphasis on the characteristics of each technology (e.g. sources of error, error rates, read length, etc) that are relevant to sequence alignment.

Technology	Read length (average)	Accuracy	Reads per run	Time per run (approx.)	Estimated cost per million bases
Illumina 2500	100-150 bp	99%	3-4 B	6-11 days	¡ $0.043
Illumina MiSeq	300 bp	99%	25 M	65h	$0.93
Illumina X Ten	150 bp	99%	6 G	3 days	$0.007
IonTorrent	200-400 bp	98%	80 M	2h	$1
PacBio	15 Kbp	85%-90%	50.000	4h	$0.6
Nanopore	6 Kbp	90%	4 M	48h	$0.5
Moleculo	1 Kbp	98%	n/a	1-7 days	¡ $0.05

Table 2.1: Sequencing technologies and characteristics

2.2.1 High-throughput sequencing

High-throughput sequencing (HTS) principally denotes those technologies capable of sequencing in the order of millions of reads per run, relatively quickly (from a few days to several hours) at very low price (less than $1 per million bases). Using them, we can deep-sequence a genome (cover every base of a genome with a large number of reads) producing massive amounts of genomic data. These techniques are often called massive parallel DNA sequencing as they rely upon millions of reactions run simultaneously to achieve very high yields of production [Reuter et al., 2015; Loman et al., 2012].

These advances in DNA sequencing have enabled individual labs to run sequencing experiments on their own. Thus, HTS and genome scale sequencing have revealed new challenges not only at the level of greater infrastructure and equipment but also for new algorithmic methods that can cope efficiently and accurately with the massive amounts of data coming from HTS. They have also lead to the development of many new analytical tools for measuring important biological processes (e.g. variant calling, splicing, detecting protein binding, gene annotation and expression, copy number variation, and more).

Illumina (Solexa) sequencing

During sample preparation, a DNA molecule is fragmented and those fragments are selected according to their size (size selection). Afterwards, adaptors are attached to both ends of the fragments and added to a flow cell. A flow cell contains probes complementary to the adaptors attached, and thus, allows the adaptors to hybridise to the flow cell. The free ends of the fragments attach with their complementary adaptors on the flow cell creating a bridge shape. Templates are then amplified using bridge amplification (PCR amplification based technique). Ultimately, this creates clusters on the flow cell of the same sequence. Once clusters are formed, reversible terminator chemistry is used to read one nucleotide at a time. For that, DNA chains are extended in each cycle, adding four types of reversible terminator bases (RT-bases). A high precision camera takes images of the fluorescently labelled nucleotides. Afterwards, the dye is chemically removed and the process is repeated until the sequenced is completely read. This process is performed in parallel for millions of clusters and, because of that, this high-throughput technology can easily sequence several million nucleotides per second. As a result, all sequence reads have the same read length (i.e total number of cycles the sequencer has performed).

Despite being one of the most accurate HTS technologies available, the resulting reads exhibit occasional sequencing errors with mismatches mismatches the most common. Errors tend to accumulate due to a failure during the removal of the the fluorescent tags. Hence, the sequencing quality decreases with each cycle (i.e. an error in an earlier cycle can be propagated and affect subsequent cycles). As figure 2.3 shows, this usually results in errors appearing more frequently towards the end of a read, which ultimately limits the overall DNA fragment size that can be sequenced [Dohm et al., 2008]. On top of that, dephasing or phasing errors can happen when a nucleotide is mistakenly added or removed. Furthermore, many steps taken during sample preparation can introduce bias. Besides PCR amplification biases (2.4.1 Replication), DNA molecule shearing turns out not to be entirely random leading to biases in the generated fragments [Schwartz and Farman, 2010].

IonTorrent sequencing

Ion Torrent [Rusk, 2011] library preparation starts with the shearing of a DNA sample and the addition of adaptors. Once this is done, the DNA is separated, attached to beads and amplified using emulsion PCR. Afterwards, primers are added to the beads and they are placed in wells on a CMOS chip (Ion Chip). Finally, the template is sequenced as the chip detects changes in pH levels as nucleotides are added.

Ion torrent's main source of error are indels, most of all related with stretches in homopolymers. Signals generated from a high repeat number are difficult to accurately measure. Hence, a long homopolymer can defy the sensitivity of the method and fail to pick up the actual length. On the other hand, the occurrence of mismatches is very low and though its throughput is lower than that of other HTS technologies, the average read-length produced is higher.

2.2.2 Synthetic long-read sequencing

In order to increase the effective read length produced by Illumina platforms, Moleculo TruSeq synthetic long-read sequencing (SLRS) was developed. This method is based on a modified library preparation protocol in which long fragments of DNA – reaching 10 Kbp long – are extracted, amplified and reassembled from regular short-reads. These long fragments are placed into wells (3000 fragments per well) and then assigned a unique barcode sequence. This is followed by a regular Illumina sequencing procedure. Later, all the sequenced reads are de-barcoded into pools and de-novo assembled independently.

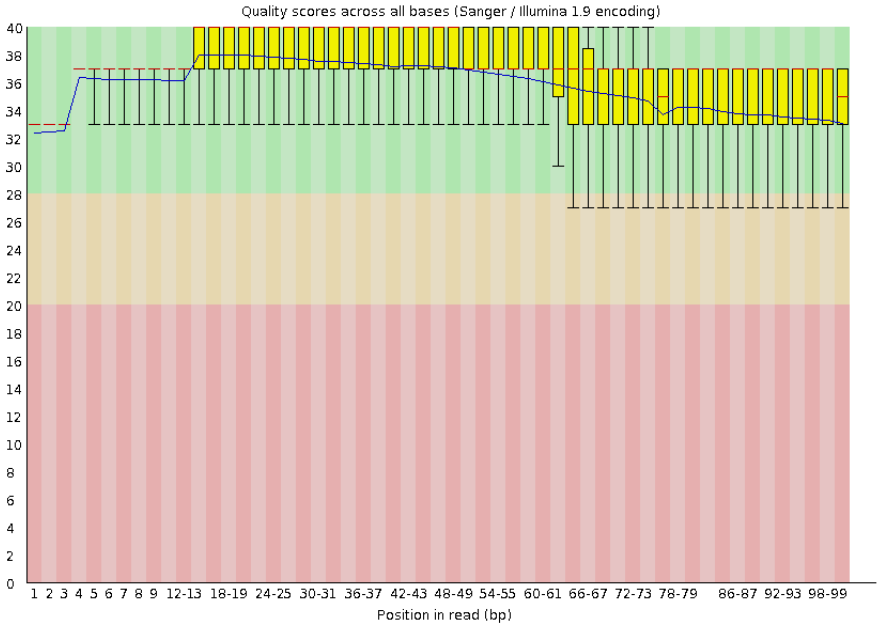


Figure 2.3: Illumina quality profile

Moleculo SLRS is primarily used in applications where traditional short-reads are simply not good enough. This higher sequencing resolution not only helps with genome finishing but also other applications like de-novo assembly, metagenomics, and whole human genome phasing (2.4 Bioinformatics applications). Nevertheless, this method has a few shortcomings. First, the reads produced are relatively short compared to other true long-read technologies – like PacBio Single Molecule Sequencing. But furthermore, this technique is known to be prone to biases, due mainly to the de-novo assembly phase of the pools and its sensitivity to the presence of repeats.

2.2.3 Single molecule sequencing

Single molecule sequencing (SMS) looks to provide solution to some of the most vexing problems coming from HTS technologies. It aims to simplify sample preparation, reduce the amount of sample required for operation and remove the need for template amplification – in turn removing biases and artifacts.

Probably the most prominent advantage of SMS is the ability to produce longer reads compared to those of HTS. More importantly, these longer reads can favour simpler analysis, increasing the sequencing resolution and removing conflicts. For that reason, SMS techniques are preferred for applications such as genome assembly where conflict heavily limits the quality of the final results – SMRT-based assembly produces 100 times longer contigs with 100 times fewer gaps compared to the SGS-based assemblies [Sakai et al., 2015]. In addition, with longer reads, we can sequence large repetitive regions and detect mutations – many of which are associated with diseases – and characterise long structural variations. SMS can also sequence those samples unfeasible to be amplified because of rich GC content, secondary structure, etc... [Ross et al., 2013]. Furthermore, SMS can resequence the same molecule multiple times for improved accuracy. However, some SMS methods are still in their developmental infancy. As a result, one of their main weaknesses is their relatively high error rate (reaching almost 20%). In addition to using SMS, many hybrid sequencing

strategies have been proposed to employ short reads in conjunction with long reads and increase overall accuracy of sequenced data. Hybrid sequencing strategies are much more affordable and scalable than using SMS technologies alone.

The Most prominent SMS technologies are PacBio Real-time Sequencing and Nanopore Sequencing. In a nutshell, PacBio sequencing is a method for real-time sequencing; not requiring a pause between read steps [Rhoads and Au, 2015]. Single molecule real time sequencing (SMRT) utilises a zero-mode waveguide (ZMW) that creates an illuminated observation volume that is small enough to observe only a single nucleotide of DNA being incorporated by DNA polymerase [Levene et al., 2003]. Improving steadily, Nanopore sequencing is being consolidated as the main alternative to PacBio. Nanopore technologies offer long read sequencing at low cost and high speed with minimal sample preparation [Laszlo et al., 2014]. It aims to use a voltage-biased nanoscale pore within a membrane to measure the passage of a single-stranded DNA molecule [Deamer et al., 2016].

2.3 Sequencing protocols

It is well understood that a successful experiment relies on proper sample and library preparation prior to actual sequencing. In actual fact, as of today many different library preparation protocols exist; they allow scientists to interrogate the genome and the inner workings of a range of compartments within the cell in different ways. In general terms, each protocol requires different methods of extracting nucleic acid (DNA or RNA) from the cell, preparing it for sequencing and analysing it after sequencing.

The possibilities and benefits of having different protocols available are manifold; not only do they enable sequencing of different parts of the genome with different content and function, but they also allow the production of sequencing data with different characteristics that can help specific biological analysis. Hence, we have different protocols requiring different analysis methods and algorithms. Here, we introduce the most relevant protocols to this thesis.

2.3.1 Single-end, paired-end and mate-pair sequencing

What we know as single-end (SE) sequencing involves the DNA molecule being sequenced from only one end (Figure 2.4.a). However, the Illumina platform allows for other protocols to take advantage of the process by reading the molecule from two ends at the same time producing different types of sequence reads that contain very useful information to many applications.

Paired-end (PE) sequencing produces pairs of reads with known relative layouts. For this, size selection is performed during sample preparation so that fragments of a specific length are isolated (e.g. 500bp). The selected fragments are then sequenced from both ends, producing pairs. While we do not know the precise orientation of the pairs and the sequence in between them, we know the relative pair orientation (i.e end1-forward end2-reverse, or end1-forward end2-reverse) and an approximation of the distant between the two ends (Figure 2.4.b).

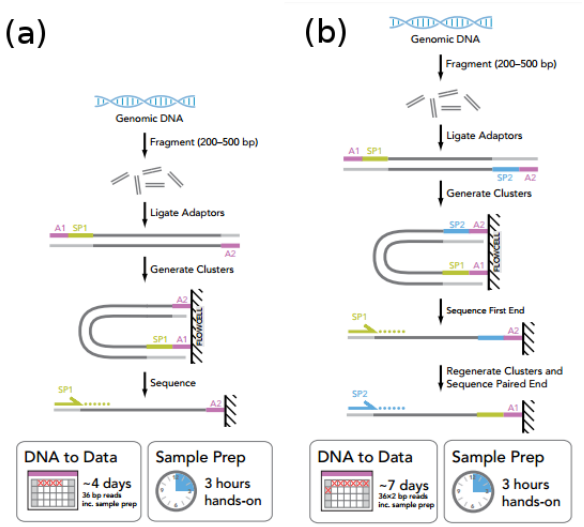


Figure 2.4: (a) Single-end protocol. (b) Paired-end protocol (Source Illumina genomic sequence datasheet)

So-called mate-pair (MP) protocols involve different sample preparation techniques that produces different relative pair orientations and allow greater insert sizes. It is essentially based on the use of larger size-selected DNA fragments (e.g. 4 Kbp) and the circulariation of those fragments. Afterwards, the fragments are randomly sheared, selected and sequenced using PE protocol. In this case, pairs face away from each other and insert sizes are much longer (Figure 2.5). Both PE and MP protocols turn out to be quite useful for conflict resolution in many applications like mapping or genome assemble. For the latter, MP protocols lead to data that can link together relatively distant parts of the genome. Thus, it can arrange together contigs from different parts of genome separated by regions that were not susceptible of being assembled (e.g highly repetitive regions).

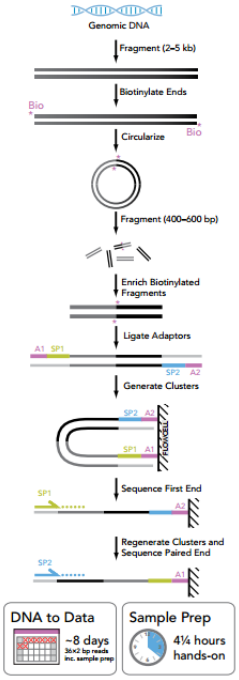


Figure 2.5: Mate pair protocol (Source Illumina genomic sequence datasheet)

2.3.2 Whole-genome and targeted resequencing

Considering the regions of the genome that we aim to sequence, we mainly differentiate between Whole-Genome Sequencing protocols and targeted protocols. Whole-Genome Sequencing (WGS) aims to sequence the whole genome of an organism (i.e. all an organism's chromosomal and mitochondrial DNA). It constitutes the most comprehensive method of interrogating the human genome.

As opposed to WGS, targeted sequencing isolates and sequences a subset of genes – or any region of interest. This focus on specific regions of the genome achieves greater coverage on those regions and allows researchers to focus efforts, time and expenses on those areas of interest [Ng et al., 2009]. Compared to WGS – which normally achieves coverage levels of 30x-50x – targeted sequencing can easily reach coverage levels of 1000x or higher in the regions of interest. This can lead to the discovery of rare variants which would be too rare or expensive to identify using WGS.

It is important to note that certain regions are harder to target due to the difficulty of designing unique probes (e.g. repetitive sequences) or due to lower binding affinity [Zhang et al., 2003]. As a result, this may lead to GC biases. Furthermore, preferential capture of the reference allele is know to introduce biases towards the allele, making it more difficult to call variant alleles [Turner et al., 2009].

Whole Exome Sequencing (WES) is perhaps the most widely used targeted sequencing method. Representing less than the 2% of the human genome, the exome contains the vast majority of disease-causing variants and therefore constitutes a good alternative to WGS in cost. With WES the protein-coding portions of the genome are selectively sequenced allowing identification of variants for a wide range of applications (e.g. cancer studies, genetic diseases, etc).

2.3.3 Bisulfite sequencing

DNA methylation is a biological process by which a methyl-group reacts with a cytosine or adenine nucleotide. This modifies the DNA molecule and its function, altering the genes expressed and proteins developed in cells. DNA methylation is an epigenetic process; although the DNA molecule is modified, it is a heritable change that does not influence the actual DNA sequence. In this way, it is sometimes explained as a pattern overlaying the actual DNA sequence. DNA methylation is a common process within living creatures, playing part in embryonic development, repression of repetitive elements, ageing or genomic imprinting to name a few. Aberrant DNA methylation patterns can have harmful consequences like cancer development and other human diseases related to epigenetic inconsistencies.

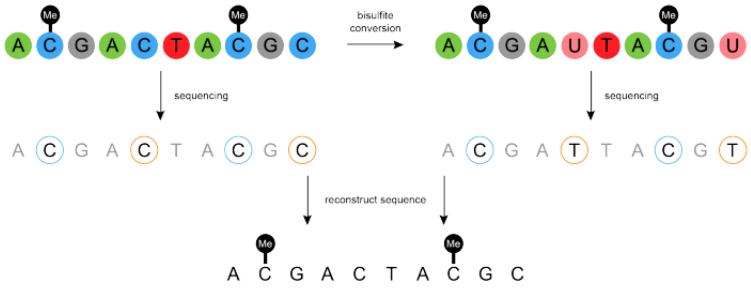


Figure 2.6: Bisulfite Protocol (Source atdbio)

Bisulfite sequencing involves the treatment of DNA molecules with bisulfite ions with the purpose of analysing methylation patterns. Treatment of DNA with bisulfite converts cytosine nucleotides to uracil, yet methylated cytosines remain unaffected. The methylation state of the original DNA can be inferred by counting the number of cytosines and thymines at the genomic cytosine sites (Figure 2.6). We denote CpG sites to those regions where cytosine and guanine are found next to one another (in that order) where cytosine can undergo methylation. CpG regions that experience methylation are referred to as differentially methylated regions (DMR). DNA methylation was the first discovered epigenetic marker, and today still remains the most studied. By means of HTS analysis, whole genome bisulfite sequencing aims to discover DNA methylation patterns of the entire genome by determining the methylation status at CpG dinucleotides.

2.4 Bioinformatics applications

Broadly speaking, bioinformatics is an interdisciplinary field that employs computers to solve problems from molecular biology. It mainly involves the development of algorithms and tools that can analyse and interpret huge amounts of biological data (e.g nucleotide sequences, protein domains, protein structures, etc) that would be unfeasible without automated computation. But bioinformatics also entails the creation and curation of databases, development of statistical techniques, and the advancement of theory to solve formal problems arising from the analysis of genome-scale biological data. Since the development of the human genome project, the field of bioinformatics has become of great interest. Nowadays, bioinformatics is evolving rapidly alongside the latest technological improvements in sequencing platforms. Many researchers have been joining forces in multidisciplinary teams so as to interrogate the insights of genome biology. Some scientists draw a distinction between the term bioinformatics and computational biology. While these areas are indeed broad and diverse, these distinctions in terms are not consistent or well-defined.

Good examples of bioinformatic applications are protein folding, construction of evolutionary trees, genome mapping and variant detection. Among many others, one of the fundamental applications of bioinformatics is genome mapping. As explained above, due to technical limitations a single chromosome cannot be read all at once. However, as we know, relatively small sequences of DNA can be obtained by sequencing. It is important to note that a read can originate from either of the two strands of a chromosome. This can become even more complicated in the case of polyploid organism, where the read can come from either of the haplotypes. For the majority of applications, all these small pieces have to be arranged together like a jigsaw to obtain a de-novo genome (de-novo assembly) or aligned to an already existing genome (resequencing). In order to illustrate the role of mapping and its relevance in these instances, we briefly present a few applications where mapping is used and plays a crucial role in its success.

2.4.1 Resequencing

The most common application of HTS is genome resequencing – most of all with the human genome. It aims to reconstruct the whole genome of an individual using a previously assembled genome as a template (reference genome). During the process, all sequenced reads are mapped against the reference genome looking for the most plausible location for where the read was sequenced from. Allowing certain degrees of divergence between the reads and the reference (due to sequencing errors and differences between haplotypes), all the sequenced reads are laid down and the actual genotype of the individual is reconstructed. By identifying the differences between the DNA of the sequenced genomes and the reference, a catalog of mutations specific to the sequenced individuals is obtained (mostly SNPs and SNVs). It is quite common that these variations are associated with specific phenotypes. Ultimately, this can provide extremely valuable insights into the genetic background of individuals.

The most important limiting factor, that determines the quality of the result, is the total coverage of the sequenced genome (Definition 2.4.1). In general, the greater the coverage, the easier it is to reconstruct the individual's genome and to accurately identify its own variants. At the same time, there are other factors that also play a fundamental role in the success of any resequencing application – such as the read length, homology between individual and reference and quality of the reference.

Definition 2.4.1 (Coverage. Lander/Waterman equation)**.** Coverage is the average number of times each base of the genome is sequenced [Lander and Waterman, 1988; Roach et al., 1995].

$$Coverage = \frac{ReadLength \times NumberReads}{GenomeLength}$$

Genome resequencing is often preferred because of its relative simplicity, practical efficiency compared to other genome reconstruction methods (i.e de-novo assembly), and reduced cost. However, its greatest shortcoming is the need for a complete and reliable reference. It is important to note that many organism's genomes haven't been assembled yet – making them unsuitable for resequencing. More genome references are steadily being released, with better quality and larger genomic spans. However, many genome references still have poor quality and many lack some genomic regions – mainly because of their complexity. On top of that, for some resequencing experiments, the sequenced sample can depict aberrant variations that might be difficult – if not impossible – to align against the reference (e.g. long insertions or translocations).

Sequence alignment plays a fundamental role in resequencing experiments as it is responsible for mapping each read in its sequenced locus. In this way, mapping tools must pick out the most likely source location in the reference genome allowing certain error divergence from it. Not only is the final outcome strongly determined by the accuracy of the mapper but also the overall performance. It is important to highlight that for common resequencing experiments with 30x coverage of the human genome, a HTS Illumina machine will generate 100 nt x 900 million reads. Mapped against 3,000 million bases of the genome, this constitutes a challenging problem of performance for modern mapping tools.

2.4.2 De-novo assembly

As mentioned before, in case there is no known reference genome or if the reference simply is not accurate enough, HTS data can be arranged together to assemble the individual's genome from scratch without a template guide (de-novo assembly). Many programs for genome assembly had been developed in recent years using a variety of different algorithms. However, they all follow the same conceptual steps (Figure 2.7). First, the algorithm has to compare every read against every other looking for overlaps. Next, overlapping reads are joined together in a sequence graph that leads to a partial assembly of contigs (i.e continuous sequence where the fragments overlap). Finally, all the contigs are assembled together into scaffolds that ultimately form the assembled genome.

This process is far from perfect and errors can occur for several reasons (e.g. sequenced reads incorrectly discarded, mistakes joining overlapping reads, misassemblies due to repeats and many others). Highly repetitive sequences, variants, missing regions and mistakes limit the final length of the contigs that assemblers can produce.

Sequence alignment plays a critical role at the moment of finding overlaps between reads. Not only is it one of the most computationally intensive steps of the process, but also the accuracy of its results will strongly influence further stages and ultimately the final outcome of the assembler. An insensitive mapper that misses overlaps between reads will also miss output links between contigs and thus, will inevitably lead to smaller and disconnected contigs. In the opposite case, a mapper that assess relations between incorrect reads will lead to unnecessarily complex assembly graphs and possibly to artefacts in the final contigs.

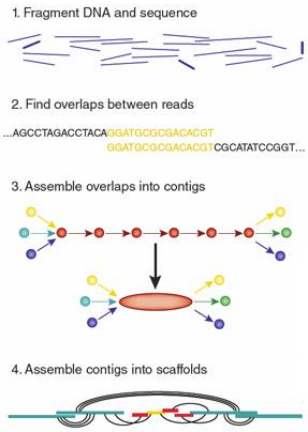


Figure 2.7: Basic steps of a genome assembler [Baker, 2012]

2.4.3 Other applications

Besides generic genome resequencing and de-novo assembly, there are many other applications where mapping plays a fundamental role. This comes to justify not only the wide application of sequencing alignment algorithms but also the need for flexibility so as to cover the requirements of different bioinformatic applications.

A good example of the non-conventional use of a mapping tool is comparisons between closely related species. This can be motivated by the lack of a reference genome for a specific species (but where a close relative reference is known) or with the aim to study genomic differences between related species. In both cases, reads sequenced from one species are mapped against the reference of a closely related species. The underlying sequence alignment algorithms have to allow for greater divergence between genomes and accurately map reads leading to variations between the two species. A good example is the comparison between the bonobo genome and the chimpanzee and human genomes [Prüfer et al., 2012].

In a more general way, high-level genome comparisons also rely internally on sequence alignment algorithms to discover mayor structural variations between species [Nelson et al., 1999]. The field of comparative genomics exploits the similarities – and differences – between genomes. It holds that common features of different organisms will often be evolutionary conserved and encoded in their DNA [Bansal and Meyer, 2002]. To that end, they use specialised alignment algorithms with the purpose of identifying not only small variations (i.e. SNP, SNV, ...) but large rearrangements, reversals, tandem repeats, etc. MUMmer [Delcher et al., 1999] is the most representative tool in bioinformatics used for this purpose.

Another application where mapping comes in very useful is avoiding sample contamination after sequencing. Within many protocols, bacterias and viruses are used to aid sample preparation. In some cases, genomic material of these organism gets mixed up with the host sample. As a result, this contamination is sequenced too and the resulting reads contain sequences belonging to these foreign organisms. In order to decontaminate the sequenced samples, a special process of mapping is performed. In this way, all the reads are mapped against a made-up genome containing the reference of all possibly involved organism. In this way, the mapping tool is supposed to disambiguate the source organism of each sequenced read and discard those ambiguous ones (if any). As a result, all the reads belonging to the host sample are isolated so as not to interfere with downstream applications or analysis.

An additional example of challenging applications for mapping can be found within metagenomics [Thomas et al., 2012]. Here, a sample containing DNA fragments from multiple organisms has to be mapped against a reference containing the genomes of multiple species. The goal is to correctly identify the organism (or the phylogenetic group) and the source putative genomic location overcoming potential ambiguities between strains and repetitive sequences of the reference [Ribeca and Valiente, 2011].

2.5 High-throughput data analysis

As bioinformatic applications evolve, their scope is increasing and becoming ever more complex. For that reason, applications cannot be fully understood in a vacuum and must be studied in the context of large analysis pipelines involving very different components. Current HTS data analysis pipelines consist of several steps and involve numerous tools. Generally, these analysis pipelines are divided into three stages: primary, secondary, and tertiary analysis.

Primary analysis consists of machine specific steps to call base pairs and compute quality scores. Basically, the base calling step converts raw sequencing signals from the sequencer into bases (i.e. A,C,G,T, and N for unknown calls). Additionally, this stage is expected to deliver a quality metric of the calling step. For this, quality values are assigned to each base call to estimate the likelihood of an erroneous call at that base. This stage is tightly coupled with the precise instrument for sequencing used and current generation machines often perform these computations using bundled hardware (i.e specific hardware setups). The most common result is a FASTQ file [Cock et al., 2010]. In this simple file format, each sequence is represented using four lines (Figure 2.8). The first contains the tag of the read, the second represents the sequenced bases, the third line is left to optionally add comments – and for padding purposes – and the fourth contains the quality of each base call in phred format [Ewing and Green, 1998]. Phred qualities are error probabilities encoded into ASCII printable characters. This file constitutes the basic raw sequencing data that secondary analysis will employ.

```
@ERR409730.2_HWI-D00200:86:H0T6GADXX:1:1101:1341:1988.1
NATACTATGATGACTTTATGAGTATATCTTCTTCCTGGAGTTAGAAATTGCTTGACTCTTATATTTCATC
+
#4=DFFFFHHHHHJJJJJJJJJJJJJJJJJJJJJJJJJJJIHIJIJJJJJJJJIJJJJJJJIJJJJJJJ
```

Figure 2.8: FASTQ read example

Secondary analysis pursues the reconstruction of the original genome using the reads within the FASTQ files. This can be achieved by means of resequencing (2.4.1 Resequencing) or de-novo assembly (2.4.2 De-novo assembly) techniques. Either way, there are several pre-processing steps that must be performed before the input reads become ready to any reconstruction operation. In the same way, there can be several post-processing steps afterwards to prepare the results for the next stages. These steps may change depending on the sequencing technology, protocols, quality of the data, etc.

In a typical resequencing analysis scenario, initially the sequenced reads should be inspected to remove any kind of undesirable artifacts. Several techniques exist for this, such as read trimming (e.g. hard-trimming, quality-trimming, etc), adaptor removal (in case the selected sequencing technology has left any kind of adaptor), read error correction [Yang et al., 2013a], contamination removal (2.4.3 Other applications), and others. Afterwards, the resulting sequences are aligned against a reference genome (read mapping). The result of this stage is usually encoded as an SAM or BAM file [Li et al., 2009].

Following the mapping stage, many post-processing steps may be required to produce compliant results for downstream analysis. One should never underestimate the computational cost of these steps as they handle quite large volumes of data. For example, in standard analysis pipelines it is very common to split input FASTQ data so as to parallelize the mapping step across many computation nodes within a cluster. Due to this, resulting SAM/BAM files have to be merged and sorted to produce a compliant and coherent mapping file to feed the following steps of the

pipeline [Li et al., 2009]. Not only that, but the results may be stored in secondary storage for future usage. In that case, the resulting SAM/BAM files are usually compressed (e.g. BZIP or GZIP). Once again, there are many parallel tools to overcome this computationally intensive task [Roguski and Ribeca, 2016].

Though it's sometimes considered part of the tertiary analysis, variant calling has become a de-facto standard analysis procedure. For that reason, many scientists argue that it has to be categorised as a step within the secondary analysis stage. In any case, it is common that Variant Calling (VC) follows at this point of the analysis. VC has no other purpose than to identify the variants in the donor sample, not recognised in the reference genome. That is to say, derive the individual genotype from the mapping files. This process usually focuses on spotting small variants like SNP or SNV. Popular variant callers like GATK [McKenna et al., 2010] are at the same time composed of several steps like base-quality recalibration, indel-realignment, etc.

During the secondary analysis, quality control (QC) should be present not only to detect failures in any of the steps, but also to ensure the quality of the final results. These procedures can be introduced at three different points within this stage: Quality control of the raw data, quality control of the alignments, and quality control on variant calling [Guo et al., 2013].

Finally, tertiary analysis – often including data visualisation – diverges depending on the specific study and research questions in-hand. Broadly speaking, its main aim is to give interpretation to the results produced by the secondary analysis. Sometimes this analysis integrates data coming from multiple experiments and samples. Examples of this analysis are gene annotation, differential expression studies, alternative splicing studies, detection of rare variants, association studies etc. Often at this point, many tools for data visualisation are employed to aid further analysis (i.e exploratory analysis of sequence data). Tertiary analysis is still in its infancy and many methods and tools for it are being delimited and developed.

As shown above, current analysis pipelines follow a series of general guidelines regarding which components preform which functions and how they are to be connected. This delimits the competences of each component and allows great specialisation and improvement. It is important to bear in mind the place that the genomic mapper has in the pipeline and its main purpose. Some mapping tools try to stand out from the rest by means of incorporating increased functionality such as error correction, read trimming or quality recalibration. This leads to mappers with a more general purpose, less specialisation, and less flexibility regarding integration as modular components in a pipeline. In such a volatile field where the techniques and approaches improve and change tremendously fast, pipeline components must be kept as modular and flexible as possible. Increasing the number of functions of modern mappers should not be the focus of future developments. Improvement efforts have to stick to the lines of better performance and accuracy.

Chapter 3

Sequence alignment

Before going any further, it is mandatory to formally present the general problem of mapping sequencing reads to a genome. In this chapter, we give a thorough description of the problem as well as some of its many reformulations in different contexts and applications.

Moreover, we emphasise the duality of the problem from both the perspective of bioinformatics – sequence alignment – and that of computer science – approximate string matching. For that, we analyse in detail current sequence alignment paradigms, error models and the potential conflicts that can arise from it. Then, using a formal approach, we introduce the problem of approximate string matching together with basic definitions and concepts of the problem.

Additionally, we present and justify the relevance of genome mapping accuracy. In this way, we introduce current methods for benchmarking mappers and the quality of its results. On the top

of that, we propose a novel approach towards resolving conflicts during mapping and assessing the quality of the alignments. Finally, we explore the limits of mapping accuracy using the concept of genome mappability.

3.1 Sequence Alignment. A problem in bioinformatics

Generally speaking, *sequence alignment* denotes a series of bioinformatic algorithms whose purpose it is to establish homology between genomic sequences. In order to do so, sequences are aligned –lined up with each other– so that the degree of similarity of suitably matching parts of the different sequences, in the case of *local alignment*, or the degree of similarity of the full sequences, in case of *global alignment*, is maximized according to a given string distance function or score. Sequence alignment is a general term and has multiple embodiments (as in pair-wise sequence comparisons, multiple sequence alignment, construction of evolutionary trees, etc).

3.1.1 Paradigms

In the context of this thesis we focus mainly on the problem of mapping DNA sequences against a reference genome (a.k.a *mapping to a genome*). Broadly speaking, mapping to a genome aims to retrieve all positions from a given reference (collection of chromosomes, contigs or so) where a given DNA sequence (relatively small and several orders of magnitude smaller than the reference) aligns (using a given distance function and error tolerance). Ultimately, the goal is to retrieve the actual genomic location which has generated the short sequenced read at hand. This represents the most common case of use and we traditionally refer to it as to *searching for the best (or "true") match*.

It is important to highlight that in general there will be more than one alignment, i.e. more than one location that might have plausibly originated the read (case of *multimapping read*). So after finding all matching locations as stated before, the best candidate/s, that is, those showing highest sequence similarity to the original read, must be selected. In cases where no plausible locus of origin is found, or multiple equally likely candidates are found, the read must be flagged as ambiguous and possibly discarded. We often refer to this disambiguation process as *multimap resolution*.

In fact certain applications do not require this final step of resolving the true source location – they simply require all the locations in the genome the read may have been originated from (i.e. all alignments to the input sequence having an error smaller than a pre-defined toleration value). If one aims to retrieve all the equally distant matches having a minimum distance we say that one is performing an *all-best* search. If one simply wants to retrieve all matches within some error threshold we refer to an *all-matches* search [Holtgrewe et al., 2011]. Though the use of the latter is less frequent, some real-life applications (for instance the study of *copy number variations* or CNVs) do require searching for all matches.

Whether we aim for a "true-match", an "all-best" or an "all-matches" search, in general sequence alignment algorithms are faced with the challenge of retrieving all requested matching locations below a certain error tolerance. We denote as *complete* –or *exact*– algorithms that return all existing matches without missing solutions. On the contrary, if the algorithm may miss alignments potentially relevant to the application at hand we denote it as *incomplete*, *inexact* or *heuristic*. It is well known that most sequence alignment algorithms are based on heuristics in order to obtain faster running times at the expense of possibly losing some valid alignment.

Whether these setups are acceptable trade-offs or if they are of use for some application is an open debate. What is really important is to acknowledge that missing plausible solutions might deflect the results. For example, mistakenly reporting as a true-match a duplicate region in the reference, modifying mapping coverage results due to the lack of some matches or just reporting matches with convoluted alignments instead of least distant and simpler ones. These behaviours should not be overlooked as this could affect the downstream analysis. It is known than some widely used tools pick up at random matches in case of conflict – some of them manifestly reporting this behaviour and documenting it properly [Li et al., 2008b; Li and Durbin, 2009; Langmead, 2010].

3.1.2 Error model

All in all, sequence alignment is fundamentally about assessing homology between sequences. It is trivial to determine the homology of two sequences when both are the same (exact match). However, when it comes to similarity allowing errors, it is necessary to understand the very nature of the biological events that can transform a given sequence into another.

Broadly speaking, an error model is defined by what is considered to be an error (error event) and a function to score them (distance function). Given two strings (w and u) an alignment is a combination of error events that can transform w into u (or vice versa). From all the possible alignments of w into u, those with the least distance are called optimal alignment (i.e. whose error events score least using the given distance function). In a way, optimal alignments aim to explain how to transform sequences with the least possible of errors. Note that an error model induces an optimization problem to transform w to u – and vice versa – using a combination of error events that minimizes the distance function (alignment function).

Error events are the most basic modifications that can transform a sequence into another sequence. They represent the most common biological events that can naturally explain sequence transformations. In this way, they model chemical changes that can change one nucleotide into other e.g. sequencing errors, duplication error, variations, etc. Most common error events are substitutions (a.k.a mismatches; one sequence of nucleotides turned into another) and single nucleotide indels (insertions and deletions of single nucleotides). However, in certain situations other variations are proposed. For example, the swapping of two adjacent nucleotides (translocations).

For a given collection of error events, a distance function scores them to measure relative distance between sequences and measure homology (e.g conservation of biological sequences) or divergency (e.g degree of error). There is a wide variety of distance functions to choose from depending on the specific context of application.

Mismatch distance allows only for substitutions and scores each one using a penalty 1 – in the simplified definition – irrespectively of the nucleotide substituted. The resulting distance is equal to the total number of nucleotides mismatching in each sequence. This distance function is limited to sequences of the same length.

Episodic distance allows only for insertions with a penalty of 1. Unlike the rest of the distances presented here, this distance is not symmetric (i.e it may be possible to convert one sequence into another but not vice versa).

Longest common subsequence distance or **Indel distance** allows only for insertions and deletions (but not for substitutions). This distance function scores each single nucleotide indel with a unitary penalty. Thus, the indel distance between two sequences is equal to the total number of nucleotides that must be deleted and inserted to transform one sequence into another.

Edit distance, as opposed to the above, also allows for substitutions. In general, this distance allows for any single edit operation (i.e. substitution, insertion, and deletion) penalizing unitarily each one. So that the resulting distance is equal to the total number of edit operations that it takes to transform one sequence into another. This is one of the most commonly used distance functions due to its simplicity, versatility, and because there are many efficient alignment algorithms based on it. **Damerau–Levenshtein distance** is a variation in which transpositions (swaps of two adjacent characters) are also allowed with a penalty 1. Another variation in the **event distance** penalizes indels with a score 1 irrespective of the length of the indel. This variation tries to model for contiguous gaps that might have been generated at the same point and thus should not be penalized more than once.

General scoring matrixes. In general, it is possible to define a function to assign a penalty to each nucleotide substitution, insertion of any given length, and deletion of any length. These tailored scoring matrixes aim to statistically model the frequency of certain errors. For example in DNA, substitution mutations are of two types: transitions (i.e. interchanges of purines (A $\leftrightarrow$ G) or pyrimidines (C $\leftrightarrow$ T)) and transversions (i.e. interchanges of purine for pyrimidine bases). It is known that transition mutations are much more frequent in nature. To model this, transversions would be penalized more than transitions in the scoring matrix. As a result, optimal alignments following this scoring scheme would favour transitions, leading to a more plausible transformation between sequences. Another good example is the so-called Smith-Waterman-Gotoh distance where gaps are scored according to their length.

A useful insight regarding error models is to understand that the error does not necessarily have to be placed in one of the sequences. In a more general understanding of the problem, the error events are just transformations between two sequences. Hence the term transformation operations seems more suitable. In principle, none of the sequence has to be erroneous and alignment between both simply leads to an understanding of its structural similarities.

3.1.3 Alignment classification

Depending on the distance function, and how it scores error events with respect to their relative position, the shape of the optimal alignment may be different. There is a widely accepted classification of alignment algorithms depending on how they score deletions at the ends of the sequences – and therefore the shape of the alignment induced. 3.1 (Types of sequence alignment) introduces the main types of alignment in use.

Global alignment

Most common alignment algorithms target global alignments which are considered the most natural form of pair-wise alignment. In this case, deletions/insertions at the ends of both sequences score as in any other position in the sequence. We say that the ends are not free and consequently must be aligned. For that reason they are also known as “end-to-end” alignments. As a result, the final shape of the alignment spans throughout both sequences trying to match them whole.

Local alignment

By contrast, in local alignments we say that ends are free. That means that deletions at the beginning or end of the sequence do not count towards the distance function and thus trimming the sequence ends is not penalised. These alignments tend to favour highly homologous alignments between local parts of the sequence as opposed to distant – and noisy – end-to-end alignments of the sequence.

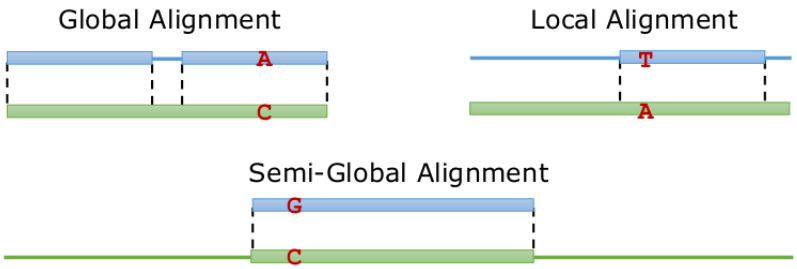


Figure 3.1: Types of sequence alignment

Semi-global alignment

In cases where one of the sequences is much larger than the other, it makes sense to allow the smaller sequence to align end-to-end to a local region of the large sequence. This is a very common case when aligning relatively short HTS sequences to a relatively large reference genome. In this case, ends are free but only on the large sequence. Hence, trims on the large sequence are not taken into account (but they are on the smaller).

Other types of alignment

Alternatively, there are many other alignment denominations that come up naturally in different applications to better explain relationships between sequences. Figure 3.2 (Alternative types of sequence alignment) depicts some of them. A common example of these are overlapping alignments. In this case, a suffix of a sequence aligns against the prefix of the other. Therefore, overlapping alignments allow for free-begining or free-end gaps at the sequences respectively. These type of alignment are commonly used for initial stages of HTS assembly to detect overlaps between input sequences and reconstruct the original donor genome by jigsawing the relatively small sequences. But also, in the case of paired-end alignment, in some situations the ends are aligned against the reference so that the ends overlap in the same region (denoting that the sequenced has twice read the overlapping region of the sequence).

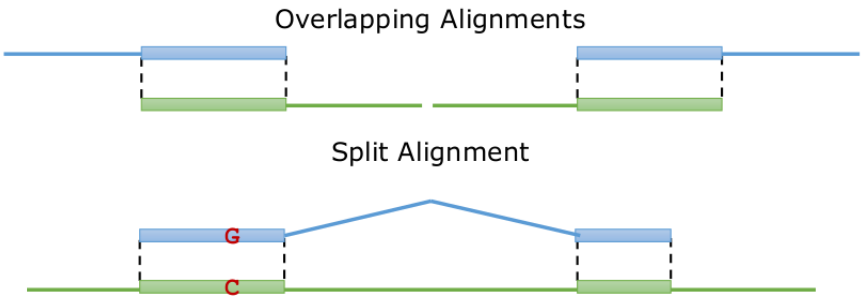


Figure 3.2: Alternative types of sequence alignment

With a more specific biological process in mind, split alignments aim to model RNA splicing events where certain parts of the genome – introns – are removed and the remaining ones – exons – are joined together (ligated). These alignments allow large gaps in the middle with no penalty. As a result, this favours proper modelling of large sections of the genome being removed. Hence, the resulting alignments can spawn over distant regions of the genome.

3.1.4 Alignment conflicts

Despite its apparent simplicity, the problem of sequence alignment hides many different obstacles which make the whole problem difficult to define. Firstly, there is a lack of a clear notion of "best-match". In the same way, there is no consensus about which distance function best suits the problem (e.g. mismatch distance, edit distance, biological custom score matrices, etc). Additionally, there are discrepancies in how to determine if two matches close to each other are different. Figure 3.3 (General conflictive alignments) illustrates some of these sequence alignment conflicts.

As we can see in the rightmost example, two different regions of the genome are plausible sources of the sequence read. The first one aligns to the read with two mismatches and the other with a deletion in the read of two bases. Depending on the distance function we use to evaluate both alignments, we can draw different and opposite conclusions. If we were to use a biological inspired error model, we could say that the first match is less plausible than the second one. In that case, two mismatches are less likely to happen than a single error event of two bases deleted – that also could be a byproduct of a homopolymer sequencing error. However, an error model based on edit distance could score both matches equally (i.e. two error events on both matches). But also, different biologically inspired scoring matrices could lead to completely different results; depending on how much two mismatches score against a deletion of two consecutive bases.

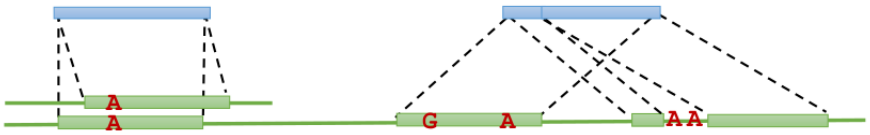


Figure 3.3: General conflictive alignments

Another example of conflict is given by the leftmost example of Figure 3.3 (General conflictive alignments). In this case, the reads match twice to the reference in very close locations. So close, that it is difficult to tell if there are two different matches or just one. Some approaches would say that these two alignments are incompatible and the whole sequence read should be discarded as ambiguous. On the other hand, both alignments lead to the same alignment result which is a mismatched 'A' nucleotide in the same position against the reference. Therefore, other approaches may support one of them being correct – if not both.

There are several other types of conflict, however most of them are related to assumptions made towards biological and biochemical behaviour in sequencing. Closely related with the alignment types, Figure 3.4 (Biological inspired conflictive alignments) depicts some conflicts that can arise in conventional sequencing applications. The first case depicts a global alignment of the sequence read, the second a local alignment of the same read, and the third a splitted alignment.

The global alignment, forced to perform an end-to-end alignment, reports three mismatches and several gaps meanwhile the local alignment trims the most divergent part of the read in favour of a much cleaner alignment. Most applications would tend to favour the local alignment against the global one simply because it leads to less error events that can more easily be explained. Either way, both alignments reveal that there is part of the alignment for which we cannot construct a plausible alignment. It is very important to acknowledge that there is some uncertainty inherent to both alignments.

In turn, the part of the read previously determined as divergent, might have been generated from another part of the genome (e.g. generated from a RNA splicing event or a chimera). The third alignment – split alignment – leads to a result in which the whole read is mapped with a long gap in the middle but with few mismatches and indels. Whether this third alignment is an artefact depends on the context of the application (e.g. some organisms or eukaryotic genes do not have introns and the alignment thus becomes bogus). This leads to the conclusion that specifics of a case-of-use cannot be captured by general purpose alignment algorithms and must deal with a precise model in mind. Otherwise, the results may be erroneous.

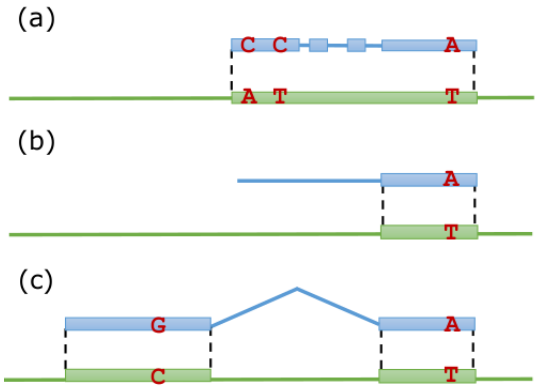


Figure 3.4: Biological inspired conflictive alignments

These problems should be tackled from a formal perspective which looks for a standardised and well-defined behaviour. In case of ambiguities, it is always better to report uncertainty rather than output the first match at random – which for some tools is a predefined behaviour. What is more important is that aligners should be sensitive enough to discover the plausible alternative alignment for a given read and detect ambiguous cases.

Conflict resolution is one of the most important parts of modern read mappers. It is crucial to understand the impact it has on the downstream analysis and how bad quality results from mapping can lead to the wrong high-level analysis results. For instance, a very sensitive mapping without a quality conflict resolution logic could produce the wrong mapping results (i.e. false positives) containing artefacts that could ultimately produce false SNPs/SNV calls. Conversely, an insensitive mapper will miss mappings and in turn, will prevent calling possible relevant variants. In addition, leaving too many sequenced reads unmapped, will disturb other future measures like coverage, CNV, etc.

3.2 Approximate String Matching. A problem in computer science

For many decades within the field of computer science, the problem of sequence alignment has been formally studied and generally known as approximate string matching. *Approximate string matching* (ASM) refers to finding strings that match a pattern approximately (i.e allowing for errors). Extensive research has been conducted in this field since the 1960s [Hall and Dowling, 1980; Navarro, 2001; Fonseca et al., 2012; Jokinen et al., 1996], mostly in the context of text retrieval (e.g database searches, spell checking, file comparisons, etc). As a result, a great many algorithms and techniques have been proposed to tackle the problem.

In the past decade however, there has been a progressive movement away from formal methods towards tailored heuristic techniques. Quite often, these approaches justify their suitability with statistical observations derived from experimental benchmarks (very often focused on specific applications and input datasets). As a result, many tools have quickly become obsolete [Langmead et al., 2009; Li et al., 2008b], being unable to scale up and keep pace with newer technologies and applications. In the ever-changing field of bioinformatics, where methods and protocols can change in less than a year, analysis tools cannot rely soley on a handful of positive results drawn from specific biological simulations. Sequence alignment problems have to be tackled from the formal perspective of ASM and research must aim to derive algorithmic guarantees in order to produce robust mappers suitable for real-life datasets.

In fact, one should understand that such a thing as a completely biology-specific mapper does not exist. All alignment tools ultimately rely on techniques borrowed from ASM which are tailored to perform searches in biological sequences. However, they sometimes incorporate biological models into their algorithms so as to better capture the requirements of their use.

Another important point is the fact that so far, most of the research focus in the field of ASM has been placed on a parameter range which is quite far away from that which is typically needed by current sequencing technologies. For example, classic ASM literature would consider a regular pattern to be around 20 characters long, while a long pattern could take up to 64 characters (mostly based on textual data and language words). In contrast, modern sequencing technologies produce data with very different characteristics. The most used short-read technologies produce sequences in the range 75-300 nucleotides, while a long read can be up to several thousand nucleotides long. In addition, the alphabets typically considered by ASM algorithms would comprise 26 symbols (the regular English alphabet) plus numbers, punctuation and symbols, while the DNA alphabet typically has only 5 letters (that is the four bases A, C, T, G plus an additional symbol, typically N, to model uncalled/unknown bases). Furthermore, ASM usually assumes large error rates (up to 50% and sometimes even more), whereas in short-read mapping one typically considers errors smaller than the 5% (but some long-read technologies can get to 30%). Yet the most striking difference is probably the size of the reference text used. In bioinformatics, genome references can be several Gbases long (for instance the human genome is approximately 3 Gbases). On the other hand, classical problems in ASM usually deal with databases of several megabytes – an order of magnitude smaller like word databases.

ASM is a very general field in computer science and has many formulations. Depending on the context, ASM problems can be easily addressed with lightweight algorithms or reveal the very intractable nature of ASM as NP-complete problems. This section focuses on definitions and techniques related with mapping sequencing reads. In this spirit, the following sections introduce the problem of mapping from the formal perspective of approximate string matching, and several formal definitions used later on to introduce and formulate ASM techniques.

3.2.1 Stringology

In the following sections, we use s, p, t, x, v, w, y, z to represent arbitrary strings, and a, b, c, d to represent individual characters. Letters written one after the other represent concatenations.

Definition 3.2.1 (Alphabet)**.** An *alphabet* $\Sigma = \{a_0, ..., a_i\}$ is a finite ordered set of symbols (characters).

Remark. We will indicate the length of the alphabet as $|\Sigma| = \sigma$.

Remark. Without loss of generality, we only consider finite alphabets. Note that the algorithms we present could be adapted to infinite alphabets as they operate on finite strings with a limited number of symbols (which can be previously analysed and used as input alphabet).

Definition 3.2.2 (String)**.** A string (or word) $w = w_0...w_{|w|-1}$ over Σ is a finite sequence of symbols $w_i \in \Sigma$.

Remark. Let $|w|$ denote the length of the string $w \in \Sigma^*$

Remark. Let $w_i \in \Sigma$ denote the i-th character of the string $w \in \Sigma^*$, for an integer $i \in \{0..|w| - 1\}$

Remark. Let ϵ denote the empty string

Definition 3.2.3 (Q-gram/K-mer)**.** In general, we denote as q-gram or k-mer to any string of length q or k respectively.

Definition 3.2.4 (Substring)**.** We denote $w_{i..j} = w_i, w_{i+1}, .., w_j$ as subtring of w (also referred as subsequence).

Remark. If $i > j$ then the subtring is equal to ϵ

Definition 3.2.5 (Prefix)**.** A substring $w_{i..j} = w_i, w_{i+1}, .., w_j$ is a prefix of w iif $i = 0$

Definition 3.2.6 (Suffix)**.** A substring $w_{i..j} = w_i, w_{i+1}, .., w_j$ is a suffix of w iif $j = |w| - 1$

Definition 3.2.7 (Lexicographical order)**.** Lexicographical order between two string $w = w_0...w_{|w|-1}$ an $v = v_0...v_{|v|-1}$ is defined as $w \leq v \Leftrightarrow w_i \leq v_i$,

3.2.2 String Matching

In the context of ASM there are certain string denominations that denote the input of many algorithms. Namely, the reference text against which sequenced reads are mapped is denoted as text. Therefore let $T \in \Sigma^*$ be a text of length $n = |T|$. In the same way, sequence reads are referred to as patterns. So, let $P \in \Sigma^*$ be a pattern of length $m = |P|$. In the same way, let $e \in \mathbb{R}$ be the maximum error allowed and $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ be a distance function.

Definition 3.2.8 (Occurrence)**.** A string w occurs in a longer string x iff w is a substring of x.

Remark. Let be $occ(w, x)$ the function occurrences count that returns the total number of occurences of the string w in x.

Definition 3.2.9 (Exact string matching)**.** Given T, P, e, and $d()$, exact string matching is defined as the problem of finding all P occurrences in the text T.

Exact string matching is one of the most simple and studied problems in stringology. There is has been extensive research on the problem and many solutions had been proposed [Charras and Lecroq, 2004; Turing, 2006; Lecroq, 2007]. Nevertheless, exact matching becomes very limited in this context as it lacks modelling divergence between strings.

Definition 3.2.10 (Match)**.** Given T, e, and $d()$, a string $m_i = T_{i..j}$ is a match (or matches P) iff $d(P, T_{i..j}) \leq e$

Remark. The position i where P matches T is denominated the matching position.

Definition 3.2.11 (Approximate string matching)**.** Given T, P, e, and $d()$, approximate string matching is defined as the problem of finding all matching positions $M = \{m_0, .., m_n\}$ between the text T and the pattern P.

Remark. Exact string matching can be defined in terms of the approximate string matching problem in the case of $e = 0$.

A common technique to enhance string searches is to preprocess the inputs of the problem so as to enable fast access to specific substrings. This preprocessing is usually applied to the input text (i.e. reference genome) to produce an index text (or just index). However, there are some approaches that also consider indexing the patterns (i.e. sequenced reads). Either way, depending on whether the search is performed using an indexed text or a plain text, we refer to the search as indexed or online respectively. Over the years, many index preprocessing algorithms have been proposed [Ukkonen, 1992; Apostolico, 1985; Gusfield, 1997; Weiner, 1973; McCreight, 1976]. (in chapter 4 we introduce the main data structures and algorithms used for indexing).

Definition 3.2.12 (Online string matching)**.** Online string matching is the problem of string matching – exact or approximate – when the input T is a plain sequence of symbols.

Definition 3.2.13 (Indexed string matching)**.** Indexed string matching (a.k.a. offline string matching) is the problem of string matching – exact or approximate – when the input I is an index (preprocessed data structure of the text T).

3.2.3 Distance metrics

Definition 3.2.14 (Transformation operation (a.k.a. error operation))**.** Transformation operation (also operation or rule), is defined as a function $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$, such that $\delta(x, y) = c$ denotes the operation to transform string x into the string y with the associated cost c.

Remark. Once an operation is applied to a substring x into y, no further operations can applied to y.

Remark. Common operations used in diverse applications are:

- Insertion: $\delta(\epsilon, a)$
- Deletion: $\delta(a, \epsilon)$
- Substitution: $\delta(a, b) \mid a \neq b$
- Transposition: $\delta(ab, ba) \mid a \neq b$

Definition 3.2.15 (Alignment)**.** Alignment between two strings v and w is defined as a sequence of operations $A = \{\delta_0, .., \delta_n\}$ that transforms v into w

$$v \xrightarrow{A} w$$

Remark. In some contexts, this sequence of operations is also denoted transcript (Gusfield, 1997).

Definition 3.2.16 (Error rate)**.** Let $\epsilon = e/|P| = e/m$ be the error rate.

Remark. By definition, $0 \leq \epsilon \leq 1$

Definition 3.2.17 (Distance function)**.** A distant function $d(x, y)$ between two string x and y is the minimal cost alignment that transform x into y. That is,

$$d : \Sigma^* \times \Sigma^* \to \mathbb{R}$$

$$d(x, y) = \min \left\{ \sum_{\delta_i \in A} \delta_i \;\middle|\; v \xrightarrow{A} w \right\}$$

Definition 3.2.18 (Symmetric function)**.** Let $d(x, y)$ be a distance function. If for each operation $\delta(x, y)$ exist a reciprocal operation $\delta(y, x)$ at the same cost (i.e. $\delta(x, y) = \delta(y, x)$), then the distance $d(x, y)$ is symmetric (i.e. $d(x, y) = d(y, x)$).

Definition 3.2.19 (Metric space)**.** Let $d(x, y)$ be a distance function and the following holds

- $d(x, y) \geq 0$ (Non-negativity)
- $d(x, y) = 0 \Leftrightarrow x = y$ (Identity of indiscernibles)
- $(x, y) = d(y, x)$ (Symmetric)
- $d(x, z) \leq d(x, y) + d(y, z)$ (Triangle inequality)

then the space of strings form a metric space.

Definition 3.2.20 (Neighbourhood)**.** Given a distance function $d(x, y)$, we define the neighbourhood of a string s at distance e as

$$N_d^e(s) = \{n_0, .., n_n \mid d(n_i, s) \leq e, \forall n_i\}$$

I Classical distance metrics

Besides these general distance definitions some distance functions used in many ASM applications are particularly interesting. In the following section we present in detail those most relevant so as to help further understand the algorithms presented later on.

Definition 3.2.21 (Hamming distance or mismatch distance)**.** Let $\delta_m(v, w)$ be the Hamming distance ([Hamming, 1950]) between two strings defined as:

$$\delta_m(v, w) = \begin{cases} \infty & \text{, if } |v| \neq |w| \\ \sum_{i=0}^{|v|-1} (v_i \neq w_i) & \text{, otherwise} \end{cases}$$

Remark. Approximate string matching according to the Hamming distance is known as the *k-mismatches problem* [Galil and Giancarlo, 1989].

Definition 3.2.22 (Levenshtein distance or edit distance)**.** Let $\delta_e^{v,w}$, or simply δ_e, be the Levenshtein distance [Levenshtein, 1966], between two strings v and w, defined recursively as $\delta_e(|v|, |w|)$

$$\delta_e(i, j) = \begin{cases} \max(i, j) & \text{if } i = 0 \vee j = 0, \\ \min \begin{cases} \delta_e(i-1, j) + 1 \\ \delta_e(i, j-1) + 1 \\ \delta_e(i-1, j-1) + (v_i \neq w_j) \end{cases} & \text{otherwise} \end{cases}$$

Remark. Approximate string matching according to the Levenshtein distance is also known as the *k-differences problem.*

II General weighted metrics

Besides these two basic distance functions, one can define an infinite number of other distances as variations of the edit distance. Such generalisations allow arbitrary weights –or *scores*– to be associated to each insertion, deletion or substitution operation. In general, the problem can be formulated in terms of either maximising a score or minimising the cost. Both are usually defined as a function of the error events between a pattern in a text. The score – or cost – is commonly

an integer value which models the divergence between the pattern and the text. In general terms, score functions are usually used when both similarities and differences are weighted into account, meanwhile cost functions usually focus only on the differences. These weighted functions are often generalised to allow the score of a substitution to depend on the exact nucleotide in the alphabet being removed and the exact one being added. Likewise, gaps might be weighted to score differently depending on the nucleotide inserted or deleted. More often, the cost of each gap models better the problem at hand if weighted by the exact amount of nucleotides being deleted or removed. In the context of protein alignment, for instance, alphabet weighted scores are used to model how likely an amino acid is to be replaced by another one. There has been extensive research on how to determine the most suitable of such amino acid transition matrixes based on biological evidence. The dominant scoring schemes are the PAM matrices [Henikoff and Henikoff, 1992] and the BLOSUM scores [Mount, 2008; Henikoff and Henikoff, 1993] Given that so many scores can be formulated, some general plausibility assumptions must be taken into account at the time of configuring these weighted scores matrices. For instance, if substitutions are allowed, then it is reasonable to assume that the cost of a deletion together with an insertion is greater than the cost of a single substitution. Also, note that the scores depend on the characters involved in the operation and the length of it, however it rarely depends on where the operation in the string is applied.

Besides the actual weights that a given weighted distance metric can have, there is a much more important factor that will heavily determine the final shape of the alignment. Initialisation conditions encode for the frontier cases that will lead to global alignments, local alignments, semi-global alignments. Depending on these conditions, the distance computations will allow gaps at the beginning and/or end of the pattern and/or text with no penalty whatsoever.

Definition 3.2.23 (Global weighted distance)**.** Given the substitution weight function $\tau_S(a, b)$, the insertion weight function $\tau_I(s, k)$ and the deletion weight function $\tau_D(s, k)$, let $\delta_H^{v,w}$, or simply δ_H, be the global weighted distance between two strings v and w. We denote $H(i, j)$ the maximum similarity score between a $v_{0...i}$ and $w_{0...j}$. Then the maximum alignment score $H(|v| - 1, |w| - 1)$ is given by:

$$H(i,j) = \begin{cases} H(i,0) = i \ (0 \leq i < |v|) & \text{if } j = 0 \\ H(0,j) = j \ (0 \leq j < |w|) & \text{if } i = 0 \\ \max \begin{cases} H(i-1,j-1) + \tau_S(a_i, b_j) & \text{Substitution} \\ \max_{0<k\leq i}\{H(i-k,j) + \tau_D(v_{i-k,i})\} & \text{Deletion} \\ \max_{0<l\leq j}\{H(i,j-l) + \tau_I(w_{j-l,j})\} & \text{Insertion} \end{cases} \end{cases}$$

Note that in the global weighted alignment initial conditions (i.e. $H(i, j)$ s.t $i = 0 \vee j = 0$) penalise initial insertions or deletions according to their length. For instance, $H(i, 0)$ is the maximum score to align $v_{0..i}$ against the empty string, which equals to the cost of deleting all of its characters. Likewise, $H(0, j)$ is the maximum score to align the empty string against $w_{0..i}$, which equals to the cost of inserting all the characters characters in $w_{0..i}$.

In a similar fashion, if we wanted to model semi-global alignment, we would incorporate to the previous definition the fact that any starting or ending deletion in the pattern is not penalised. This is commonly know as ends-free alignment.

Definition 3.2.24 (Semi-global weighted distance)**.** Given the substitution weight function $\tau_S(a, b)$, the insertion weight function $\tau_I(s, k)$ and the deletion weight function $\tau_D(s, k)$, let $\delta_H^{v,w}$, or simply $delta_H$, be the semi-global weighted distance between two strings v and w. We denote $H(i, j)$

the maximum similarity score between a $v_{0\ldots i}$ and $w_{0\ldots j}$. Then the maximum alignment score $H(|v|-1,|w|-1)$ is given by:

$$H(i,j) = \begin{cases} H(i,0) = 0 \ (0 \leq i < |v|) & \text{if } j = 0 \\ H(0,j) = j \ (0 \leq j < |w|) & \text{if } i = 0 \\ \max \begin{cases} H(i-1,j-1) + \tau_S(a_i, b_j) & \text{Substitution} \\ \max_{0<k\leq i}\{H(i-k,j) + \tau_D(v_{i-k,i})\} & \text{Deletion} \\ \max_{0<l\leq j}\{H(i,j-l) + \tau_I(w_{j-l,j})\} & \text{Insertion} \end{cases} \end{cases}$$

Finally, local alignments are the special case where both pattern and text are ends-free. Additionally, negative scores are forbidden. Therefore, the computation of the alignment can start at any point in the pattern and the text, leading to maximal alignments between substrings of the text and pattern.

Definition 3.2.25 (Local weighted distance)**.** Given the substitution weight function $\tau_S(a,b)$, the insertion weight function $\tau_I(s,k)$ and the deletion weight function $\tau_D(s,k)$, let $\delta_H^{v,w}$, or simply $delta_H$, be the local weighted distance between two strings v and w. We denote $H(i,j)$ the maximum similarity score between a $v_{0\ldots i}$ and $w_{0\ldots j}$. Then the maximum alignment score $H(|v|-1,|w|-1)$ is given by:

$$H(i,j) = \begin{cases} H(i,0) = 0 \ (0 \leq i < |v|) & \text{if } j = 0 \\ H(0,j) = 0 \ (0 \leq j < |w|) & \text{if } i = 0 \\ \max \begin{cases} 0 & \\ H(i-1,j-1) + \tau_S(a_i, b_j) & \text{Substitution} \\ \max_{0<k\leq i}\{H(i-k,j) + \tau_D(v_{i-k,i})\} & \text{Deletion} \\ \max_{0<l\leq j}\{H(i,j-l) + \tau_I(w_{j-l,j})\} & \text{Insertion} \end{cases} \end{cases}$$

In the biological literature there is an ever present misconception in terminology. Global alignment is quite often referred to as Neddleman-Wunch alignment after the authors that first proposed an algorithm to compute global pair-wise sequence similarity. In the same way, local alignment is usually referred to as Smith-Waterman after the authors who first introduced this variation. Nevertheless, these terms are often confused and the solutions proposed can sometimes be misleading. Note that the original algorithm proposed by Neddleman-Wunch run in $\mathcal{O}(n^3)$ is rarely used. Meanwhile, there are known solutions that run in $\mathcal{O}(n^2)$. Likewise, Smith-Waterman is not the only method available to compute local alignment and the term is sometimes used to inaccurately specify any method related with local alignments.

The final movement towards accurate modelling and scoring alignment comes with a more natural model of the gap cost. Many proposed distances in the context of computer science weight a gap proportionally to its length. However, in the context of biology, the cost of gap may not increase linearly with its length. In some cases, a logarithmic decrease in its cost would model better these error events (i.e. $\tau(k) = \alpha + \beta\log(k)$). However, this approach would lead to impractical algorithms. To overcome this problem, affine gap penalties are often used to mimic this behaviours. Intuitively, once a gap is "opened" (i.e. created or establish in the alignment), the cost of extending it (i.e. enlarging the gap) should be inferior (i.e. $\tau(k) = \alpha + k\beta$). Algorithms computing this type of distance are often referred to as Smith-Waterman-Gotoh after the authors that first proposed the model. Among other distances, this one is often preferred by biologists and is thus widely used and implemented with many popular alignment tools. Such is the popularity of this metric, that it is not uncommon to see a mapping tool preferred simply because it employs gap-affine penalties, even if its run-time is several times slower than other available tools.

Algorithm 1: Edit distance computation using dynamic programing

input : v, w strings
output: edit distance between v and w

begin
  for $i = 0$ **to** $|v|$ **do**
    $\delta_e[i, 0] \leftarrow 0$
  end
  for $j = 1$ **to** $|w|$ **do**
    $\delta_e[0, j] \leftarrow 0$
  end
  for $i = 1$ **to** $|v|$ **do**
    for $j = 1$ **to** $|w|$ **do**
      $$\delta_e[i, j] \leftarrow \min \begin{cases} \delta_e[i-1, j] + 1 \\ \delta_e[i, j-1] + 1 \\ \delta_e[i-1, j-1] + (v_i \neq w_j)?1 : 0; \end{cases}$$
    end
  end
  return $\delta_e[|v|, |w|]$
end

III Dynamic programing pair-wise alignment

For the sake of completeness and to illustrate one of the most fundamental algorithms for pair-wise alignment, we present the solution based on dynamic programming. This approach has been rediscovered over the years within the context of several different areas. Many studies of this method have produced many fundamental theoretical bounds for the problem and suggested several approaches to avoid unnecessary computations. However, in the eyes of this author, its most relevant contribution has been to conduct and inspire many practical algorithms based on properties highlighted by these dynamic programming approaches. Despite not being the fastest approach, it is without a shadow of a doubt the most flexible solution and the easiest to adapt to the many different distance metrics.

Following Bellman's principle of optimality, the minimum edit distance (definition 3.2.22) between two strings – v and w – can be computed using a dynamic programing algorithm. To avoid repeated computations given by the recursive formula, recursive calls are organised in a dynamic programing table of size $(|w| + 1) \times (|v| + 1)$. Algorithm 1 shows how to compute this table. For instance, in following example we show the computation of the full dynamic programing table aligning the pattern "GAGATA" against the text "GATTACA".

$$\delta_e(\text{"}GAGATA\text{"}, \text{"}GATTACA\text{"}) = \begin{bmatrix} & - & G & A & T & T & A & C & A \\ - & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ G & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ A & 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ G & 3 & 2 & 1 & 1 & 2 & 3 & 4 & 5 \\ A & 4 & 3 & 2 & 2 & 2 & 2 & 3 & 4 \\ T & 5 & 4 & 3 & 2 & 2 & 3 & 3 & 4 \\ A & 6 & 5 & 4 & 3 & 3 & 2 & 3 & 3 \end{bmatrix}$$

The dynamic programing algorithm fills each cell based on the content of the upper, left and upper-left neighbours. In this way, it can progress column-wise from the left most column (initial conditions) to the last column. Likewise, it can perform the computations row-wise from the upper most row to the last row. Either way, this approach runs in $\mathcal{O}(|v||w)$. However, if only the distance is required, computations can be confined to just one column – or row – and progress forward until the last column – or row – is computed and the alignment distance is known. As we can see in the example, the minimum edit distance between the string is 3 edit operations. Also, the path of the computations that lead to the minimum is highlighted in blue.

$$\Delta_e(\text{"}GAGATA\text{"},\text{"}GATTACA\text{"}) = \begin{bmatrix} & - & G & A & T & T & A & C & A \\ - & \bullet & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ G & 1 & \nwarrow & \leftarrow & 2 & 3 & 4 & 5 & 6 \\ A & 2 & 1 & \nwarrow & \nwarrow\!\!\leftarrow & 2 & 3 & 4 & 5 \\ G & 3 & 2 & 1 & \nwarrow & \nwarrow\!\!\leftarrow & 3 & 4 & 5 \\ A & 4 & 3 & 2 & 2 & 2 & \nwarrow & 3 & 4 \\ T & 5 & 4 & 3 & 2 & 2 & 3 & \nwarrow & 4 \\ A & 6 & 5 & 4 & 3 & 3 & 2 & 3 & \nwarrow \end{bmatrix}$$

In the previous example shows, the path of the computations that lead to the minimum is highlighted in blue. In order to recover the sequence of operations that aligns one string into the other, we simply have to follow the path that updates each cell from $\delta_e(6,7)$ to $\delta_e(0,0)$ (in this context usally called edit transcript). This process in known as traceback and, as we can see, can lead to multiple possible paths – all of them equally valid.

3.2.4 Stratifying metric space

As explained above, depending on the error model properties, they define a metric space. This space of strings can be stratified not only to organise the search space, but also to guide the search towards deeper and more distant solutions. In this way, we divide the search space into strata. In other words, we group together all possible strings at a fixed distance of the pattern P.

Definition 3.2.26 (Search Stratum)**.** Given P, e, and $d()$, a search stratum is the set of all strings from the neighbourhood of P at distance e.

$$S_e^P = \{w_i \in N_d^e(P) \mid d(P, w_i) = e\}$$

Definition 3.2.27 (Search Space)**.** Given P, e, and $d()$, the search space is defined by the $e+1$ search strata.

$$\mathbb{S}_e^P = S_0^P, \ldots, S_e^P$$

However, not all strings in the search space necessarily occur in the reference text T. For that reason, we would like to focus only on the actual matches towards classifying them. For that reason, we group all the possible matches at a given distance e from the pattern P in a match stratum.

Definition 3.2.28 (Match stratum)**.** Given T, P, e, and $d()$, a match stratum is the set of all matches between T and P aligning with distance e.

$$M_e^{T,P} = \{m_i = T_{i..j} \mid d(P, m_i) = e\}$$

Similarly, all the match strata, up to a given distance e, defines the matches space.

Definition 3.2.29 (Matches space)**.** Given T, P, e, and $d()$, the matches space is define by the $e+1$ alignment strata.

$$\mathbb{M}_e^{T,P} = M_0^{T,P}, \ldots, M_e^{T,P}$$

Figure 3.5 depicts the search space for the word P=GATTACA grouped by strata for Hamming and Levenshtein distances (i.e. the neighbourhood of P classified by distance). However, as the figure shows, in most cases not all combinations occur in the text and only a few of them happen to be matches. Depicted in green, we can see the actual matches for an hypothetical text T.

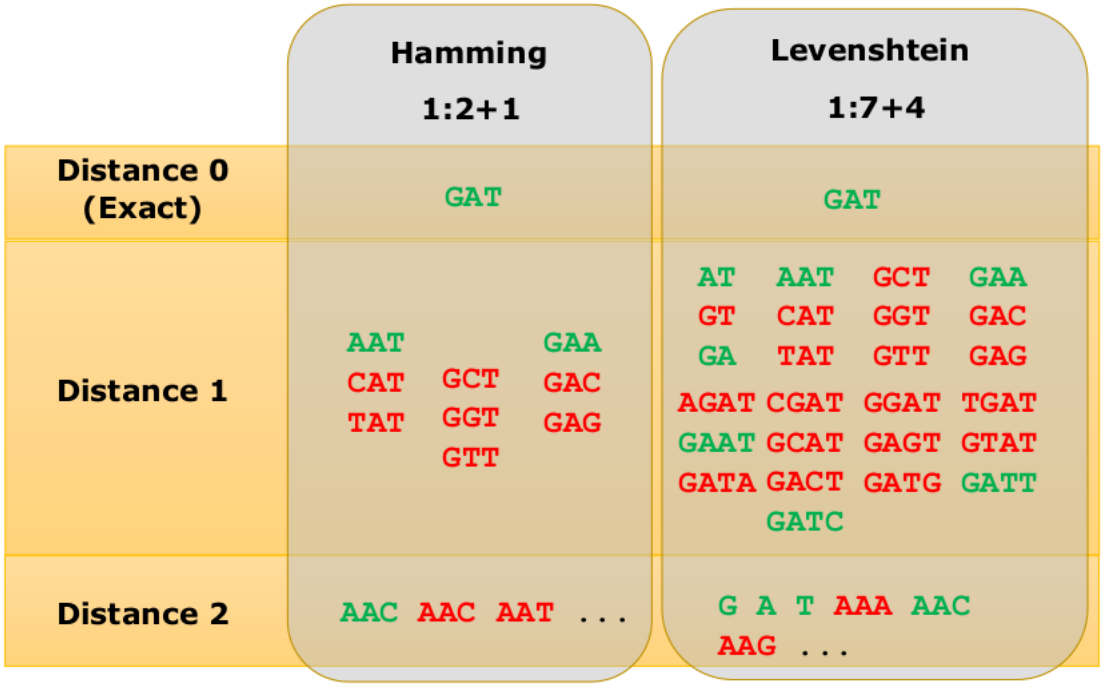


Figure 3.5: Stratified search space for P=GATTACA

As we can see, depending on the distance metric employed, the size of each stratum grows differently allowing various possible matches. Under distances like Hamming or Levenshtein, it is well known that the cardinal of each stratum grows exponentially [Myers, 1994; Navarro, 2001](Navarro, Myers). Because of this, some mappers will fail to capture all the matches in a certain stratum. Sometimes due to heuristic approaches and others because it is just unnecessary. In both cases, if a given matches stratum $M_e^{T,P}$ contain all the matches of P against T at distance e, we say the stratum is complete (i.e. contains all possible matches). Otherwise, the stratum is incomplete. Consequently, we can say that a mapper – or alignment algorithm – is complete up to e if all strata reported up to e is complete. Otherwise, the mapper is incomplete.

Strata notation

Matches space is of a great help not only in understanding the results from mapping tools, but also in resolving multimap conflicts. For our convenience, we will introduce the strata notation so as to aid following explanations. Strata notation can depict the cardinals of each stratum in a search or matches space. Then, for a given $\mathbb{S}_e^P = S_0^P, \ldots, S_e^P$ or $\mathbb{M}_e^{T,P} = M_0^{T,P}, \ldots, M_e^{T,P}$ we notate a stratified spaced by putting the cardinals of each of each stratum in the space in order separated by colons.

$$\mathbb{S}_e^P \rightarrow |S_0^P| : \cdots : |S_e^P|$$

$$\mathbb{M}_e^{T,P} \rightarrow |M_0^{T,P}| : \cdots : |M_e^{T,P}|$$

To denote incomplete strata, we use a plus sign (+) instead of a colon to separate complete strata from incomplete. In this way, we clearly establish the point from which a given mapper did not report all possible matches and there exists an uncertainty (i.e. there could be valid matches beyond that stratum). In this way, we can visualise how many matches are contained in each stratum of the space – and which of them are complete. For example, Hamming match space in figure 3.5 can be described as $(1 : 2 + 1)$. We can see that in the first two strata, there is 1 exact match – distance zero –, and 2 matches with a mismatch. These two strata are complete, meanwhile stratum at distance 2 has not been fully searched. For this reason, we have just found 1 match and we can not trust there is not another match in the text at distance 2.

In the last example we depicted a clear situation for which we would not be able to tell which match is the "true-match" (i.e. multimap conflict). Nevertheless, the search depth enables us to assess a conflict and consequently declare the sequence read ambiguous. However, a narrow search would not have been able to determine this conflict. For example, a search leading to $(1 + 0 : 0)$ could mistakenly derive that the exact match is unique – meanwhile we certainly know that it is not. This example should raise awareness of the hazards of not exploring enough strata completely and how complete mappers are of the upmost relevance in order to accurately determine conflictive cases.

3.3 Genome mapping accuracy

In the context of mapping benchmarking, evaluation of accuracy plays a fundamental role. Not only does it help to improve the design of sequence alignment algorithms, it also plays a crucial role in quality control (QC) steps taken as part of HTS analysis pipelines. More generally, it would be hard for downstream tools to do their job properly without being able to obtain and incorporate this information into their inner workings.

However, as explained above, inherent ambiguities in sequence alignment set an upper limit on the amount of reads that can be accurately mapped to their source genome loci. Additionally, the lack of a ground truth – and the impossibility of computing it – makes the problem even more challenging. And last but not least, different alignment algorithms and tools use slightly different definitions of what they mean by mapping, which generally makes comparisons among them quite difficult and inaccurate.

In recent years many metrics and frameworks have been proposed for mapping quality evaluation [Guo et al., 2013; Patel and Jain, 2012; Yang et al., 2013b]. Nonetheless, most of these metrics only account for specific features of mapping, and can be misleading if not put together with complementary metrics. And although some attempts to define a common gold standard for evaluation have been made, no consensus has been reached so far. Most algorithms are still being designed with bespoke metrics in mind — often following the underlying principle that accuracy can be sacrificed at the expense of performance.

In this section we present and discuss the most commonly used metrics and frameworks for mapping quality evaluation. Afterwards, we present our improvements on them and an analysis of their impact on assessing accuracy. We aim to define a common ground not only to evaluate read mapping results, but also to guide sequence alignment algorithms towards better accuracy. Ultimately, we pursue an evaluation framework in which we can properly compute the likelihood of a given mapping being the "true-match", compare results obtained by different sequence alignment algorithms and, in general, evaluate the overall quality of mapping results.

3.3.1 Benchmarking mappers

In this section we present the most commonly used metrics and methodologies for evaluating mapping results. We also conclude, however, that if one wants to obtain robust results one should refrain from using one of these metrics in isolation as a single quality indicator.

I Quality control metrics

Traditionally, evaluation of the quality of results from read mapping has always been performed by QC tools within HTS analysis pipelines. Within these QC steps, mappings were analysed to collect basic metrics trying to derive an overall idea of the quality of the results. These metrics represent a good starting point in the evaluation of the results. In most cases, its usefulness is relegated to assessing the quality of the sequencing data itself more than the capabilities of the alignment tool used. In this way, its suitability in comparing mapping algorithms is very limited and, in some cases, can produce misleading conclusions.

Reads mapped. Probably the most straightforward metric is to count the total number of mapped reads against the reference. Total number of reads mapped is far more helpful for detecting anomalies than assessing correctness. A low number of reads mapped is a clear indicator

that something along the process has gone wrong. It may be due to errors during sequencing, reflect the lack of sensitivity of the mapping tool, or be the by-product of a bad quality or incomplete reference genome (among many other causes). However, the opposite should not always be taken as a good sign. An excessive number of reads mapped (e.g. mapping 100% Illumina-like ¿100nt reads) might be concealing an unduly sensitive mapper that allows too much divergence in order to align every single sequenced read. It is very important to understand that mapping more reads is not a synonym of a better mapping tool.

Error distribution. It is important to realise that, allowing for enough error tolerance, every read can align to any locus of the genome. In this way, many mapping tools [Marco-Sola et al., 2012; Zaharia et al., 2011; Langmead and Salzberg, 2012] base their algorithms on progressively allowing more divergence until a match is found. This can potentially lead to mapping artefacts; sequence reads that might have no biological relation with the locus they are aligned to. In order to control this, error distribution plots can help to identify these excessively divergent mappings and in turn filter them out from further downstream analysis.

Total bases mapped. Towards retrieving a plausible mapping location, many mapping tools [Marco-Sola et al., 2012; Li, 2013; Langmead and Salzberg, 2012] would switch from semi-global to local alignment search. Hence, many widely used tools will output reasonably good local alignments (i.e. local chunks of a read highly homologous to a locus in the reference) as a best mapping effort. Nonetheless, this does not mean that the reported match is likely to lead to the "true-match". But, as a matter of fact, all local alignments will be computed as mapped, hiding the fact that for a certain amount of reads the mapper was unable to find a valid semi-global alignment. For this reason, computing the total number of mapped bases can highlight cases where a high number of reads mapped is achieved by means of local alignments or large indels in the read (e.g. large trims or frequent deletions).

Insert size distribution. In paired-end protocols (2.3 Sequencing protocols), due to the size selection step during sample preparation, we dispose of a previous estimation of the insert size between the paired fragments. After mapping both ends of each sequenced template, it is expected that the resulting insert size distribution follows a normal distribution around the insert size estimated from sample preparation. Any deviation in this curve – often isolated spikes – might indicate errors in sequencing or anomalies produced by mapping. But also, comparing the curves produced by different mapping tools can reveal artefacts and biases produced by such tools.

Coverage. As a standard procedure after mapping, a coverage analysis can not only reveal biases during sequencing (e.g. low coverage on GC-low regions), but might also indicate problems during mapping. In this way, coverage profiles can reveal mapping insensitivity towards certain regions in the genome (e.g. reference regions highly divergent from the donor sample). As a result, variant detection can fail to detect putative variants in those regions during the downstream analysis. On the contrary, comparing different coverage profiles can lead to spotting excessively covered regions. Mapping tools allowing too much divergence can mistakenly align reads to incorrect regions introducing noise in downstream analysis and potentially leading to incorrect results (e.g. convoluted and artificial variants).

Others. In general, QC tools include a wide variety of basic analysis that can be used to assess the accuracy of mapping tools provided proper and careful interpretation of the results and their implications. Among other useful metrics, we can often make use of strand quantification, mismatch frequency analysis, indel frequency analysis, pair orientation analysis, etc.

II Simulation-based evaluation

However, for the sake of benchmarking mapping tools, simulated datasets are often generated so as to employ a ground truth and derive further accuracy results. Nowadays, there are plenty of sequence read simulators [Huang et al., 2012; Holtgrewe, 2010; Pratas et al., 2014] to choose from – depending on the selected technology and the characteristics we aim to model. In the case of evaluating simulated results, we can employ standard statistical measures of performance taking the mapping algorithm as a binary classification test. For the sake of clarity, we first present what we believe is the most logical classification in the context of read mapping.

		Predicted condition (Mapper)	
		Reported Match	Unmapped
True condition (Simulator)	True-Match	TP	FN
	Otherwise	FP	TN=0

Table 3.1: Mapping confusion table

True positive (TP) is when the mapper reports a match position as putative and it is actually the source position used by the simulator (i.e. true-match).

False positive (FP) (type I error) or “”false alarm” , is when the mapper reports an incorrect match position (i.e. one which differs from the “true-match” generated by the simulator).

True negative (TN), under our criteria, is when the mapper reports a read as unmapped and the read is truly unmappable. This case is not really considered in the context of our application as the simulator always produces reads proceeding from some region of the reference. For that reason – and limited to our specific domain – we will consider that the amount of true negatives is always zero.

False negative (FN) (type II error), is when the mapper cannot find the putative location of the sequenced read (i.e. it reports it as unmapped incorrectly) meanwhile there is a valid match (produced by the simulator).

In this way, we can define some metrics to assess the quality of the results produced by a mapping tools based on the number of correct and incorrect matches reported.

Definition 3.3.1 (True positive rate (TPR))**.** Also known as sensitivity or recall, measures the proportion of mapping locations that are correctly reported (i.e. where the read did indeed originate from). Note that, sensitivity quantifies avoiding false positives (FP). A mapper achieving a high TPR can potentially report a lot incorrect positions without it being reflected in this measure.

$$TPR = \frac{TP}{TP + FN} = \frac{hits}{hits + unmapped}$$

Definition 3.3.2 (False discovery rate (FDR))**.** FDR measures the amount of incorrect mapping locations reported in terms of the total number of matches mapped. Note that FDR does not account for unmapped reads. This is a conservative magnitude as a mapper could achieve a very high FDR by means of reporting only those matches with a high likelihood of being correct and leaving the rest unmapped.

$$FDR = \frac{FP}{TP + FP} = \frac{fails}{hits + fails}$$

Definition 3.3.3 (Accuracy (ACC))**.** Accuracy is formally defined as the amount of correctly reported matches in terms of the total number reads – including incorrect matches and unmapped.

$$ACC = \frac{TP}{TP + FP + FN} = \frac{hits}{hits + fails + unmapped}$$

III MAPQ score

MAPQ score was introduced within the tool MAQ [Li et al., 2008a] and later adopted as a mandatory field within the SAM specification [Li et al., 2009]. This tool's publication is one of the first to acknowledge the subtleties of accurate mapping and the conflicts that can arise from repetitive positions of the genome (i.e multimaps). In this way, they propose mapping quality Q_s [Ewing and Green, 1998].

Definition 3.3.4 (MAPQ score)**.** MAPQ is the phred-scaled probability that a read alignment may be wrong.

$$Q_T = -10 \log_{10} Pr\{\text{read is wrongly mapped}\}$$

Remark. For a $Q_s = 30$ the probability that the read is incorrectly mapped is 1 in 1000.

Based on this definition, the authors propose a method to measure the confidence that a read actually comes from the position it is aligned to (i.e MAPQ). For that, they simplistically assume that reads are known to come from the reference and that a search is complete until a given error threshold which guarantees that no relevant match is missed. Also, they assume that the sequencing errors are independent at different sites of the read. Then, given a reference sequence T and a read sequence P, they define the probability $p(P|T, q)$ of P coming from the position q as the product of the error probabilities of the mismatched bases at the aligned position.

$$p(P|T, q) = \prod_{m \in \text{Mismatches}} (10^{-Q_{\text{phred}}(m)/10})$$

For example, in the case of a read with two mismatches with phred base quality of 20 and 10:

$$p(P|T, q) = 10^{-20/10} \times 10^{-10/10} = 0.001$$

Lets assume uniform distribution over the reference $p(u|T)$. Applying Bayesian formula gives the probability of $p_s(u|T, P)$

$$p_s(u|T, P) = \frac{p(P|T, u)}{\sum_{v=1}^{n-m+1} p(P|T, v)}$$

Scaled to phred logarithmic scale, leads to the mapping quality of the alignment.

$$Q_s(u|T, P) = -10 \log_{10}(1 - p_s(u|T, P))$$

However, computing this formula is impractical as all positions of the genome have to be aligned against the read and added to the computation. Therefore, the authors of the paper propose an approximation of the formula [Li et al., 2008a].

$$Q_s = \min\{q_2 - q_1 - 4.343\, log_4(n_2) + (3 - k')(q - 14) - 4.343\, log_{10}(p_1(3 - k', 28))\}$$

- q_1 is the sum of quality values of mismatches of the best hit
- q_2 is the corresponding sum for the second best hit

- n_2 is the number of hits having the same number of mismatches as the second best hit
- k' is the minimum number of mismatches in the 28-bp seed
- q is the average base quality in the 28-bp seed
- 4.343 is $10/log10$
- $p_1(k, 28)$ is the probability that a perfect hit and a k-mismatch hit coexists given a 28-bp sequence that can be estimated during alignment.

At the time of publishing, sequenced reads were quite short (i.e. 28-36 nt). Furthermore, some of the assumptions previously made are too simplistic. Unsurprisingly, this method has had some criticisms.

First, we can not assume right away that all the sequence reads come from the reference. In addition, we can not assume uniform distribution as references – like the human genome – exhibit strong biases towards repetitive regions. At the end of the day, this score is approximated employing those suboptimal alignments that the mapper consider relevant (i.e. expected to have a meaningful contribution to the score). Yet, depending on the search depth, there might be many subdominant matches which can strongly affect this computation. Hence, finding them all becomes paramount as to avoid wrong approximations of the MAPQ score.

Secondly, these MAPQ scores are computed using a Bayesian statistical model based mainly on the error probabilities from the raw sequence quality scores. Therefore, this formula fails to incorporate the fact that errors can come from variants between the donor and the reference. In longer reads, this becomes more concerning as the model systematically prefers error events in low base quality positions – irrespective of if they are sequencing errors or putative variants.

Many other MAPQ computation proposals have been made [Siragusa, 2015; Tennakoon, 2013; Zaharia et al., 2011] – in fact, a new one for every mapping tool published –, yet they all base their computation on general statistical assumptions which overlook the fact that for each reference, sequence biases can debunk these expectations and lead to inaccurate estimations of the MAPQ score.

ROC Curves

Receiver operating characteristic (ROC) curve is a graphical plot which depicts the performance of any statistical binary operation. In this context, it is quite informative to break down all alignments by MAPQ score so as to evaluate the accuracy of a mapper attending to the quality scores reported. Without lose of generality, in our analysis we pursue the goal of ROC curves by plotting ACC (definition 3.3.3) against the FDR (definition 3.3.2) on the different MAPQ scores output by the mapper. By plotting ROC curves produced by different mappers we can derive how well these tools compute MAPQ scores and make accuracy comparisons among them.

Additionally, ROC curves are often used to establish empirical thresholds towards limiting the number of erroneous mappings passed on to the downstream analysis. It is important to note that these thresholds might change between mapping tools – depending on how conservative the computation of MAPQ scores is. As a result, some well-known down-stream analysis tools (e.g. GATK [DePristo et al., 2011; McKenna et al., 2010]) use fix MAPQ thresholds so as to filter-out unwanted mappings. Nevertheless, these fix thresholds seem tailored to specific mapping tools and can not be directly generalised to other mappers.

IV Rabema. Read Alignment Benchmark

To the best of our knowledge, Rabema (Read Alignment BEnchMArk) is the first published attempt to formally define a framework for benchmarking mappers accuracy. For this, their contribution well deserves a presentation in the context of this thesis.

In their paper, the authors realise that evaluating only in terms of total mapped reads or unique reads found can be deceiving. For instance, a very sensitive mapper could get very high rates of reads mapped without reporting necessarily the "true-matches". Meanwhile a non-complete mapper could easily miss conflictive matches and report many unique mapped reads. In order to disambiguate conflictive cases, they propose a formal definition of a match towards merging similar alignments in equivalence classes. Then, similard to the paradigms presented in section 3.1.1, they define three sequence alignment paradigms to perform their comparisons. Following their description:

- Search for all feasible matches ("all"): That is, searching for the whole matches space up to a given error degree.
- Search for all best matches ("all-best"): That is, searching for all strata until – and including – a non empty stratum (first matching stratum)
- Search for any best matches ("any-best"): That is, searching for any match in the first matching stratum.

Classifying mapping tools among these paradigms, they compared the reported mappings by each one against a gold standard previously computed. The gold standard is a reference with all the possible matches up to a certain error rate (i.e. matches space $\mathbb{M}_e^{T,P}$). The method breaks down how many matches are found in each stratum and, compares it to the gold standard, how many are missed. The authors also provide a software package implementing the method so as to evaluate the results of different mappers.

This method represents a well defined approach to comparing mappings in an organised fashion classified by strata. Furthermore, it introduces the idea of comparing mapping results as sets of mappings. Applying set operations between the results of different mappers and comparing them against a gold standard can highlight potential flaws of these tools. What is more, stratum-wise comparisons enables a focus on the matches at a specific distance leading to more valuable insight into the performance of mappers.

However, for the purpose of this thesis, we differ in the use of some methods and definitions. Firstly, we reject the idea of the "any-best" paradigm as it does not really represent a useful approach. Searching for any match in the first matching stratum at random does not account for any kind of multimap conflicts. Hence, this paradigm is prone to generate misleading results and should be avoided. Secondly, their matches equivalence class can be too relaxed in some cases and merge close-by matches from tandem repeat regions. Alternatively, we believe that multimaps to pathological repetitive regions simply reflect the ambiguous nature of the problem – in turn, limited by the resolution of current sequencing technologies). For that reason, it seems more informative to report these cases "as they are", instead of merging them. Finally, and depending on the application targeted, forcing complete searches of strata beyond interest is deemed to lead to unnecessary computations. As we explain in the following sections, deeper analysis of the information given by each stratum can lead to a better assessment of the requirements of the search. Intuitively, missing matches of very distant strata is irrelevant if the conflicts raised by less distant strata are deemed unsolvable. For that reason, comparing against the whole matches space $\mathbb{M}_e^{T,P}$ can sometimes be computationally expensive and unnecessary.

3.3.2 Conflict resolution

I Matches equivalence

In the context of sequence alignment, we do not usually desire to retrieve all possible matches as has been formulated above. There are many situations for which many matches at different distances over the same region create unnecessary conflicts. Furthermore, it is often the case that for a given match m_p at position p with distance e we can find many another valid matches in the nearby positions (e.g. just a nucleotide before or after by adding a single gap). These cases are obviously uninformative and irrelevant. Thus, they should be discarded. For this reason, for each position p of the reference text T we define its canonical match.

Definition 3.3.5 (Canonical match)**.** Given T, P, and $d()$, from all possible matches ending at that position p, the canonical match $\overline{m}_p = T_{i..p}$ is the one with least distance and larger span (i.e. $max\{|T_{i..p}|\}$).

Then, among the $|T|$ canonical matches to the text we define an equivalence class.

Definition 3.3.6 (Equivalent canonical matches)**.** Two canonical matches $\overline{m}_p = T_{i..p}$ and $\overline{m}_q = T_{j..p}$ are equivalent if they start at the same position (i.e. $i = j$).

Remark. From a given equivalence class of matches, we define its representative as the one with largest span.

In this way, from now on we will be only interested in finding those matches representative of their class (i.e. the canonical match with largest span). As a result, we will remove trivial conflicts and irrelevant repetitions. However, in the case of overlapping matches, one might argue that both should be considered the same. For example, this is the case for a sequence read that can match a tandem repeat each few bases. Many approaches – like Rabema – argue that computing all possible matches in these regions could take a lot of time and memory meanwhile its acknowledgement could create biases towards long repeated regions (which would weight more than shorter repeats). Nevertheless, these cases constitute real conflicts as there is no evidence that can help to correctly place the sequence read in any feasible match. In any case, the goal of the mapper is to report the inherent ambiguity of the read and pass on this information to the downstream analysis. A multimap-aware variant caller could process this information and weigh each plausible match over the total number of matches possible (or just discard it). Here it is important to note that the limiting factor is not the mapper accuracy but the sequencing technology. Had the machine produced a longer enough read, these multimaps could possibly be merged in the same equivalence class – resolving the conflict.

Despite its simplicity, the computation of equivalence between canonical matches in local regions of the reference at the time of mapping adds an extra computational difficulty. In practical terms, the most common approach is to pick the least distant match in case two canonical matches overlap. As a matter of fact, experiments show that at lengths of 100 nt less than 5 reads in a million are sensitive to this heuristic (i.e. failure to report the conflict).

II Stratified conflict resolution

As shown before, the concept of stratified matches not only helps to understand approximate string matching searches but also helps understand its results. For the paradigm of all-matches, where all matches up to certain error level are requested, the problem is reduced purely to searching completely for the matches space – without added complexity. It is well understood that completeness is paramount to achieving quality results.

However, in the endeavour of retrieving the "true-match" we are often presented with several feasible mapping locations to chose from. For a long time, the most commonly used criteria to solve this ambiguity was based on maximum identity under a certain distance metric (presumably accurate modelling biological events). In other words, out of a set of feasible mapping locations, the one with least distance is supposed to be the most likely sequenced. Progressively, this has lead to the common misconception that the least distant mapping is always the correct one. Ultimately, this has become an implicitly accepted criteria for some mapping tools that just look for the lest distant matches. However, this is a very simplistic assumption and in many cases can lead to erroneous results.

To better present the problem, Figure 3.6 illustrates very common cases of multimap resolution. In the first case – Figure 3.6 (a) – we can see that two matches are feasible; both at the same distant and with the same alignment (i.e. a $C \rightarrow T$ mismatch). Without more information there is no evidence that one of the two is the "true-match". Consequently we call it a tie and we report this ambiguity conveniently (i.e. MAPQ=0). Furthermore, this case is what we commonly know as a perfect tie; where all tied matches have the same alignment.

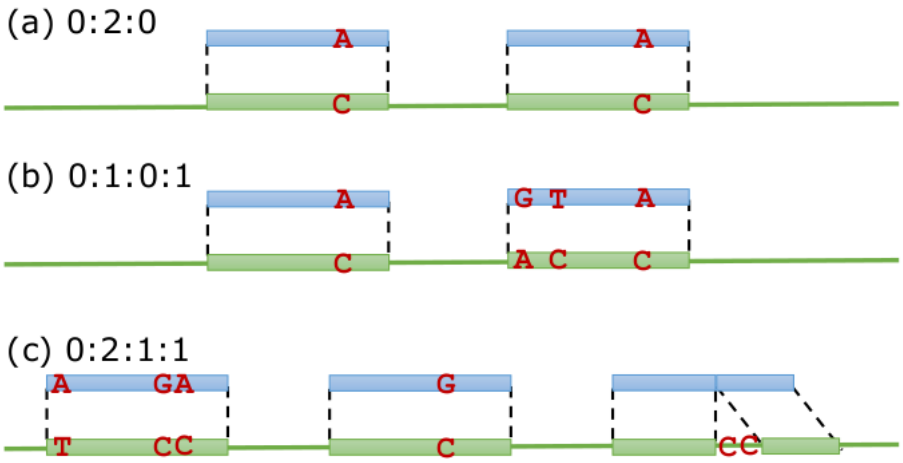


Figure 3.6: Stratified conflict resolution

The second case – Figure 3.6 (b) – pictures a different scenario where two alignments are possible with distant 1 and 3 respectively (i.e. 0:1:0:0:1). Despite this, there is still a certain probability that the most divergent is the "true-match", allthough rational suggests that the least distance has greater chances of being the true positive. This intuition get blurred when we add another match to the conflict with distance 2 – Figure 3.6 (c). In this case, we find three possible matches in contiguous strata (i.e. 0:1:1:1). Here, assessment of the match with one mismatch becomes less straightforward. In turn, this case cannot be classified as unambiguous and a low MAP should be reported.

Once again, it is critical to realise that these assessments can only be made under the condition that the strata is complete. Otherwise there could be potential hidden matches that change the scenario and therefore the outcome. Interestingly, these apply differently to each case. For example, in the case of a perfect tie – Figure 3.6 (a) –, once the two matches that determine the tie are found no more strata – or extra matches – are needed. Even if a mapper keeps searching deeper into more distant strata the final outcome would remain the same; a perfect tie. That is, once an unsolvable conflict if found, there is no need for further search exploration as more distant

mappings are neither going to solve the conflict nor add any meaningful information. Conversely, not only is it important to retrieve the matches themselves, but also to explore enough search space so as to assess the absence of conflicts. The goal is to report a measure of confidence so that no other match – which can cause a conflict – can be found until a certain stratum.

III δ-strata groups

All in all, our focus is to evaluate if a unique match found in the least distant stratum is certainly unique and free of conflict (i.e. a "true-match"). In this way, attending to the relation between the two least distant matches, we can define a classification of match spaces.

Definition 3.3.7 (δ-strata group)**.** The delta group δ_d of degree d is the set of all match spaces $\mathbb{M}_e^{T,P} = m_0, ..., m_n$ where the two least distant matches $m_i and m_j$ are at a distance d (i.e. $|i-j| = d$).

Remark. Note that with enough error tolerance, any sequenced read would eventually map any position in the reference genome. Without lose of generality, for searches up to d errors, some match spaces can only contain one match (i.e. unique reads). We classify these cases as δ_∞.

Intuitively, these δ-strata group classifications capture the different cases of stratified conflicts depicted before. In essence, each group represents a conflict category where there is match susceptible to being considered the "true-match" – by means of the maximum identity criteria –, and another at distance d. Depending on the relative distance between these two matches, the likelihood of the first being the "true-match" is expected to be higher. Note that δ-strata groups are not classified according to the error of the matches, but based on the relative distance between matching strata. Furthermore, they are agnostic to the distant metric chosen being of practical use for many distances widely used. Also note, that δ-strata groups are not defined towards estimating the probability of finding a match at a given distance from the least distant match. The ultimate goal is to assess the likelihood of the least distant match of being the source region of the sequenced read (which is a problem of biological nature).

This correlation between δ-strata groups and "true-match" prediction power is empirically depicted in the table 3.2. Here, three simulated datasets Illumina-like – all of them having 5x coverage at different lengths – have been mapped against the human genome reference (GRCh37). The mapping search was performed completely up to 5% error rate. Afterwards, all reads were classified according to its δ-strata group and the FDR was computed using the information from the simulator.

As the table 3.2 shows, there is a strong correlation between the δ-strata groups and the FDR. In fact, the higher the δ-strata group the smaller the FDR. This relation holds for all read lengths. As the read length increases, so does the number of uniquely mapped reads and in turn, the total amount of reads belonging to each δ-strata group decreases.

Note that the marginal increase in the FDR at the unique class is due to the limited scope in the search up to fixed 5% error. In other words, for some sequenced reads the search lacks enough depth and a relative small amount of reads are misclassified as unique. This exemplifies the very usefulness of δ-strata groups. Being able to accurately classify sequenced reads among this groups leads to an estimated FDR when searching for the "true-match". In turn, heuristic searches can be misleading by incorrectly classifying as unique reads that are not. Furthermore, knowing precisely the δ-strata group of a read enables downstream analysis to accurately discriminate reads. This allows for regulating the maximum FDR that downstream tools will introduce in their operation.

	75 nt		100 nt		150 nt	
	Total	FDR	Total	FDR	Total	FDR
δ_0 (perfect tie)	1.72%	44873	1.47%	34032	1.25%	25505
δ_0 (tie)	0.03%	88	0.02%	40	0.01%	14
δ_1	2.85%	915	1.84%	574	1.27%	307
δ_2	2.61%	66	1.60%	32	0.96%	12
δ_3	2.28%	34	1.56%	15	0.82%	5
δ_4	1.77%	13	1.49%	4	0.75%	1
δ_5	1.60%	3	1.33%	1	0.71%	0
$\delta_{>5}$	4.48%	2	6.50%	1	7.91%	0
δ_∞ (Unique)	82.65%	46	84.19%	9	86.33%	1

Table 3.2: δ-strata groups of 5x coverage Illumina-like simulated reads (FDR is given by each million reads).

δ-strata groups not only allow the estimation of mapping quality, but are also used to guide the search algorithm. Once again, it is not enough to search up to the least distant match (i.e. best-match), but to progress further in the search so as to guarantee accurate membership to the highest feasible δ-strata. Similarly, fixed error rate search algorithms are also not suitable to determine accurately the δ-strata group. The key idea towards uniqueness is the relative error between the least distant match and the next. For that reason, an accurate mapping algorithm is expected to (1) find the least distant match and, if the match is unique, (2) search further on until the next match is found or no other match is found up to a given delta (selected as to satisfy a target FDR). Interestingly, as the read increases in length, the FDR of each δ-strata group decreases. In a counterintuitive manner, this result suggests that forcing mapping tools to search up to a fixed delta rate d irrespective of the read length is not only unaffordable, but also potentially unnecessary. As experimental results show, once the least distant match is found, searching up to $d = 2$ (i.e. two errors more) at lengths of 150nt has the same expected FDR as searching up to $d = 4$ at lengths of 75 nt. This behaviour reflects the fact that longer reads are naturally more specific as they encode for more information w.r.t the same reference genome size.

Another relevant observation – quite useful in following sections – is that most of the δ_0 members are exact duplicates (i.e. perfect ties), meanwhile there is quite a small amount of repetitive reads that match with the same error at different locus with different error operations. This depicts the very bias nature of genome references and inspires observations to exploit these repetitions in the reference.

δ-strata groups towards benchmarking mappers

Additionally, δ-strata group are not only useful to assess accuracy and guide search alignment algorithms, but can also be employed to benchmark mapping tools. As opposed to Rabema, we are not overly concerned if a mapper misses a match at a distant stratum for a hypothetical repetitive read. As a matter of fact, benchmarking frameworks based on comparing against the set of all possible matches up to a given distance can be bias towards repetitive reads. For instance, the contribution of a very repetitive read to the bulk of matches is unbalanced with respect to that of a unique mapping read. More importantly, missing the right match (e.g. the one that can convert a δ_∞ read into a δ_1) is much more critical than missing the n-th match of a tandem repeat.

For that reason, comparing abundance of δ-strata groups between mapping results seems a more balanced and suitable framework to base comparisons (when searching for the "true-match"). Ideally, we would compare mapping results from different mappers against a ground truth given by a simulator. In this way, we would compute the abundance of sequenced reads in each δ_d for the results of a simulator and the mappings of every mapper in the comparison. Fluctuations in the abundance of sequenced reads for each δ_d – compared to the ground truth – reveals cases where the mapper at hand lacks accuracy and reports misclassified reads. Note that incorrect results are shown as deviations in the abundance towards higher δ_d. It is always easier to assess that a read is δ_∞ (i.e. unique) rather than searching for a more distant match than can prove membership to lower δ_i.

As opposed to comparing MAPQ scores, where a given mapper might output a low MAPQ without further indication on the reason of the low score (i.e. just reporting a match and MAPQ=0), there is no possible way to speculate comparing δ-strata groups. To assess the membership to an specific δ-strata group, a mapper should prove it by giving at least two matches (i.e. the least distant and the next one).

Even lacking a ground truth to compare against, abundance in δ_i can be compare across mapping results. In this way, even if the true abundance of a give δ_i is not know, differences in calculations between mappers reveal cases where the mapper reporting more abundance has mistakenly reported matches at incorrect δ-strata groups. That is, missing relevant matches towards potential mapping conflicts.

IV Local alignments

Before finishing this section, it is almost mandatory to make a short comment on evaluating accuracy at local alignments. Note that almost all metrics proposed involve end-to-end alignments. Yet, local alignments are often the fall-back strategy of many mappers when they cannot retrieve a "good" global alignment. Because of that, it is also very important to be able to evaluate the accuracy of local-alignments, compare them, and resolve potential conflicts that can arise.

The first thing to highlight regarding local alignments is that, in a way, we waive an explanation of the true origin of the whole sequence read. Looking for these types of alignments, we miss explaining why a regular end-to-end alignment can not be found and what the series of biological events – or sequencing artifacts – behind the read are. Even if a "good" local alignment is found, there is always room to question the nature of the missing part to align.

In fact, there are many benign reasons why a sequenced read has to be locally aligned to a reference genome, for example, remaining adaptors in the sequence, abnormal sequencing errors, cross contamination, etc. Leaving part of the sequence in these cases has no impact whatsoever in the analysis. Nevertheless, there are other cases – like big variations (i.e. large insertions or deletions, inversions, etc), chimeras, and other genomic rearrangements – that can potentially lead to a "good" local alignment missing the important nature of the genomic region sequenced. For that, the experiment specifics must be taken into account and local alignment should be biologically evaluated with these considerations at hand.

Principally there are two broadly accepted criteria used as to evaluate and compare local alignments. The most widely accepted is based purely on scoring the alignment using a widely known weighted scoring scheme during pair-wise alignment (between the sequenced read and the matching region). However, certain situations may be very sensitive to the actual scores used as to discriminate between local alignments. In these cases, a maximum span criteria is usually applied to favour the match with the largest span over the reference – which intuitively seems the one

explaining most of the sequenced read. In some of those cases, the scoring criteria is relegated to a minimum threshold that every local alignment has to fulfil in order to be considered. The purpose of these criteria is not only to enable local alignment comparisons, but also to regulate the amount of possible local alignments deemed relevant enough to be reported. Note too, that flexible constrains could virtually output any locus of the reference genome as locally matching the sequenced read.

In any case, resorting to local alignments should not mean that anything goes. Whenever local alignments are found – in most cases as the byproduct of a seed-and-extend algorithm –, it is crucial to assess its relevance not only in terms of alignment score but with respect to its uniqueness. Any given local alignment focus on a substring of the input sequenced read (i.e. local pattern). Here, it is mandatory to evaluate if the local pattern can potentially match any other region of the reference. In this way, any specific local alignment can be evaluated with the tools previously presented (e.g. δ-strata groups). Consequently, a local match can have a very high alignment score but fail to be unique, for example, being classified within δ_0 (i.e. at least there is another match for the local pattern with at the same error level). In this case, the local pattern is not pointing out an unequivocal region of the genome but leading to an ambiguity. For that reason, that local alignment is no better than another one for the same read with lower a alignment score but higher δ_i.

3.3.3 Genome mappability

In actual fact, the content of the genome itself is the real limiting factor in conflict resolution. The structure of the reference text determines which sequences can be aligned to a single location –the *uniquely mapping reads*– and which reads have more than one plausible locus of origin – the *multiply mapping reads*. In turn, this determines the final outcome of the mapper and the downstream analysis.

A recurrent concept in the context of sequencing mapping –and genome mappability– is the *proper length* of the reference. Roughly speaking, it is the minimum sequencing length at which one would expect to find unique matches were the genome random (in general the number of unique matches tends to increase together with the length of sequencing reads, because the longer the sequence the higher the number of mutations it tends to accumulate).

Definition 3.3.8 (Proper length)**.** Given an alphabet Σ, the *proper length* l_p of a string w is the minimum length for which any substring of w is expected to occur in w about once, i.e.

$$l_p = \log_4(|Genome|)$$

Remark. For example, the proper length for the human genome is 17 (since $4^16 < 6Gb < 4^17$), meanwhile that of D.Melanogaster is 15 and E.Coli is only 13.

One would expect that at sequencing lengths longer than the proper length, almost all sequenced reads turn out to be unique. However, this is often not the case. One must be aware that genomes are far from being random and their content is not . Genomes are the outcome of a long evolutionary history including frequent duplications at the level of the whole genome and at specific regions [Ohno, 1993]. As a result, the genome resembles a fractal-like structure with various different types of repetition [Cordaux and Batzer, 2009; Kazazian, 2004] (e.g. long and short interspersed repeats, segmental duplications [Bailey and Eichler, 2006], copy number variations [Freeman et al., 2006], paralogous gene families, pseudogenes, and modular domains appearing within the sequence of functionally diverse genes [Ohno, 1987]). On top of that, HTS technologies eventually introduce errors that must be accounted for and, in turn, will effect mappability computations lowering the number of unique k-mers.

Therefore, genome mappability strongly depends on the genome itself and the characteristics of the sequenced reads (i.e. average read length and expected mapping error). Once these parameters are known, it is possible to compute a priority mappability of the whole reference text (i.e the inverse of the number of times a read originating from any position in the reference genome maps to the reference itself).

Definition 3.3.9 (Genome mappability)**.** For a given read length l, maximum error e, and position p of a given genome G, let genome mappability $M_l^e(p)$ be defined as the inverse of the total number of matches of $P = G_{p..p+l-1}$ at distance e or less against the genome (i.e. cardinal of the matches space up to e).

$$M_l^e(p) = \frac{1}{\sum_{i=0}^{e} |M_i^{G,P}|}, \text{ for } P = G_{p..p+l-1}$$

Remark. Note that $M_l^e(p) \in (0, 1]$. Also, if the region P of the genome is unique, it holds that $M_l^e(p) = 1$. Otherwise, $M_l^e(p) < 1$ and the region is repetitive.

This metric allows the identification of truly "mappable" regions (i.e. those regions producing reads which map back unambiguously). In this way, highly mappable regions lead to unique mappings and, conversely, lower mappable regions tend to produce ambiguous mappings. Similarly, $M_l^e(p)$ can be understood as the probability that a read sequenced from position p of the genome can be correctly mapped back against its true-locus. This mappability information can be used not only to aid mapping tools to guide mapping algorithms and quickly discard low mappability regions, but also to fine-tune HTS experiments to increase the number of uniquely mappable reads.

In [Derrien et al., 2012] we compute mappability for several eukaryotic genomes (for different lengths allowing two substitutions). As a result, in table 3.3, a large fraction of eukaryotic genomes are not uniquely mappable. That is, even exact matches to such loci cannot be unambiguously assigned to their originating positions – which has obvious important implications for quantitative estimates.

	H.sapiens	M.musculus	D.melanogaster	C.elegans
Proportion of repeats (%)	45.25%	42.33%	26.50%	13.08%
Unique at M_{36}^0 (%)	80.12%	79.92%	71.07%	92.07%
Unique at M_{50}^0 (%)	84.56%	83.18%	72.14%	93.51%
Unique at M_{75}^0 (%)	87.84%	86.20%	73.54%	94.96%
Unique at M_{36}^2 (%)	69.99%	72.07%	68.09%	87.14%
Unique at M_{50}^2 (%)	76.59%	77.06%	69.44%	89.80%
Unique at M_{75}^2 (%)	83.09%	81.65%	71.00%	92.11%

Table 3.3: Mappability of eukaryotic genomes

Interestingly, according to table 3.3, approximately 30% of the human genome is not unique. Even at lengths of 75 nt, nearly 17% of the genome presents repetitions. As shown across different mappability profiles, mappability correlates with sequence length, genome size and number of errors. That is, the longer the reads and the smaller the number of mismatches, the higher the uniqueness of the sequence reads. Likewise, for the same read length, the smaller the genome, the higher the uniqueness of the sequence reads.

Furthermore, in [Derrien et al., 2012] it was shown that mappability profiles vary greatly with the transcript class (protein-coding genes, non coding RNAs, orthologous families, etc). Consequently, an improved measure of RNA quantification is proposed taking into account the mappability at the level of the single locus. In this way, using the mappability can lead to a better experiment design when the focus is on a particular element in the genome.

Additionally, mostly related to SNP calling, we can consider a build-up of reads over a given position and compute the associated mappability. Here, the mappability of the individual position can be calculated as the average contribution of all reads spanning that position.

$$P_l^e(G_{i..j}) = \frac{1}{j-i+1} \times \sum_{p=i..j} M_l^e(p)$$

Mappability scores are computed using the GEM-mapper [Marco-Sola et al., 2012]. Note that for these computations a complete mapper is required so as to perform full searches without missing valid matches. At the same time, computation of the mappability of a complete genome can be a challenging task for a genome mapper. In this paper, we propose several optimisations to avoid unnecessary computations related with highly repetitive regions of the genome. Finally, the mappability scores produced by GEM can be easily converted to other formats, like those suitable for display in the UCSC genome browser (for which GEM-mappability tracks are currently integrated).

Chapter 4

Approximate string matching I. Index Structures

Despite the advances in online searching during the last decades, sequential text searching is no longer a feasible approach for many applications. As the reference text becomes larger and the number of input patterns increase exponentially, many approximate string matching algorithms have begun to exploit the advantages of preprocessing the text into an index structure. In some cases, even the input patterns are indexed to exploit repetitiveness among them. In this way, large references are compiled into indexes that provide fast access to specific regions of the text at the cost of some additional space. As a result, indexed search algorithms do not need to explore the whole text to retrieve the matches of a pattern – searches can be restricted to isolated regions of the reference.

The first indexes were designed for structured text (typically natural language) where the concept of a "word" was clearly defined (*word-level indexes*). For instance, the popular and widely-used inverted list structure [Witten et al., 1999] records all the occurrences of every distinct word in the text, thus enabling fast access to any specific word. However, applications like molecular biology may require unbounded string searches, as there is no such a thing as a word in biological sequences. To solve this problem one has to move to *full-text indexes*, that allow one to search for arbitrary substrings in the reference text. In addition, some of the indexes can reproduce the

reference text –or any part of it– without accessing the original text. The main purpose of such so-called *self-indexes* is to reduce the memory footprint of searching algorithms. Full-text self-indexes have received a lot of attention in the past few years because of the latest developments in HTS technologies. Essentially, these data structures were exploited to address the challenge of indexing genome-scale references on computers with modest amounts of memory (for instance, one might wish to index the 3 Gbase of a human genome using less than 4GB of RAM). Full-text indexes can even be coupled with compression algorithms in order to generate *compressed full-text self-indexes*, thus reducing memory requirements even further. [Navarro and Mäkinen, 2007; Grossi et al., 2003; Ferragina et al., 2009; Sirén et al., 2008].

Although space consumption has always been a concern for index design, we must acknowledge that any general-use computer is currently equipped with large memories modules. During the last decades, memory production has adhered to Moores' law integrating exponentially more memory each year [Schaller, 1997; Mack, 2011]. For the time being, any regular laptop offers as much as 8GB of RAM, meanwhile high-end computing nodes can easily integrate 64GB and more. Yet, improvements in memory integration have not been reflected in the overall efficiency of the memory hierarchy. So, despite having access to large memories, latency penalties are still a mayor concern as they usually become the bottleneck of applications accessing large regions of memory. In fact, the memory wall [Wulf and McKee, 1995; McKee, 2004] represents the main reason for designing index structures with a reduced memory footprint. In general, smaller indexes translate into faster running times as they help to mitigate memory penalties. Having said that, there is subtlety worth mentioning: what really matters towards reducing the impact of the memory wall is not the size of the index itself, but the memory access pattern of the algorithm using it. That is, the size of the index may not be a concern provided that the accesses to it depict locality in time and space. For that reason, not only is it important that the overall size of the index remains small but also that the access exploits locality and depicts architecturally friendly access patterns.

Modern index designs are tightly coupled with architectural considerations in order to provide fast access times meanwhile retaining a low memory footprint. And due to their myriad benefits, these index structures have been widely incorporated into many mapping tools. Note that the process of building some index structures can take a considerable amount of time and resources. However this cost is amortised as usually they are built only once and used many times. Moreover, most commonly used genomic references are well established and unlikely to change in months or even years.

In this chapter, we will mainly focus of the full-text self-index FM-Index. Firstly, we introduce the concepts behind the classic index structures, the succinct indexes and the basics of the FM-Index. Then, we present the main practical considerations towards FM-Index design and implementation. Finally, we present some specialised index structures that enable more complex operations in the context of biological sequence mapping.

4.1 Classic index structures

Hashes based indexes

Possibly the most straightforward indexing idea is that of recording every k-mer in the original text. In this way, a hash-like table allows direct access to each k-mer and all the positions in the text where it occurs. Despite the simplicity of this idea, hashes have been the focus of researchers for many years and the central index structure of many important mapping tools [Homer et al., 2009; Hercus, 2012; Zaharia et al., 2011; Lee et al., 2014] – some of them still being widely used.

Depending on the design of the hash table, the length of the k-mer, and whether it stores all k-mer occurrences or not, the hash can take large amounts of memory. In its more simplistic approach, a hash can take up to $|\Sigma|^k$ entries to address all possible k-mers plus the pointers to each k-mer position recorded. Hence, the space consumption of these hash tables make their use quite limiting.

Suffix tree

The suffix tree is one of the most fundamental theoretical data structures in information retrieval and computer science in general. The suffix tree [Weiner, 1973; McCreight, 1976; Gusfield, 1997] of a string T of length n is a trie [Mehta and Sahni, 2004] built from all the suffixes of T where unitary paths are compressed. It occupies $\mathcal{O}(n)$ space.

Definition 4.1.1 (Suffix Tree)**.** A suffix tree for an n-character text T is a rooted, directed tree with the following properties.

- The tree has exactly n leaves numbered 1 to n.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of T.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of T that starts at position i (i.e. $T[i..n]$).

In this way, the suffix tree contains n leaves – each corresponding to a suffix of the text. Moreover, each internal node corresponds to a unique substring of T that occurs more than once in the text.

The main advantage of the suffix tree is that it can count the number of occurrences of a pattern P in the text in $\mathcal{O}(m)$ time (irrespective of the length of the text and the number of occurrences of the pattern). One searches the pattern P in the suffix tree by descending from the root of the tree according to the successive symbols of P. Once the node corresponding to the pattern has been located, all the occurrences can be listed in optimal time $\mathcal{O}(occ(P,T))$ by traversing the subtree of the node.

Despite having optimal theoretical construction and query time, practical implementations of the suffix tree typically require from 12 to 20 times the text size [Kurtz, 1999]. Thus, the suffix tree is mostly limited to theoretical studies and rarely used in real applications.

Suffix array

The suffix array (SA) [Manber and Myers, 1993] is a practical index structure proposed with the intention of reducing the memory footprint of the suffix tree. In a way, the SA is a compact representation of the suffix tree. However it still requires $\Theta(n \log(n))$ bits, although in most implementations, assuming strings have a maximum length of 4 GB, the size is 4 times that of the reference text.

Definition 4.1.2 (Suffix Array)**.** The suffix array $SA_T[0..n-1]$ of a text T is an array containing all the starting positions of the suffixes of T sorted in lexicographically order. That is:

$$T_{SA_T[0],..,n-1} < T_{SA_T[1],..,n-1} < ... < T_{SA_T[n-1],..,n-1}$$

Theoretically, the SA can be obtained by traversing in pre-order the leaves of the suffix tree. In practice, the SA can be constructed independent of the suffix tree in O(n) time using one of the multiple algorithms proposed in the literature [Kärkkäinen and Sanders, 2003; Puglisi et al., 2007]. As an example table 4.1 shows the SA of the text T=`GATTACA`.

G	A	T	T	A	C	A	$
0	1	2	3	4	5	6	7

Suffix Array	Sorted Suffixes
SA[0]=7	$
SA[1]=6	A$
SA[2]=4	ACA$
SA[3]=1	ATTACA$
SA[4]=5	CA$
SA[5]=0	GATTACA$
SA[6]=3	TACA$
SA[7]=2	TTACA$

Table 4.1: Suffix array of the text T=GATTACA$ ($ denotes the end of text, and is by definition lexicographically smaller than any other symbol belonging to the alphabet)

Note that any substring P of T is at the same time a prefix of a text suffix. In this way, any pattern P can be binary searched over the suffixes of SA in $\mathcal{O}(m \log(n))$ time as each step in the binary search requires comparing up to m symbols between the pattern P and the corresponding text suffix. The search time can be reduced to $\mathcal{O}(m + \log(n))$ by using an auxiliary structure described in [Manber and Myers, 1993] or even to $\mathcal{O}(m + \log(|\Sigma|))$ by using the so-called *suffix trays* [Cole et al., 2006].

Once the interval of the SA[lo, hi) corresponding to the pattern P is known, the number of occurrences of P in T is occ = hi − lo and the occurrences are located at $T_{\mathrm{SA}_T[\mathrm{lo}]}, .., T_{\mathrm{SA}_T[\mathrm{hi}-1]}$.

4.2 Succinct full-text index structures

Research in succinct data structures is motivated by the need for data structures which consume less space when handling massive amounts of data. In fact, succinct data structures aim to reduce the space requirements to as close to the theoretically lowest bound possible, while achieving fast string searching operations and a complexity comparable to that of a representation having no space constraints. In other words, succinct indexes aim to perform fast searches over massive amounts of data stored in little space.

4.2.1 The Burrows Wheeler Transform

According to reliable reconstructions, in 1983 David Wheeler discovered a cyclic text transformation which was left unpublished for many years. It was only in 1994 that he presented (with Michael Burrows) the paper entitled *A Block-sorting Lossless Data Compression Algorithm.* This paper introduces the BWT (Burrows-Wheeler Transform) which is exploited towards improving the efficiency of text compression algorithms. It is remarkable how an idea that would critically impact the future of compression algorithms and data structures was at first seen as impractical, and left in a drawer for so many years.

The BWT (also called block-sorting compression) [Burrows and Wheeler, 1994] is a fundamental concept in the field of succinct and compressed full-text indexes. The BWT is a permutation of the original text which has useful properties when one has to implement not only compression, but also string searching. Contrary to popular belief, the BWT does not cluster together "similar" characters. In fact, what the BWT really does is arrange together characters followed by the same context. In most cases, this leads to runs of the same characters which makes the BWT permutation susceptible to being easily compressed. For instance, using a locally adaptive zeroth-order compressor, the BWT of a text can be compressed down to the k-th order entropy of the text.

In order to obtain the BWT of a text (table 4.2 shows an example using the text T=`GATTACA$`), we must perform a series of steps as follows:

1. Let M_T be a conceptual matrix whose rows are the cyclic shifts of T`$` (the terminator character `$` is added at the end).
2. Sort the rows of the matrix M lexicographically.
3. Let T_{bwt} be the BWT of the text defined as the last column of M.

Relation with the SA

It is easy to note that there is a close relationship between matrix M_T and SA_T. When sorting the rows of the matrix M_T lexicographically, we are essentially sorting the suffixes of T. Note that $A[i]$ points to the suffix of T which prefixes the i-th row of M. Another way to construct T_{bwt} is to concatenate the characters that precede each suffix of T in the order listed by SA_T.

$$T_{bwt}[i] = \begin{cases} T[n-1] & \text{if } SA_T[i] = 0 \\ T[SA_T[i]-1] & \text{otherwise} \end{cases}$$

Also note that, given the way matrix M is built, all columns of M are permutations of the text T. So the first and last column of M are indeed one a permutation of the other.

G	A	T	T	A	C	A	$
0	1	2	3	4	5	6	7

T_{bwt}	SA_T	M_T
A	7	$GATTACA
C	6	A$GATTAC
T	4	ACA$GATT
G	1	ATTACA$G
A	5	CA$GATTA
$	0	GATTACA$
T	3	TACA$GAT
A	2	TTACA$GA

Table 4.2: BWT of the text T="GATTACA$" ($ denotes the end of text)

Properties of the BWT

Beyond the compression features, the BWT transform leads to a transformed text that depicts useful properties for string searching. To explain those properties we need to define the rank function $rank(c, i)$ and the accumulated counters $C[a]$.

Definition 4.2.1 (Rank function)**.** Let the function $rank(c, i)$ be defined as the occurrence function over the suffix of T_{bwt} from 0 to i.

$$rank(c, i) = occ(c, T_{bwt}[0..i])$$

Definition 4.2.2 (Accumulated counters)**.** Let $C[a](a \in \Sigma)$ be an array containing the number of characters in T that are smaller than a.

$$C[a] = \sum_{c<a} occ(c, T) \ , \forall a \in \Sigma$$

Interestingly, the occurrences of equal characters preserve their relative order in the last and the first columns of M. Hence, the i-th occurrence of a character c within T_{bwt} corresponds to the i-th occurrence of c in the first column.

Lemma 4.2.1. Let T be a text and SA_T its suffix array. Also, let c be the i-th letter of the BWT of T (i.e. $c = T_{bwt}[i]$ corresponding to the suffix $SA_T[i]$). Then, among all the suffixes starting with c, there are $rank(c, i-1)$ suffixes lexicographically smaller than $c \cdot T_{SA_T[i]..n-1}$.

Proof. Let $c = T_{bwt}[j]$ for some $j < i$. By construction of M_T, it holds that $T_{SA_T[j]..n-1} < T_{SA_T[i]..n-1}$. Consequently, it holds that $c \cdot T_{SA_T[j]..n-1} < c \cdot T_{SA_T[i]..n-1}$. Therefore, we have exactly $rank(c, i-1)$ suffixes starting with c and lexicographically smaller than $c \cdot T_{SA_T[i]..n-1}$. □

Lemma 4.2.2. Let T be a text and SA_T its suffix array. Also, let c be the i-th $(i > 0)$ letter of the BWT of T (i.e. $c = T_{bwt}[i]$ corresponding to the suffix $SA_T[i]$). Then, there are $C[c]+rank(c, i-1)$ suffixes smaller than $c \cdot T_{SA_T[i]-1..n-1}$.

Proof. From lemma 4.2.1, we know that are $rank(c, i-1)$ suffixes starting with c and lexicographically smaller than $c \cdot T_{SA_T[i]..n-1}$. By definition, suffixes starting with $d < c$ $(d \in \Sigma)$ are smaller than any $c \cdot T_{SA_T[i]..n-1}$. Thus, the total amount of suffixes smaller than $c \cdot T_{SA_T[i]..n-1}$ is the total sum; $C[c] + rank(c, i-1)$. □

It is important to note that, by definition of T_{bwt}, for any given $SA_T[i]$ $(i > 0)$, we know that $T_{SA_T[i-1]..n-1} = T_{bwt}[i] \cdot T_{SA_T[i]..n-1}$ – as $T_{bwt}[i]$ is the previous character of the suffix $SA_T[i]$. Then, using the previous lemmas, given the suffix $SA_T[i]$, we can compute the amount suffixes smaller than $SA_T[i-1]$ (i.e. the previous suffix in the text T) by counting characters within T_{bwt}. This enables one to map symbols in the last column of M_T back to the first column. This operations is commonly known as LF-mapping.

Definition 4.2.3 (Last-to-First mapping)**.** Let the $LF(i)$ LF-function which given the i-th suffix computes the position of the $(i-1)$-th suffix $(i > 0)$. Given that $T[i-1] = T_{bwt}[C[T[i]] + rank(T[i], i)]$, then:

$$LF(i) = C[T[i]] + rank(T[i], i)$$

Unwinding the BWT

The LF-function allows one to traverse the original text T backwards and retrieve the original text T from the BWT text T_{BWT} by starting at the first row (i.e. \$T) and repeatedly applying LF for n steps (algorithm 2).

Algorithm 2: Reconstruct T from T_{BWT}

```
input  : T_BWT and C[·]
output: Original text T
1 begin
2 |  j ← 0
3 |  for i = n − 1 to 0 do
4 |  |  T[i] ← T_BWT[j]
5 |  |  j ← C[T[i]] + rank(T[i], i)
6 |  end
7 end
```

4.2.2 FM-Index

The FM-index [Ferragina and Manzini, 2000, 2001], which stands for Full-Text Index In Minute Space and not for the name of the inventors (Ferragina and Manzini) as one might think, exploits the BWT transform to build compressed text indexes. Its authors observed that the BWT implicitly encodes for the SA of the text. In the end, they figured out a way of exactly searching patterns using the BWT text and some auxiliary data structures. At the same time, they proposed compressing the BWT text so as to reduced the index space requirements to $\mathcal{O}(n) - bits$, which in practice can index the 3GB of the human genome in less than 1GB depending on the compression algorithms selected – about 10 times less than the SA of the same reference text. Despite their compression properties, most current FM-Index designs do not compress the BWT for the sake of providing fast query accesses as the overall memory usage without compression is quite reasonable in practice – about the same size of the text encoded in ASCII.

LF_c function

Similarly to a search in the SA_T, an exact search using the FM-Index uses two sentinels (i.e. pointers) to delimit an interval of the SA – conceptually encoded in T_{bwt} – containing all the suffixes that have a common prefix (i.e. the pattern searched at each step).

Like the LF function, the LF_c function "pretends" that a given character is present at the end of a given row at M_T. Then, given a sentinel pointing to the i-th row of M_T (i.e. $T_{SA_T[i]-1..n-1}$), the LF_c can determine how many suffixes are smaller than $c \cdot T_{SA_T[i]-1..n-1}$ using lemma 4.2.2.

Definition 4.2.4 (LF_c function)**.** Let the $LF_c(i, c)$ be the function that which given the i-th suffix computes the amount of suffixes smaller than $c \cdot T_{SA_T[i]-1..n-1}$

$$LF_c(i, c) = C[c] + rank(c, i)$$

The main idea behind the FM-Index is to use this LF_c function to backward search a pattern by iteratively updating two sentinels – search interval on the SA – and exploiting the properties of the T_{BWT} (Algorithm 3).

Algorithm 3: FM-Index backward search

```
   input  : P pattern, T_BWT, and C[·]
   output: SA_T interval for P
 1 begin
 2 |  lo ← 0
 3 |  hi ← n − 1
 4 |  for j = m − 1 to 0 do
 5 |  |  c ← P[j]
 6 |  |  lo ← C[c] + rank(c, lo − 1) + 1
 7 |  |  hi ← C[c] + rank(c, hi)
 8 |  end
 9 |  return (lo,hi)
10 end
```

In this manner, the backward search delimits the SA_T rows that exactly match a given pattern in $\mathcal{O}(m)$ time. Moreover, we can compute the number of matching positions by subtracting the sentinels at any point of the search (i.e. $occ(P, T) = hi - lo$), without computing the actual matching positions in the text. In case $lo > hi$, we know that there is no exact match for the given pattern.

Note that once the SA interval for a given pattern is known, all the positions inside its range are encoded in SA-space. That means that, for a given i-th row the M_T matrix we lack the corresponding position in the text (i.e. $SA_T[i]$). In order to decode positions from SA-Space to Text-Space we need to unwind the BWT until a known corresponding position is reached. In a trivial case, we would traverse the text backwards, applying the LF function s times, until the first row is reached (i.e. beginning of the text T). At that point we would know that the source i-th row corresponds to s suffix of the text. Of course, unwinding all the way back to the beginning of the text in order to decode each position is quite expensive. Instead, we could think of storing periodical samples of SA_T to reduce the number of LF calls needed to decode a given position. For that purpose, efficient implementations of the FM-index store a separated sampled SA (Section 4.3.5).

FM-Index elements

Basically, the FM-Index consists of the BWT of the text and some auxiliary data structures used to accelerate the computation of the LF_c function, decode positions from SA-space to Text-space and fetch substrings of the original text.

As mentioned above, so as to accelerate the computation of the LF_c function, any FM-Index implementation includes the $C[a \in \Sigma]$ array which stores the **accumulated occurrences** of each

character in the text. In addition, the FM-index stores partial counters interleaved with the BWT text to accelerate the computation of the rank function. In this way, the BWT text is divided into equal blocks of b characters (b is the block length). Each **FM-Index block** b_i contains b characters of the BWT (T^i_{bwt}) and the partial counters ($c_i[a \in \Sigma]$) holding the count for each character until that position (depicted in figure 4.1). Note that the number of blocks in the FM-Index is equal to $|b_0, ..., b_n| = \lceil |T|/b \rceil$ and that, for a given character a, the content of the partial counters is equal to $c_i[a] = rank(a, i \cdot b) = occ(a, T_{bwt}[0..i \cdot b])$.

In this manner, computing $rank()$ avoids counting occurrences up to this point and leaves the counting to potentially few characters past the counters. Depending on the FM-Index layout, regular inclusion of counters in the index will speed up rank queries at the cost of more index space. This leads to a trade-off between space consumption and cost per LF-operation described in section 4.3.

Counters c_i				T^i_{bwt}
c_i[A]	c_i[C]	c_i[G]	c_i[T]	ACCGTACGGGATCGACACTGGTCGTACG

Figure 4.1: FM-index block design

Additionally, so as to accelerate the decoding of positions from SA-space to Text-space, any efficient FM-Index implementation stores **samples of the SA**. Depending on which sampling scheme is chosen, the average number of LF operations performed to reach a sampled position will vary. Also, different schemes allow for better average performances at the expense of extra index space. Section 4.3.5) offers a comprehensive explanation on the topic.

Finally, for many indexed approximate string matching algorithms it is very important to be able to retrieve any substring of the original reference text efficiently. Note that most alignment algorithms end up pairwise aligning the pattern against several candidate text regions. For that, efficient recovery of text regions is of key relevance. However, as it is initially formulated, the FM-Index can only retrieve text regions by means of iterating LF operations in $\mathcal{O}(|P|)$ time. Not only does this text retrieval mechanism perform several sparse index accesses before retrieving the whole text region, but each LF operation also involves several instructions just to retrieve a single character.

Also, in many cases from a given position p in SA-space, we need to to jump forward to start the LF text retrieval from several bases ahead. To do so, we need to decode that position into Text-space, increment the position p, and encode back to SA-space. As we can see, full-text built-in operations given by the FM-Index are very space efficient but quite convoluted for simple text queries.

For these reasons, it is more convenient to store the **full reference text** together with the FM-Index. Despite the index space increasing notably, DNA texts are rather suited to being compacted – using 3 bits per character – reducing the space overhead. For instance, for the human genome ($\tilde{3}$Gnt), the plain text could use up to 1GB of extra space (which is affordable using current computers). In this way, once we decode a position p into Text-space, accessing its corresponding text region becomes trivial and more efficient (i.e. all the region is stored contiguously in memory).

4.3 Practical FM-Index design

4.3.1 FM-Index design considerations

One of the most important aspects of an efficient FM-index is the actual index lay-out in memory. Though many papers have analysed the algorithmic implications of different FM-index designs from a theoretical standpoint [Hon et al., 2004; Ferragina and Manzini, 2001], very few emphasise enough the relevance of a careful choice of parameters for a practical and efficient implementation. Needless to say, as accessing the index is the key building block of any indexed search algorithm, the cost per access is going to be the major determinant of searching performance.

An efficient FM-index design must provide a convenient trade-off between fast-memory access and computation. In more detail, to perform an LF computation we need to access the index to fetch the text, partial counters, and $C[\cdot]$ and compute the occurrences for a given character. In this way, an efficient design should enable fast memory access and a lightweight rank computation.

From the memory standpoint, it is desirable to reduce the number of accesses so as to reduce the total memory bandwidth required per LF operation. But also, to efficiently access regions of memory in the order of gigabytes, it is very important to be aware of the penalties imposed by the memory wall [Wulf and McKee, 1995]. In the end, we want to achieve the maximum memory locality possible and the least possible number of failures in all the levels of the memory hierarchy (being TLB misses the least desirable). For that reason, we want to concentrate all the index-data associated with an LF operation in the same memory region and keep the overall index size as small as possible.

From a computation standpoint, counting occurrences of a character in a given FM-Index block of BWT text is the main bottleneck. Several factors like the encoding of the characters in the text bitmaps or the number of text bitmaps between partial counters must be carefully taken into account. Furthermore, depending on the encoding, some layouts allow using SIMD instructions to enhance the rank computation.

FM-Index memory access pattern

Towards efficient FM-Index designs, it is of critical importance to understand the memory access pattern of backward searches using the FM-Index. As opposed to SA binary searches – where the search progressively shrinks the range of the SA searched, focusing each iteration on a more specific region of the SA –, experimental results show that the memory accesses of backward searches are far from depicting locality. However, these accesses are not random at all. At each step, the backward search delimits those suffixes from SA_T prefixed by the pattern searched so far $P_{i..m-1}$. Each time a character $P[i-1]$ is added to the search, the two sentinels delimiting the search suffixes jump to another region of the SA_T, this time, prefixed by the new character and the previous prefix $P_{i-1..m-1}$. It is easy to see, that the accesses to the FM-index change from one region of the index to another depending on the next character added to the backward search. As presumably most of the characters of the pattern are different, each step of the backward search potentially jumps to a different, remote position of the FM-Index. Hence, the memory access pattern of the FM-Index is clearly not local at all. Furthermore, for genome-scale FM-Indexes, most accesses are bound to hit different cache blocks and – more troublesome – a different operating system page. For this reason, the backward search is probably one of least memory friendly algorithms and in practice is bound to generate a lot of cache misses and page faults.

Although there is little we can do to change the memory pattern of the backward search algorithm, it is quite advisable to chose an FM-Index design such that each block b_i fits entirely into a cache line. For instance, for Intel's core i7 the line sizes in L1, L2 and L3 are the same; that is 64 Bytes. Fitting and aligning whole FM-Index blocks into a cache line guarantees not only

that just one cache block has to be fetched to compute the LF operation (potentially incurring in just one cache miss), but also minimises the memory bandwidth needed per LF call. Here, the key insight is that, depending on the FM-Index design, it is preferable to add padding and waste index space for the sake of aligned accesses to cache blocks and page boundaries, rather than compacting the FM-Index blocks at the expense of failing twice when accessing blocks in the middle of two cache blocks – or even two system pages.

Architectural friendly FM-Index design

At the same time, an often overlooked feature of efficient FM-Index implementations involves the selection of architecturally friendly dimensions. For instance, note that any block size b for the FM-Index is theoretically possible. However, selecting b in such a way that T^i_{bwt} fits a computer word enables easier access and faster counting; for instance, using a 2-bits alphabet and allocating $b = 32$ characters in a 64 bits word. Furthermore, the power of two block lengths leads to power of two divisions to locate the FM-Index block at each LF operation. Since the power of two divisions and modules can be implementing shifts and masks, selecting power of two block lengths can avoid frequent and expensive machine instructions as divisions. Likewise, block counters c_i can be squeezed into the domain of positions of the reference text length. However, dealing with counters not a power of two (e.g. 17 bits counters) does not seem efficient due to the logical shifts overhead and penalties due to misaligned accesses to memory. For that reason, it seems quite reasonable to base the FM-Index design on 8,16,32 and 64 bits counters and block lengths power of two.

Strings collections and alphabet considerations

Furthermore, in the context of bioinformatics, most applications are required to index several genomic sequences (string collection) such as chromosomes, contigs, etc. For this reason, the most common approach is to concatenate all sequences into a single reference text. To avoid erroneous alignments at the boundaries, individual sequences are separated from each other using a special character denominated separator ('|'). Afterwards, the concatenated text is indexed all together to produce the FM-Index of the string collection and used as any other FM-Index reference.

It is important to note that, as the text-space is composed of several sequences, we require decoding positions from the text-space into positions at the individual sequences. This process is usually achieved by means of a translation table – often named locator table – which contains the offsets of all sequences comprised in the text reference. Given a match position p at the reference text, by binary searching the sorted intervals of the sequences contained in the locator, we can compute the source sequence and its source position from the collection of strings.

As a result, the FM-Index of the string collection artificially introduces an extra character (i.e. separator). But also, during construction the BWT added another extra character (i.e. '$') to mark the end of the text. Furthermore, the DNA alphabet is not solely composed of the four standard bases (i.e. $ACGT$), but allows uncalled bases – or unknowns; 'N'. In the end, using the FM-Index in the context of collections of genomic strings suggests the addition of three characters to the alphabet. However, this is quite unfortunate as the FM-index layout is very sensitive to the addition of extra characters. Each extra character not only requires the addition of an extra partial counter per block of the FM-Index, but also increases the bits required to encode the BWT text of the block.

Note that for a four character alphabet – like strict DNA – we only need 2 bits to encode the text. This is the main reason why some mapping tools employ "tricks" so as to avoid enlarging the alphabet:

- Removing the BWT terminator.

  Is easy to see that the '$' character only appears once. Thus it can be removed by storing its position in the BWT explicitly, and adding it conditionally within the LF operation. Despite it introducing a branch within a critical building block, it removes the need of an extra character.

  We propose a better and more elegant solution based on the observation that sequence separators never appear together. Is important to understand the main purpose of the BWT '$' character so as to (a) prevent circular traversals – and comparisons – from the beginning of the text to the end, and (b) establish "$T" as the lowest suffix of the M_T. For that, we could use two separators instead (e.g. "||T") retaining the same properties. Note that if the '|' character is lower than any other character in the alphabet, "||T" would be the lowest suffix of the M_T.

- Removing sequence separators.

  Moreover, sequence separators '|' can be removed by substituting them with artificial sequences uninteresting to the application or unlikely to map against any input sequence. For instance, many mappers choose to separate sequences using a long stretch of poly A's. The idea is to make the stretch long enough to avoid matches between two sequences through the poly A stretch. Furthermore, the only read that can map to that region is composed only by As and lacks interest for the majority of genomic applications. Though this is an arguable technique, it is often used in many widely used mapping tools.

- Removing uncalled bases on the reference.

  An even more controversial technique to reduce the alphabet to four letters is to replace every uncalled base in the reference with another nucleotide (e.g. A). Surprisingly, there are many real mappers that take this approach and defend it based on the number of cases where these replacements lead to incorrect mappings being negligible.

In short, many techniques can be used to reduce the alphabet back to four characters in order to save index space. However, even if adverse situations are unlikely, some of these techniques can potentially lead to silent mistakes during mapping. Therefore, it seems preferable to pay the price of a larger alphabet for the sake of flexibility and the certainty that no mistake can appear due these aggressive alphabet reduction techniques.

Genome-scale and double-stranded FM-indexes

Yet another important design decision involves the maximum size of the counters used. That is, the total addressable positions in the text or simply the maximum text length supported by the FM-Index design. In this respect, many early mappers heavily constrained by the maximum memory available at the time, assumed that 32 bits would suffice – all in all, a single strand of the human genome is barely 3Gnt. Nevertheless, restricting the maximum size of the genome to 4Gnt is quite unrealistic – leaving out widely used genomes and collections of genomes in the scientific community. Hence, current mappers implementing the FM-Index use 64bits counters.

Moreover, being able to index as much as 64bits can address (i.e. $2^{64} - 1$ positions), we can think of indexing both strands of the genome in the same FM-Index. The main advantage being the ability to search in both strands of the genome at the same time using a single backward search – which reduces vastly the complexity of stranded searches. In this way, given the forward strand T_f, we concatenate the reverse complement strand T_{rc} to generate the double-stranded reference $T_{ds} = "T_f|T_{rc}"$.

$$T_f = \texttt{"N G A T T A C A"}$$
$$T_{rc} = \texttt{"T G T A A T C N"}$$
$$T_{ds} = "T_f|T_{rc}" = \texttt{"N G A T T A C A | T G T A A T C N"}$$

Figure 4.2

Note that the addition of the reverse complement strand T_{rc} does not need to be reflected in the full reference text stored together with the FM-Index – saving index space. For this, whenever a text region of the reverse complement strand is required (i.e. positions beyond the forward strand), the corresponding p_{rc} position can be projected into the forward strand ($p_f = |T| - p_{rc} + 1$), retrieving the text region using the forward text T_f, and reverse-complementing it on the fly.

4.3.2 FM-Index layout

FM-index text layout

In terms of computation, the most expensive part of each LF call is counting the number of occurrences of a given character in a text block T^i_{bwt}. This strongly depends on the bitmap encoding and arrangement of the characters inside the block. For instance, a simple encoding of the **characters packed** would arrange one after another (i.e. $T^i_{bwt} = c_0 \cdot c_1, ..., c_{b-1}$). This naive approach not only forces several convoluted masking operations before the actual occurrences can be properly counted, but can also potentially waste bits from a computer word if the alphabet size is not a power of two (e.g. an alphabet using 3 bits would waste 2 bits per 32 bits machine word; 76MB for the human genome).

To avoid this, we could use a **character bitmap representation** – one bitmap per character – each using a computer word. In this case, counting occurrences are simplified to just shifting to remove the characters beyond the rank position and counting ones (i.e. popcount). However, this representation grows in space linearly with the number of characters in the alphabet.

For that reason, we propose a more practical and efficient design based on storing the bits of each character separated in different layers according to their significance; **bit-significance layers encoding**. For example, in the case of 3 bits per character, this encoding would store 3 separated words containing all the character's bits grouped from least to most significant (figure 4.3).

Character	Encoding
A	000
C	001
G	010
T	011
N	100
|	101

	N	G	A	T	T	A	C	A	|	T	G	T	A	A	T	C	N
$layer_0$	0	0	0	1	1	0	1	0		1	1	0	1	0	0	1	1	0
$layer_1$	0	1	0	1	1	0	0	0		0	1	1	1	0	0	1	0	0
$layer_2$	1	0	0	0	0	0	0	0		1	0	0	0	0	0	0	0	1

Figure 4.3: Bit-significance layers encoding

In this way, counting characters becomes relatively simple by (1) negating some layers – depending on the LF_c character –, (2) combining the result using bitwise AND, (3) shifting to remove the characters beyond the rank position, and (4) counting ones left in the bitmap. Assuming $b = 64$ – therefore each layer is a 64-bit machine word – and a 3-bit alphabet, algorithm 4 shows this computation.

Bit-significance layers encoding not only simplifies the counting occurrences operation, but also reduces the space occupation to strictly that of the character encoding, making it suitable for avoiding the waste of bits at the moment of encoding text into a FM-Index block.

Note that despite algorithm 4 being populated with conditional expressions, these can easily be removed (see section 4.3.3) as branch instructions are not desirable at this kernel function bound to be called billions of times during mapping. In general, we would like FM-Index design that lead to branch-free and orthogonal implementations of the LF_c function.

Algorithm 4: LF_c using bit-significance layers encoding

input : FM FM-Index, i position, and c character
output: $LF_c(i, c)$

```
1  begin
2      p_b ← i/64
3      b_i ← FM.b[p_b]
4      layer_0 ← (c&1)?b_i.T[0]:  b_i.T[0]
5      layer_1 ← (c&2)?b_i.T[1]:  b_i.T[1]
6      layer_2 ← (c&2)?b_i.T[2]:  b_i.T[2]
7      bitmap ← layer_0&layer_1&layer_2
8      shifted ← bitmap >> (64 − (i mod 64))
9      return FM.C[c] + b_i.c[c] + popcount(shifted)
10 end
```

Basic FM-Index block layout

Throughout the following sections we present design considerations and improvements to an initial FM-index block design. This simple scheme (figure 4.4) serves as a first approach to illustrate many practical considerations which lead to more efficient designs. This initial FM-Index design is based on a simple 1-level bucketing design – one level of partial counters and text bitmaps –, and a 3-bits alphabet encoded using bit-significance layers.

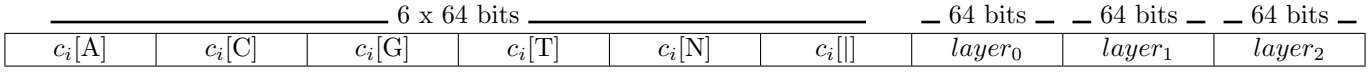


Figure 4.4: Basic 1-level bucketing FM-index design

Using this FM-index design (FM.1b.64c), each **block size** is 72 Bytes (i.e. $S(b_i) = 6x8B + 3x8B$). As each block encodes for 64 characters, the **overall index blocks size** is equal to $S(\text{FM.1b.64c}) = \lceil |T|/64 \rceil * 72$ Bytes. This implies that the **encoding efficiency** – average size per character – is equal to $E(\text{FM.1b.64c}) = S(\text{FM.1b.64c})/|T| = 1.125$ Bytes per character (i.e. more or less the same space as the ASCII encoding).

However, if we look deeper, we soon realise that the **partial counters size** is equal to $S(c_i) = 48 = 2/3 \cdot S(b_i)$. As much as 2/3 of the FM-Index blocks space is consumed by counters. This clearly shows that the space consumption of the partial counters is the main problem with most FM-Index designs. To palliate this effect, we will later explain three techniques to reduce the counters size: double-bucketing layout, alternated counters and increasing block length.

Besides overall memory consumption, big block sizes also impact the memory hierarchy. That is, assuming cache lines of 64B, note that a single block size is bigger than the cache line. Then, accesses to the FM-Index will have to fetch 2 cache blocks to compute a single LF_c call. Thus, each access to the FM-Index requires 72B of data (i.e. bandwidth) but effectively fetches 128B of data (effective bandwidth).

Double bucketing layout

Based on the previous FM-Index design, note that the difference between partial counters of contiguous blocks cannot be greater than the block length b. In this case, it seems natural to encode partial counters using a double bucketing scheme. Basically, double bucketing stores **mayor partial counters** at the beginning of a mayor FM-Index block. These mayor counters store the

counting occurrences of each character up to that position in the index. At the same time, these mayor blocks are composed of minor blocks with their own **minor partial counters** and text layers. However, minor counters only store the occurrences of each character from the beginning of the mayor block. In this way, the range of minor counters is reduced and smaller counter sizes can be used. This idea is known as double bucketing FM-Index, its theoretical formulation is included in several publications about FM-Indexes [Ferragina and Manzini, 2001; Siragusa, 2015; Chacón et al., 2015] and used in many practical implementations like GEM [Marco-Sola and Ribeca, 2015].

For this purpose, many combinations of mayor counter size and minor counter size are possible. Nevertheless, as mentioned earlier, power of two counter sizes are preferred to allow easier access and memory alignment. Besides, mayor counters need to address the maximum possible range of index positions. For this reason, the number of practical double-bucketing designs gets reduced to a few combinations (depicted in table 4.3).

Magnitude	Formula	$\|c_i\| = 8$	$\|c_i\| = 16$	$\|c_i\| = 32$
Minor block length	$\|b_i\|$	64 nt	64 nt	64 nt
Minor block size	$S(b_i)$	$8x\|c_i\| + 3x64 = 32$ B	$8x\|c_i\| + 3x64 = 40$ B	$6x\|c_i\| + 3x64 = 48$ B
Mayor block length	$\|B_i\| = 2^{\|c_i\|} - 1 + \|b_i\|$	319 nt	65599 nt	4294967359 nt
Max. minor blocks per mayor block	$\lfloor \|B_i\|/\|b_i\| \rfloor$	4 blocks	1024 blocks	65536 Kblocks
Max. Mayor block Size	$S(B_i) = S(b_i) \cdot \lfloor \|B_i\|/\|b_i\| + 64 \rfloor$	192 B	40 MB	3.5 GB
Efficiency	$E = \frac{S(B_i)}{\|b_i\| \cdot \lfloor \|B_i\|/\|b_i\| \rfloor}$	0.75	0.63	0.75

Table 4.3: Principal magnitudes for double-bucketing FM-Index designs varying minor counter size. Note that minor counters are always padded to align at 64bits (padding 2B per minor block using $|c_i| = 8$ and 4B using $|c_i| = 16$).

As shown in table 4.3, the most efficient design is found by balancing the counter size (i.e. minor counter size of 16 bits) using 0.63 Bytes per character. This double bucketing design (FM.2b.16c) is depicted in figure 4.5. One of the most striking design decisions on this model is the 2x16 bit padding on the counters so as to leave the layered text 64-bits aligned. However, this benefits the LF_c computation access time and leaves space to add useful information required for later sections (section 4.3.5).

Note that in this case, minor counters occupancy ratio gets reduced to $S(c_i) = 16 = 2/5 \cdot S(b_i)$ (i.e. the minor counters barely take as much memory as the BWT text of the block T^i_{bwt}). Also note, that mayor counters take a negligible amount of space. For instance, for the human genome, this FM-Index design will generate mayor counters with an overall size of 3MB. For that reason, mayor counters are separated and held together on a separate table susceptible to being cached to exploit locality of recurrent accesses. Furthermore, note that each minor block (40B) fits in a cache line. However, not all minor blocks are aligned with a 64B cache line. In this case, it is easy to see that on average 4/8 blocks will span over 2 cache blocks.

Alternated counters layout

Until this point, so as to compute the *rank*() operation, we have to count characters in a given block and add the result to the corresponding partial counter of the same block. However, another powerful observation suggests that the same computation can be fulfilled by using the partial counters of the next block and subtracting the occurrences beyond the rank position in the block. That is:

Mayor Counters (64 bits)
C_i[A]
C_i[C]
C_i[G]
C_i[T]
C_i[N]
C_i[|]

8 x 16 bits | 3 x 64 bits

c_i[A]	$\cdots$	c_i[|]	n/u	n/u	$layer_0$	$layer_1$	$layer_2$

Figure 4.5: Double bucketing FM-index design (FM.2b.16c). Figure shows a separated table for mayor counters ($C_i[\cdot]$) and minor counters interleaved with text bitmaps in each minor block.

$$
\begin{aligned}
rank(c,i) &= b_p.c[c] + occ(c, T^p_{bwt}[0..p_m]) \\
&= b_{p+1}.c[c] - occ(c, T^p_{bwt}[p_m+1..|b|-1]) \\
\text{Given that, } p_b &= i/64 \text{ and} \\
p_m &= i \bmod 64
\end{aligned}
$$

This method is called alternated counters and was first proposed by A. Chacón in [Chacón et al., 2015] towards storing a succinct FM-Index into modern GPUs. Using this index design and depending on the character query, the LF_c function will use the counters from its block or from the next one. Besides, note that LF_c operations for different characters are symmetrical, all cases require a single access to the FM-Index, and a single $occ()$ operation in required. And yet, the overall design has reduced the occupancy of the partial counters by half. Figure 4.6 depicts the double-bucketing FM-Index using alternated counters (FM.2b.16c + AC).

4 x 16 bits | 64 bits | 64 bits | 64 bits

	C_i[A]	C_i[C]	C_i[G]	C_i[T]	$b_i.layer_0$	$b_i.layer_1$	$b_i.layer_2$	...
...	C_{i+1}[N]	C_{i+1}[|]	n/u	n/u	$b_{i+1}.layer_0$	$b_{i+1}.layer_1$	$b_{i+1}.layer_2$	

Figure 4.6: Alternated counters FM-index design (FM.2b.16c + AC). Figure shows two contiguous minor blocks, each with alternated set of minor counters.

In practical terms, this method can be combined with any other design leading to an better overall encoding efficiency (table 4.4). As we can see, by only applying this method to the basic FM-Index design (FM.1b.64c) we reduce the average size per character by 3 bits and, combined with the double-bucketing scheme, we can reduce to an efficiency of 0.5 bytes per character (i.e. 4 bits per nucleotide).

	Efficiency	Efficiency using AC
FM.1b.64c	1.13	0.75
FM.2b.16c	0.63	0.50

Table 4.4: Encoding efficiency improvement by employing alternated counters

Generally speaking, this idea can be extended further towards alternating counters across many blocks. Ultimately, we could store a single counter next to each block, alternating the character

accounted for from block to block (figure 4.7 shows an schematic example of the idea). Depending on the counter size – and the level of bucketing used –, the resulting FM-Index designs can achieve efficiencies from 0.50 to 0.41. However, these designs require more than a single *occ*() computation to resolve a LF_c operation. In the extreme case of figure 4.7, each LF_c would require from 2 to 2.5 *occ*() computations on average.

C_i[A]	$b_i.layer_0$	$b_i.layer_1$	$b_i.layer_2$	...	C_{i+5}[[]]	$b_{i+5}.layer_0$	$b_{i+5}.layer_1$	$b_{i+5}.layer_2$

Figure 4.7: FM-Index design example extending the alternated counter

In practice, this design extension turns out to be quite convoluted and complex to implement. Provided that we are willing to perform more than a single *occ*() computation per LF_c operation, the next section presents a technique than can reduce counter occupancy even more at the cost of extra computations.

Increasing FM-Index block length

The next technique to reduce overall counter occupancy is based on increasing the length of each block. In this way, we avoid storing partial counters in favour of encoding more text per block. As a result, text content per block can no longer be fitted into a single computer word. Hence, this design technique often leads to more costly *occ*() computations – across larger text bitmaps – per LF_c operation. Nevertheless, this situation opens the possibility of introducing SIMD instructions to accelerate *occ*() computations over using larger machine words. In the same manner, conditional *occ*() can be implemented to further reduce *occ*() computations.

8 x 16 bits					128 bits	128 bits	128 bits
C_i[A]	$\cdots$	C_i[[]]	n/u	n/u	$layer_0$	$layer_1$	Text $layer_2$

Figure 4.8: Double bucketing using 128-bits bitmaps FM-index.

Figure 4.8 shows an example of this FM-Index design increasing the block length to $b = 128$ characters. In this example, *occ*() computations have to account for $2x64$ bit text bitmaps. It is easy to note that on average only half of the accesses to a given block will have to perform computations on both words of 64 bits (i.e. those having index i such that ($i \bmod 128 < 64$)). Therefore, a conditional implementation of the the LF_c operation would reduce to half the number of cases involving 128bit word operations. For those cases, SIMS instructions can be used to fit the whole block text into a machine word and compute the *occ*() at once.

In general, we could extend this idea to larger block lengths. Table 4.5 shows the encoding efficiency achieved by each configuration increasing the block length. Note the slow convergence of the efficiency towards the theoretical minimum at $E = 0.38$. Hence, beyond increasing b to 128 or 256 bits, there is little benefit in encoding efficiency at the cost of heavily increasing the computations per LF_c operation. Besides, the most notable observation is the equivalence between using alternated counters and doubling the block length. Then, using the FM-index design FM.2b.16c, is equivalent to increasing b to 128 bits to implement alternated counters; both achieving an overall encoding efficiency of $E = 0.50$. However, alternated counters only require computations over 64 bit words meanwhile using 128 bit blocks involves more intensive computations.

N-step FM-Index

	$\|b_i\| = 64$	$\|b_i\| = 128$	$\|b_i\| = 256$	$\|b_i\| = 512$	Theoretical Min 3bits per character
FM.1b.64c	1.13	0.75	0.56	0.47	0.38
FM.2b.16c	0.63	0.50	0.44	0.41	0.38
FM.1b.64c + AC	0.75	0.56	0.47	0.42	0.38
FM.2b.16c + AC	0.50	0.44	0.41	0.39	0.38

Table 4.5: Comparative of encoding efficiency increasing block length.

The last FM-Index layout design technique discussed is based on the observation that LF_c operations can be extended to cope with more than one character at a time. Described first in [Chacón et al., 2013], the N-step FM-Index enables advancing N characters per LF_c step at the cost of increasing the index size. To give more detail, a 2-step FM-Index design (figure 4.9) would not only have to store the last column of the M_T matrix, but also the previous one. With those 2 columns plus the partial counters for each pair of character combinations, a 2-step FM-Index could perform a backward search by querying two characters at a time [Chacón et al., 2013].

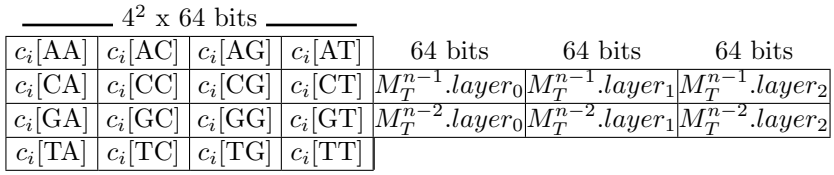


Figure 4.9: 2-Step FM-index design considering 4 letters alphabet.

This design not only enables faster searches, but also depicts more local index accesses that impact less on the memory hierarchy. That is, the computations preforming the equivalent to two LF_c operations are localised in the same region of the index. In practice, a 2-step FM-Index requires fewer instructions to compute the combined LF_c operations compared to a 1-step FM-Index design. In contrast, the main drawback of this technique is the exponential increase in index size as N increases. In fact, the example depicted in figure 4.9 leads to an overall encoding efficiency of $E = 2.75$; much higher than previous designs.

4.3.3 FM-Index operators

Just as important as the specific index layout, the actual implementation of the FM-Index operators plays a fundamental role in the overall performance of indexed approximated search algorithms. For this reason, careful considerations at the time of implementing these functions can have great performance impact. Additionally, basic search algorithms often present opportunities to implement tailored operators that can exploit particular conditions and perform better than single unspecific operators. For this reason, this section presents practical considerations for implementing FM-Index operations and efficient tailored operators for particular search scenarios to reduce computations and obtain better performance.

LF_c operator

Presumably, the LF_c operator is the most recurrent function within any approximated string matching algorithm based on top of the FM-Index. Within a real mapping tool, the LF_c operator can get to be called billions of times. For that reason, an efficient implementation of this function is crucial to achieve an good overall performance. Moreover, tiny implementation improvements can be reflected in the overall performance. The following optimisations can easily be implemented within any FM-Index lay-out design and quantitatively help to reduce the overall number of operations to compute the LF_c (figure 4.10 shows the implementation of these optimisations at the core of the LF_c function).

- Incorporated C[c] counters

  Recalling the basic LF_c definition ($LF_c(i, c) = C[c] + rank(c, i)$), to the computation of $rank(c, i)$ we need to add the content of the accumulated occurrence array (i.e. $C[c]$). Note that this addition is repeated each time we call the LF_c function. For that reason, we can add the content of $C[c]$ to each mayor counter of the FM-Index ($C_i[c]$) and avoid this addition whenever LF_c is called.

- XOR character table

  In order to avoid conditional code in the LF_c function, we can replace the conditional negation of the text bitmaps (algorithm 4) using small translation tables indexed by the character of the query. The resulting code is compact and avoids the use of conditional expressions which can lead to possible stalls in the pipeline.

- Bitmap mask table

  Note that the last computation of the LF_c function involves shifting unwanted bits to compute the popcount of the remaining ones. In order to reduce computations, we can replace the subtraction and the shift using a mask table with the precomputed masks. The resulting code reduces arithmetic operations in favour of an indexed access to a small array that can easily be cached.

- Power-of-two divisions/modulus

  Though not explicitly depicted in figure 4.10 and 4.11, divisions and modulus operations by powers of two can easily be replaced for shifts and masking instructions. Indeed, a modern compiler is expected to perform this substitution on-the-fly. As a result, some of the most computationally intensive instructions are avoided and much faster code can be generated.

- Specialised popcount hardware instruction

  Out of all the instructions executed within the LF_c function, the popcount operation can be the most time consuming. In fact, this operation is so computationally intensive and recurrent in computer applications that many hardware architectures implement tailored instructions

for it. In the case of Intel architectures, the SSE 4.2 instruction set extensions offer popcount hardware instruction that can greatly accelerate this computation. LF_c implementations using it can experiment a speedup of almost a 30% (compared to other bit-wise solutions available to perform the popcount [Warren, 2013]).

```
1 /*
2  * LF Implementation for FM.2b.16c FM-Index design
3  */
4 uint64_t bwt_LF(
5     bwt_t* const bwt,
6     const uint8_t char_enc,
7     const uint64_t position) {
8   // Locate BWT Block
9   const uint64_t block_pos = position / 64; // DIV
10  const uint64_t block_mod = position % 64; // MOD
11  const uint64_t* const mayor_counters =
12    bwt->mayor_counters[position/BWT_MAYOR_BLOCK_LENGTH]; // DIV + LD
13  const uint64_t* const minor_block =
14    bwt->minor_blocks[block_pos*BWT_MINOR_BLOCK_SIZE]; // MUL + LD
15  // Calculate the exclusive rank for the given DNA character
16  const uint64_t sum_counters = mayor_counters[char_enc] +
17    ((uint16_t*)minor_block)[char_enc]; // 2 LD + ADD
18  const uint64_t bitmap = (minor_block[2]^xor_table_3[char_enc]) &
19                          (minor_block[3]^xor_table_2[char_enc]) &
20                          (minor_block[4]^xor_table_1[char_enc]);
21                          // 6 LD + 3 XOR + 2 AND
22  // Return rank
23  return sum_counters +
24    POPCOUNT_64(bitmap & lf_mask_ones[block_mod]); // LD + AND + POP + ADD
25 }
```

Figure 4.10: Optimised LF_c operation implementation (FM.2b.16c)

This implementation leads to a LF_c implementation involving up to 24 operations. Note that these small optimisations are very sensitive to the specific compiler, architecture and other hardware used. Nevertheless, from several experimental benchmarks performed, we estimate that these optimisations can improve the running time spent on calls to the LF_c function from 30% to 50%compared to the naive implementation (algorithm 4).

Same FM-Index block exact search

Exact searching using the FM-Index (algorithm 3) involves the use of two sentinels that progressively reduce the SA_T interval corresponding to the pattern searched. In general, these sentinels will query different blocks of the FM-Index. Statistically speaking, once the search pattern is longer than the reference proper length (section 3.3.3), the resulting SA_T interval is bound to delimit a single occurrence. For that reason, searches close to the proper length – and beyond – are bound to query the same FM-Index block for both sentinels. In these cases, computations for both sentinels LF_c can be merged together to save computations. In fact, most computations are shared except for the final shift and popcount. Figure 4.14 shows the implementation of the combined LF_c operation for cases when both sentinels query the same FM-index block (i.e. $(lo/64) == (hi/64)$).

This combined LF_c function reduces the number of arithmetic/logical operations to 29 operations (i.e. only 5 operations more than to compute two LF_c calls). Experimentally, this translates to practical exact searches gaining a speed-up factor of 1.8x on average.

```
/*
 * XOR tables to mask bitmap depending on the character
 */
const int64_t xor_table_1[] =
  {-1ll, -1ll, -1ll, -1ll, -0ll, -0ll, -0ll, -0ll};
const int64_t xor_table_2[] =
  {-1ll, -1ll, -0ll, -0ll, -1ll, -1ll, -0ll, -0ll};
const int64_t xor_table_3[] =
  {-1ll, -0ll, -1ll, -0ll, -1ll, -0ll, -1ll, -0ll};
/*
 * Bitmap mask to replacing shift + subtraction
 */
const uint64_t lf_mask_ones[] = {
  0x0000000000000000ull,
  0x0000000000000001ull,
  0x0000000000000003ull,
  /* ... */
  0x7FFFFFFFFFFFFFFFull,
  0xFFFFFFFFFFFFFFFFull
};
```

Figure 4.11: Optimised LF_c operation tables

```
void bwt_LF_interval(
    bwt_t* const bwt,
    const uint8_t char_enc,
    uint64_t* const lo,
    uint64_t* const hi) {
  // Locate BWT Block
  const uint64_t block_pos = *hi / 64; // DIV
  const uint64_t* const mayor_counters =
    bwt->mayor_counters[*hi/BWT_MAYOR_BLOCK_LENGTH]; // DIV + LD
  const uint64_t* const minor_block =
    bwt->minor_blocks[block_pos*BWT_MINOR_BLOCK_SIZE]; // MUL + LD
  // Calculate the exclusive rank for the given DNA character
  const uint64_t sum_counters = mayor_counters[char_enc] +
    ((uint16_t*)minor_block)[char_enc]; // 2 LD + ADD
  const uint64_t bitmap = (minor_block[2]^xor_table_3[char_enc]) &
                          (minor_block[3]^xor_table_2[char_enc]) &
                          (minor_block[4]^xor_table_1[char_enc]);
                          // 6 LD + 3 XOR + 2 AND
  *hi = sum_counters +
        ( POPCOUNT_64(bitmap & lf_mask_ones[*hi % 64]) );
        // MOD + LD + AND + POP + ADD
  *lo = sum_counters +
        ( POPCOUNT_64(bitmap & lf_mask_ones[*lo % 64]) );
        // MOD + LD + AND + POP + ADD
}
```

Figure 4.12: Same-Block LF_c implementation (FM.2b.16c)

Precomputed LF_c operation

Another scenario of intensive use of the FM-Index is when searching for approximate strings by generating all possible words (i.e. neighbourhood search) in a suffix-tree like search. In this case, so as to explore all possible combinations, from a given search node the search has to explore the following nodes by adding all possible characters to the current search. This is translated

by doing LF_c queries at the same index position for all possible characters. Consequently, many computations are shared among all these calls. Figure 4.13 shows how to precompute all the LF_c elements needed to solve any LF_c query afterwards (figure 4.14).

```
1  void bwt_precompute_block_elms(
2      bwt_t* const bwt,
3      const uint8_t char_enc,
4      const uint64_t position,
5      bwt_block_elms_t* const bwt_block_elms) {
6    // Locate BWT Block
7    const uint64_t block_pos = position / 64; // DIV
8    const uint64_t* const mayor_counters =
9      bwt->mayor_counters[*hi/BWT_MAYOR_BLOCK_LENGTH]; // DIV + LD
10   const uint64_t* const minor_block =
11     bwt->minor_blocks[block_pos*BWT_MINOR_BLOCK_SIZE]; // MUL + LD
12   // Compute block elements
13   const uint64_t erase_mask = lf_mask_ones[block_mod]; // LD
14   const uint64_t bitmap_1 = block_mem[2] & erase_mask; // LD + AND
15   const uint64_t bitmap_N1 = (block_mem[2]^(-1ull)) & erase_mask; // LD + XOR + AND
16   const uint64_t bitmap_2 = block_mem[3];       // LD
17   const uint64_t bitmap_N2 = bitmap_2^(-1ull); // XOR
18   const uint64_t bitmap_3 = block_mem[4];       // LD
19   bwt_block_elms->mayor_counters = mayor_counters;  // ST
20   bwt_block_elms->minor_block = minor_block;        // ST
21   bwt_block_elms->bitmap_1__2[0] = bitmap_N2 & bitmap_N1; // ST + AND
22   bwt_block_elms->bitmap_1__2[1] = bitmap_N2 & bitmap_1;  // ST + AND
23   bwt_block_elms->bitmap_1__2[2] = bitmap_2  & bitmap_N1; // ST + AND
24   bwt_block_elms->bitmap_1__2[3] = bitmap_2  & bitmap_1;  // ST + AND
25   bwt_block_elms->bitmap_3[0] = (bitmap_3^(-1ull)); // ST + XOR
26   bwt_block_elms->bitmap_3[1] = bitmap_3; // ST
27 }
```

Figure 4.13: Precomputing LF_c-elements (FM.2b.16c)

Note that the number of computations is proportional to the logarithm of the size of the alphabet $\mathcal{O}(\log(|\Sigma|))$. In this particular example, the total number of operations to precompute an LF_c call is 27. Once these computations are done, resolving each LF_c call can be done using 9 operations. By only assuming that each LF_c call would be applied to 4 characters (i.e. A,C,G,T), the overall number of operations would be reduced from $24x4 = 96$ down to $27 + 9x4 = 63$ (i.e. reduction to 65% of operations).

```
1  uint64_t bwt_precomputed_LF(
2      const bwt_block_elms_t* const block_elms,
3      const uint8_t char_enc) {
4    const uint64_t sum_counters =
5      bwt_block_elms->mayor_counters[char_enc] +
6      bwt_block_elms->minor_block[char_enc];   // LD + ADD
7    const uint64_t bitmap =
8      block_elms->bitmap_1__2[char_enc & 3] &
9      block_elms->bitmap_3[char_enc>>2];       // SHIFT + 2 AND + 2 LD
10   return sum_counters + POPCOUNT_64(bitmap); // ADD + POP
11 }
```

Figure 4.14: Precomputed LF_c (FM.2b.16c)

4.3.4 FM-Index lookup table

On a higher level, a very powerful observation tells us that the first steps of every exact search have a high likelihood of hitting the same positions in the FM-Index. In other words, given a small q, the possible amount of q-grams is relatively small. Thus, one might think to 'memoizate' the exact search SA_T intervals for those q-grams, avoiding repetitive computations, and allowing fast direct access to pre-searched small q-grams. This is the key insight behind implementing a lookup table for the FM-Index. For limited length queries, the lookup table will resolve exact searches by direct access to a relatively small table in memory; avoiding LF_c intensive computations and potential memory slowdowns.

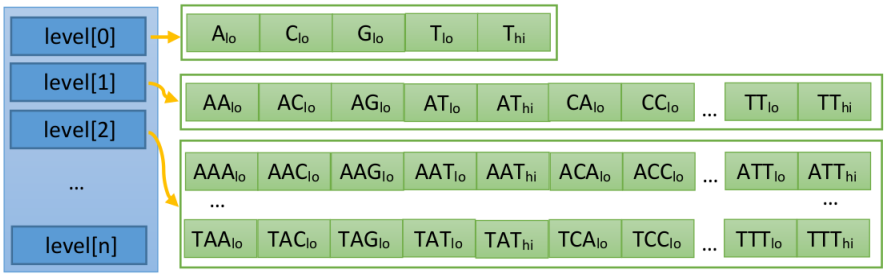


Figure 4.15: FM-Index lookup table (memoizated LF_c calls)

In more detail, the lookup table we propose is arranged in levels; one for each g-gram length or search depth. Each level points to an array of contiguous integers storing the limits of each q-gram search interval. Note that the upper and lower limits of contiguous q-grams intervals denote the same SA_T position (i.e. lo and hi pointers for q_i and q_{i+1} are equal). Hence, it is possible to merge them and only store non-repeated interval pointers (figure 4.18). Also note that due to the reduced size of the DNA alphabet, we can store every search interval for relatively large q-grams using little memory space. In practice, we are only interested in searching canonical nucleotides (i.e. A, C, G, T). Thus, we only store search intervals for q-grams drawn from a four letter alphabet (despite the face that the FM-Index could handle queries for any other character). As a result, each level l_i has a total of $S(l_i)$ pointers stored.

$$S(l_0) = 2$$
$$S(l_i) = 5 \cdot 4^{i-1}$$

Table 4.6 shows the overall size of memory for FM-Index lookup tables of different sizes. Interestingly, a lookup table allowing direct access to search intervals up to 11 characters (i.e. $l_{max} = 11$) only takes 53.33 MB of memory. As a result, this lookup table avoids querying the index for the first 11 nucleotides in every exact search. Combined with any filtering technique extracting seeds from the input pattern, this method avoids a remarkable number of queries to the FM-Index. In the case of q-grams, it is easy to see that this table serves as a replacement for any q-gram index up to l_{max}. But also, note that in the case of the human genome reference, the proper length l_p (definition 3.3.8) is roughly 15 nucleotides. That is, on average any exact search is expected to lead to an interval containing a single match once 15 characters are searched. Hence, any filtering algorithm based on finding exact unique seeds (i.e. seeds leading to a single candidate) is bound to save on average 73% accesses to the index.

Additionally, computations to access the lookup table are very straightforward and computationally lightweight. As shown in figure 4.18, intervals from the same search level are contiguous

Table levels i	Pointers per level $S(l_i) = 5 \cdot 4^{l-1}$	Total pointers $\sum_{j=0}^{i}(S(l_i))$	Lookup table size
0	2	2	0.02 KB
1	5	7	0.05 KB
2	20	27	0.21 KB
3	80	107	0.84 KB
4	320	427	3.34 KB
5	1280	1707	13.34 KB
6	5120	6827	53.34 KB
7	20480	27307	213.34 KB
8	81920	109227	853.34 KB
9	327680	436907	3.33 MB
10	1310720	1747627	13.33 MB
11	5242880	6990507	53.33 MB
12	20971520	27962027	213.33 MB
13	83886080	111848107	853.33 MB
14	335544320	447392427	3.33 GB
15	1342177280	1789569707	13.33 GB
16	5368709120	7158278827	53.33 GB

Table 4.6: FM-Index lookup table size

and can be accessed by just computing the offset $offset_i$ to the table based on the characters of the searched q-gram. Moreover, once computed the offset at a given level l_i, computing the offset to the search interval resulting from adding another character (i.e. offset at the next level l_{i+1}) can be done using the previous offset.

$$offset_0 = 1$$
$$offset_i = offset_{i-1} + P[n-i] \cdot S(l_i)$$

```
1 void rank_mtable_fetch(
2     const rank_mtable_t* const rank_mtable,
3     uint64_t const level,
4     uint64_t const offset,
5     uint64_t* const lo,
6     uint64_t* const hi) {
7   *hi = rank_mtable->sa_ranks_levels[level][offset];
8   *lo = rank_mtable->sa_ranks_levels[level][offset-1];
9 }
```

Figure 4.16: FM-Index lookup table query

Once the level of the search required is reached – or the limit of the table –, retrieving the actual interval pointers is as simple as indexing the corresponding level array (figure 4.16).

Computing thresholds on the lookup table

Until this point, the FM-index lookup table can be used within any approximate string matching algorithms for the first l_{max} steps of any search. However, for specific filtering algorithms the lookup table can be enhanced so as to avoid further computations.

Within dynamic filtering techniques, the adaptive pattern partitioning (APP) technique (section 5.4.3) partitions a given pattern into chunks so that each chunk occurs in the FM-index less than a given threshold number of times. This algorithm is greedy and proceeds backwards – from the end of the pattern – searching for each chunk until the search interval length drops below a given threshold (i.e. $hi - lo \leq th$).

In this scenario, a lookup table could be used to avoid the first l_{max} queries to the index for each chunk – or factor – computed by the algorithm. Experimental results show that the APP algorithms using a lookup table (for the first 11 steps) replace almost half of the accesses to the FM-Index with queries to the lookup table (i.e. 49% reduction of accesses to the FM-Index).

However, a more subtle approach would even avoid querying the lookup table until there is a chance that the searched interval is smaller than the threshold. In this way, using the lookup table beforehand, we can compute the **minimum factor length ($f_{min|threshold}$)**, which is the minimum length for which a q-gram in the index occurs less than the threshold. In the case of the human genome and for a threshold of 20 occurrences, $f_{min|20} = 10$ which implies that with a minimal modification to the APP, the algorithm could avoid any kind of memory access until the 10th character of each seed. In practice, this avoids most of the accesses to the lookup table; as they are known to be uninformative to the APP algorithm.

However, the minimum factor length is quite sensitive to the reference genome content. For instance, a bias reference genome could contain a single occurrence of a given four-mer ($q = 4$) and thus reduce the effectiveness of using the minimum factor length. Besides, the minimum factor length is a global magnitude which for some cases might not be a very accurate estimation.

For this reason, we propose a generalisation of this idea to exploit more accurate bounds within the lookup table for any reference text. To this end, we store the **actual minimum factor length** at the last level of each branch of the lookup table (level l_{max}). That is, at the end of each path in the lookup table we store the level on which the interval started to be below the threshold. In this way, the APP would progress until reaching the maximum depth of the lookup table, query the actual minimum factor length of that specific search branch, backtrack to that position – if needed –, and progress as the regular algorithm would do. Results vary depending on the actual reference genome. Nevertheless, the computation of the actual minimum factor length guarantees that the APP algorithm only queries the lookup table when strictly necessary.

4.3.5 FM-Index sampled-SA

As explained in previous sections, once a given SA_T search interval is computed, all the positions inside its range – encoded in SA-space – need to be decoded into text-space. In order to do so, we need to unwind the BWT from each position of the interval until a known SA-position is reached (i.e. a SA-position for which we know its corresponding text-position). The closer one of these "known" SA-position are to the initial SA-position, the less LF computations the algorithm will have to do to decode the position. For that, "known" SA-position are sampled from the original SA and stored in the so-called sampled-SA. In this way, we say that a position is sampled or marked if it is contained in the sampled-SA. It is easy to see that the more samples the sampled-SA contains, the more memory it will consume – ultimately as much as the entire SA. In turn, the more positions sampled, the fewer LF computations that will be needed to decode a given position. This leads to a trade-off between space-consumption and decoding performance. But also, in addition to the sampling rate selected, the sampling scheme chosen (i.e. which positions to sample or not), will determine the average number of LF computations per decode operation. Pseudo-code to decoding positions from SA-space to text-space using a sampled-SA is shown in algorithm 5.

Algorithm 5: FM-Index decode positions from SA-space to Text-space

input : p_{SA} position in SA-space, sampled-SA, T_{BWT}, and $C[\cdot]$
output: Corresponding p_{SA} position in text-space

```
1  begin
2      s ← 0
3      p ← p_SA
4      while p ∉ sampled-SA do
5          c ← T_BWT[p]
6          p ← C[c] + rank(c, p)
7          s ← s + 1
8      end
9      return (sampled-SA(p) + s)%|T|
10 end
```

Compacting SA samples

Note that the space consumption of the sampled-SA can have great impact on the overall FM-Index size. Irrespective of the sampling rate or scheme selected, an efficient sampled-SA design technique mandates to compact the compact the SA-samples to the minimum number of bits required by the reference text length. In the case of the human genome, only 33 bits are strictly needed to encode the almost 2x3GB of forward and reverse reference. Table 4.7 shows the space consumption of the human genome for various sampling rates.

SA-Space sampling scheme

The first and most simple sampling scheme is to sample positions regularly in SA-Space. That is, given a sampling rate s, this scheme would sample one each s position in the SA-space. Usually, any p_{SA} SA-position is sampled if $p_{SA}\%s = 0$. In this way, given any position to decode, it is enough to compute the modulus to check if the position has been sampled, otherwise the decode algorithm has to jump to the next position and try again. Once a sampled position is found, it is enough to divide the position by the sampling rate to compute the offset in the sampled-SA array in which the corresponding text-position is stored. Additionally note, that so as to reduce

Sampling rate	Sampled-SA size
4	6336 MB
8	3168 MB
16	1584 MB
32	792 MB
64	396 MB
128	198 MB

Table 4.7: Sampled-SA size for the human genome an various sampling rates

the impact of the modulus and division operation, sampling rates should be selected to be a power of two.

The main benefit of this scheme is its simplicity: simple to implement and efficient to check whether a position is sampled or not. However, this scheme lacks any guarantees on the maximum amount of LF steps a given decoding operation can take. Recall that the positions are sampled in SA-space and that LF traverses the position of the text backwards in text-space. For this reason, the number of LF steps in the worst case can be several times the sampling rate. In spite of this, it is easy to see that on average the number of LF steps per decode operation is expected to be s (assuming i.i.d. of the initial positions to decode).

Text-Space sampling scheme

In contrast, a more sophisticated and efficient sampling scheme is based on sampling positions in text-space. In this way, we can derive guarantees on the maximum number of LF per decode operation. Given a sampling rate of s, this scheme bounds the maximum number of LF computations to s (as one every s positions are sampled in the text-space). In contrast to the previous scheme, the average expected number of LF computations is equal to $\frac{0+1+2+3+...+(s-1)}{s}$ $= \frac{((s-1)+1)\cdot s}{2} = s/2$. That is, half the number of LF computations for the same sampling rate (i.e. the same sampled-SA space occupation).

However, this method has a downside: knowing which positions have been sampled is not as easy as before. In fact, as sampling is decided according to corresponding text-positions, we need an additional data-structure to mark which SA-positions have been marked. For that purpose, we can accommodate another bitmap to the previously presented layouts and mark those sampled positions in the bitmap. Figure 4.17 shows the FM.2b.16c FM-Index design incorporating a sampled bitmap. Also note that this design uses one minor counter to accumulate the number of sampled positions until that point. Given a sampled SA-position, so as to compute the offset in the sampled-SA array in which the corresponding text-position is stored, we count the number of sampled positions before this position (using the sampled bitmap and the counters). As a result, this scheme increases the FM-Index efficiency from $E = 0.63$ to $E = 0.75$ (i.e. one extra bit per character encoded).

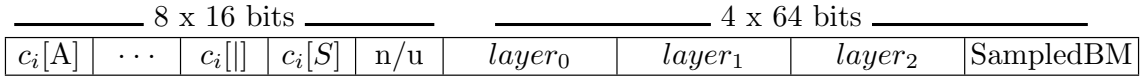


Figure 4.17: Double bucketing FM-index design (FM.2b.16c + sSA) with interleaved sampling bitmaps

Exact searching and decoding unique intervals

As mentioned above, once an exact search has reached the proper length of the genome, there is a high likelihood that the resulting search interval has length one (i.e. statically a unique match).

From that point on, the exact search will continue searching for the remaining characters; updating the interval of the only candidate left. In this case, instead of waiting for the search to finish and decode the resulting position, we could keep track of the intermediate search intervals queried (i.e. single FM-Index positions) and store whether one of these positions is sampled or not. By doing so, and irrespective of the result of the exact search, we could avoid additional LF_c operations to decode the result of the exact search.

This technique is of special interest for exact searches and filtering algorithms based on exact searching potentially long chunks of the input sequence. In the case of MEMs – or SMEMs –, as the algorithm determines maximal exact matches, we could keep track of the sampled position to reduce the overhead of LF_c operation at decoding MEMs for later use.

4.3.6 Double-stranded bidirectional FM-Indexes

As mentioned in section 4.3.1, double stranded indexes not only allow searches in both strands of the genome at the same time, but also depict symmetrical properties that can be exploited as to perform bidirectional searches. The ability of performing bidirectional searches leads to a new class of efficient approximate string matching algorithms. These bidirectional algorithms exploit this feature as to search for patterns starting from the better conserved parts of the sequence, and progressing forward and reverse until completion of the search. This method allows to better restrict the number of combinatorial possibilities in neighbourhood searches, leading to faster algorithms when the error rate is high or the input sequence is repetitive.

Forward search

Using the FM-Index of T_{ds}, we can perform backward exact searches of any pattern using algorithm 3. However, the FM-index doesn't allow to perform forward exact searches out-of-the-box. Nevertheless, with the aid of a complementary index of the reverse text we can simulate a forward exploiting simple properties of the SA.

Given the forward and reverse indexes of a text, let us assume that we know the search interval I_0 of the pattern P_0 (e.g. P_0="TAG") and the interval $\overline{I_0}$ of the reverse pattern $\overline{P_0}$ (i.e. "GAT"). Also, note that forward pattern's intervals – like I_0 – are computed using the regular index, and reverse pattern's intervals are computed using the reverse index. Then, we can update the interval I_0, with a character c, to forward search for P_1 (i.e. "GTAG"), by noticing that P_1 occurs in the forward index the same number of times $\overline{P_1}$ occurs in the reverse index. For simplicity, let us say that $|I_1| = |\overline{I_1}|$. In contrast, $\overline{I_1}$ can be easily obtained by means of a backward search (algorithm 3) of $\overline{P_1}$ ("GATG") using $\overline{P_0}$ (i.e. "GAT"). Now that we know the length interval $\overline{I_1}$, we know the length of the interval I_1. At the same time, as the suffixes in a SA-array are sorted, we know that $I_1 \subseteq I_0$. Hence, we only need to compute the offset of I_1 in I_0. That is, the length of the intervals corresponding to smaller suffixes: $'A' \cdot P_0 =$ and $'C' \cdot P_0 =$ (i.e. "ATAG" and "CTAG"). Using the previous idea, we compute the length of the reverse of those intervals (i.e. "GATA" and "GATC"), using the backward search on $\overline{P_0}$. This completes the computation of the an interval doing in a forward search. Notice that when we forward search we update the reverse interval and vice versa. In this way, a bidirectional search keeps both intervals – forward and reverse – updated as it progresses.

Double-stranded symmetry

Generally, this method would require of two indexes – forward and reverse. Nevertheless, exploiting the symmetrical properties of a double stranded index (lemma 4.3.1), we could simulate the forward search using the reverse-complement text. The double stranded index T_{ds} is built from the concatenation of the forward strand T_f and the reverse-complement strand T_{rc}). By

construction, it depicts the property that the number of occurrences in the index of any string s its the same as the number of occurrences of its reverse-complement s_{rc}.

Lemma 4.3.1. Given the reference text T, its double stranded index $T_{ds} = "T_f|T_{rc}$, and a string s, it holds that the number of occurrences of s in the index is the same as the number of occurrences of its reverse-complement s_{rc}.

$$occ(s, T) = occ(s_{rc}, T)$$

Corollary 4.3.1.1. The length of a given search interval I is equal to the corresponding reverse-complement interval I_{rc} (i.e. interval of the reverse-complement pattern).

In this way, we can replace reverse queries against a reverse index, with reverse-complemented queries against a double stranded index. As opposed to other approaches proposed [Lam et al., 2009], this method requires no additional index space beyond a regular double-stranded indexes.

4.4 Specialised indexes

For some specific algorithms and applications, specialised indexes offer tailored operations that can enhance approximate string searches and help model the particular characteristics of the experiment at hand. In this section we sketch some of the most notable index variations that play a fundamental role in certain algorithms and specialised application contexts. These variations are implemented within the GEM-mapper.

4.4.1 Bisulfite index

In the context of bisulfite sequencing, converted reads obtained fall into two groups of reads with different observed conversions. Using Illumina protocols, read 1 has the majority (i.e. $\sim 95\%$) of C's which are converted to T's and read 2 has the majority (i.e. $\sim 95\%$) of G's which are converted to A's. This bisulfite conversion generates two distinct sources of DNA sequences corresponding to the original strands of the original DNA molecule. Each of the 2 groups of sequences can be read in either direction, leading to 4 distinct types of read.

One of the most common alignment strategies involves fully converting initial sequences so that; any remaining C's in read 1 are converted to T's, and any remaining G's in read 2 are converted to A's. Afterwards, sequenced reads are then aligned against the two references generated by fully converting the standard reference genome. For this, the original reference text T is processed, generating the concatenated reference of converting $C \rightarrow T$ ($T^{C\rightarrow T}$) and converting $G \rightarrow A$ ($T^{G\rightarrow A}$). In the case of double stranded indexes:

$$\begin{aligned} BS_{ref} &= T_{ds}^{C\rightarrow T} | T_{ds}^{G\rightarrow A} \\ &= T_{f}^{C\rightarrow T} | T_{rc}^{C\rightarrow T} | T_{f}^{G\rightarrow A} | T_{rc}^{G\rightarrow A} \end{aligned}$$

This strategy is unbiased with respect to methylation status, but does result in a small loss of information from the remaining C's in read 1 or G's in read 2. Nevertheless note that the percentage of C's in read 1 is very small (i.e. ¡2%) so the information loss is not substantial.

4.4.2 Homopolymer compacted indexes

It is common in many experiments to preprocess the input reference genome to avoid highly repetitive regions. These regions tend to generate biases in many analyses and become the source of expensive computations. Yet the results generated in many cases from these regions turn out to be of little interest or have little overall impact. For that reason, it is common to remove these region in a process commonly known as masking the repetitive regions. In order to select these regions accurately, we can use the mappability score so as to determine highly repetitive regions and mask them out from the reference and posterior downstream analysis. Masking the input reference is a simple example of how a preprocessing step of the reference can lead to simpler and faster analysis procedures by reducing the complexity of the problem.

In the context of HTS technologies, long read sequencing technologies – like Pacbio or nanopore – stand out from traditional HTS technologies as they able to produce much longer sequences. As a result, this long-read data has unlocked many genome analyses and experiments unfeasible using short read sequencers – such as better genome assembly or analysis of low complexity regions. However, these technologies have a mayor shortcoming: they produce reads with a much higher error rate (i.e. 20%) most errors being produced at the time of sequencing homopolymers. In

fact, the longer the homopolymer, the greater the chances are that the sequencer will produce an incorrect read by adding or removing bases to the stretch of nucleotides. This has a great impact on mapping algorithms as not only do higher error rates force deeper approximate string searches, but also because of the nature of indels and their inherent complexity.

Inspired by the idea of masking the reference to reduce the complexity of the problem, we want to propose the homopolymer compacted index (HC-index). The main idea behind this technique is to compact homopolymers in both the input reference and the input sequences to align. As a result, any discrepancy – or error – in the length of a sequenced homopolymer is transparently avoided and the process of mapping is restricted to match potential homopolymers of any length. Figure 4.18 shows a scheme of mapping in both spaces; regular text-space and HC-space.

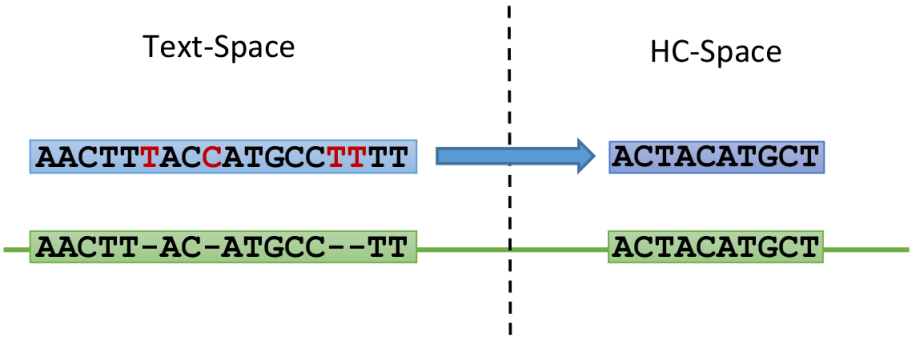


Figure 4.18: Example of mapping using a HC-index

The HC-index is designed to be coupled with an approximate search algorithm based on filtering. Then, the main purpose of the HC-index would be to enhance the process of "seeding" in any filtering algorithm. In this way, candidate generation of filtering algorithms would be transparently insensitive to homopolymer errors. Afterwards, a candidate verification step against the original text could report the complete alignment in text-space. In this manner, the HC-index can be seen as an implicit and transparent technique for gapped seed generation, tailored for homopolymers errors.

Note that, in order to translate positions from HC-space to text-space – and vice versa – we need to store an extra sampled array – in the spirit of the sampled-SA. Yet, this extra array, sampling as little as 1 position each 100, has little impact on the overall size of the final index (e.g. 150 MB for the human genome).

As an additional advantage, generating the neighbourhood of a string using the HC-index becomes computationally less expensive. Note that single insertions count for all possible insertions of the same character and mismatches get to represent not only mismatches, but also insertions.

Chapter 5

Approximate string matching II. Filtering Algorithms

Since 1990 filtering algorithms have been studied as a very effective approximate matching technique. They base their success on the observation that, for a reasonable choice of tolerated error degree, it is easier to look for regions of the text containing pieces of the pattern without errors rather than trying to align the pattern against every position of the text. Since looking for exact chunks of the pattern (or *occurrences* of the pattern in the text) can be much faster than trying to do approximate matching with the whole pattern, filtering algorithms can quickly discard large regions of the text and perform much faster than other approaches in actual practice.

For instance, while searching for the pattern $P =$`GATTACA` up to one error, we could inspect the text and quickly discard the regions that do not contain either one of the substrings $s_0 =$"GATT" or $s_1 =$"ACA". Indeed, no matter where in the pattern the possible error is located, either s_0 or s_1 must occur in all patterns containing up to one error.

In general, filtering techniques aim to filter out as many regions of the text as possible to reduce the search space being explored down to a few candidate regions. This approach can potentially avoid inspecting the whole text, often leading to algorithms that are sublinear in the length of T in most practical cases.

A significant amount of research has been conducted in this area and a number algorithms have been proposed for different applications and typical parameter ranges [Burkhardt and Kärkkäinen, 2003; Weese et al., 2009; Kehr et al., 2011; Fonseca et al., 2012]. Furthermore, many filtering algorithms have been proposed and adapted for both online and indexed searches. As of today, all practical sequence alignment tools rely on filtering one way or another.

In this chapter we first introduce the traditional filtering paradigm together with a classification of filters and how to evaluate their practical efficiency. Then we present the most relevant and representative filtering techniques used in sequence alignment algorithms. Subsequently we analyse how some filtering techniques scale with the read length, and how they can be enhanced. After that we present dynamic partitioning techniques and propose an adaptive filtering scheme able to obtain almost optimal pattern partitions – an embodiment of what we call *data-aware filtering*. Going on we introduce the idea of chaining specialised filters in order to achieve better performance.

We also discuss how different error functions can be used to produce an error model that can be computed much more quickly than biologically-inspired metrics, while obtaining equivalent results. Finally we state the effectiveness of progressive filtering by ranking several filtering methods, and deriving bounds on the number of candidates for a match when such methods are applied.

5.1 Filtering algorithms

Classic filtering paradigm

Generally speaking, all filtering algorithms are based on partitioning the pattern in order to reduce the approximate string matching problem needed to search for the individual parts. This partition of the pattern –known as *filtering scheme*– induces a set of conditions that any substring of the text has to satisfy in order to be a valid match (i.e. necessary connection but maybe not sufficient). Afterwards, using an index of the text – indexed search – or the text itself – online search –, all the candidate substrings are gathered and aligned against the pattern to discover the true matches.

Traditionally, filtering algorithms are presented as just a pre-selection step to filter out regions of the text (candidate generation). In this way, filtering algorithms are unable to discover matches by themselves. Instead, a verification algorithm – potentially any regular alignment algorithm – has to be coupled with them so as to compute the actual matches (i.e. align with distance below a given threshold). Figure 5.1 depicts the general filtering paradigm.

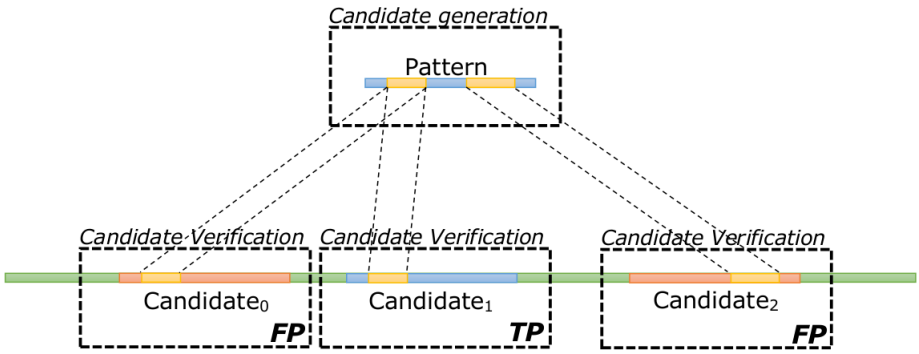


Figure 5.1: Filtering paradigm

Filtering algorithm performance strongly depends on the balance between these two stages. Very restrictive filters can produce very few candidates at the expense of intensive computations meanwhile fast and nonspecific filters can produce too many candidates. Quite often, the selection of the algorithm in the verification step is overlooked under the assumption that previous candidate generation will have lead to few candidates. Nevertheless, it is extremely important to understand that both steps have different complexities. For instance, in most practical scenarios, candidate verification grows quadratically with the length of the pattern, whereas candidate generation operates linearly.

Filtering efficiency

Back to filtering algorithms, specifically to the candidate generation step, we can encounter several cases depending on the nature of the candidates reported by the filter (see table 5.1).

True positive (TP); when the filter reports a candidate that aligns with the pattern.

False positive (FP) (type I error) or “”false alarm”; when the filter reports a candidate that doesn’t align with the pattern.

True negative (TN); when the filter discards a region of the text that truly doesn't align against the pattern.

False negative (FN) (type II error); when the filter discards a region that aligns against the pattern.

		Candidate generation	
		Matching	Discarded
True condition	Matching	TP	FN
	Not Matching	FP	TN

Table 5.1: Filtering candidates confusion table

We define the sensitivity and specificity of a filter accordingly to this. Sensitivity (SEN) – a.k.a true positive rate (TPR) – measures the proportion of candidates reported that are valid matches (w.r.t the number of matches missed). Specificity (SPC) – also called the true negative rate – measures the proportion of candidates that are correctly filter out (w.r.t. the number of valid matches incorrectly discarded).

$$SEN = \frac{TP}{TP + FN}$$
$$SPC = \frac{TN}{TN + FP}$$

Depending on the filtering algorithm, the number of discarded regions can be different – and consequently the overall computational cost. To evaluate the performance of a filter – and compare against others –, we commonly use the filtering efficiency (i.e. precision or positive predictive value; PPV) as the ratio between the number of matches found (i.e true positives) and the number of candidates produced by the algorithm (i.e. true positives + false positives).

$$f_e = \frac{TP}{TP + FP}$$

It is important to note that the maximum number of true positives reachable is independent of the algorithm – but related to the content of the text and the pattern. Therefore it is the number of FP that we would like to minimise – without increasing the filter complexity –, understood as the capability of the filter to reduce the noise (FP).

On text searching and filtering

It is widely understood that performance of filtering algorithms is very sensitive to the error rate. Under moderate or low error rates, they usually perform very well. However, under high error rates, filtering algorithms fail to effectively discard text regions and saturate the verification algorithm with candidates.

On the other hand, what is often overlooked is the relationship between the pattern length and the text itself. For instance, looking for all the occurrences of the pattern P_0 = "word" up to one mismatch – $e_r = 25\%$ – in the dictionary is deemed to be a challenging task as it will lead to multiple matches. However, searching for the pattern P_1 = "desoxyribonucleic" with the same error rate – five mismatches in total – can be an easier problem. Note that any valid substring aligning P_1 has to match at least 12 letters in the same relative order (with mismatches in between). Intuitively, this is much more restrictive than matching 3 letters in the case of P_0 = "word". In this case, longer patterns (w.r.t the size of the text) yield more information that can be used to restrict the search space.

Conversely, a very sharp statistician could highlight the fact that the neighbourhood of P_0 (i.e. N_h^5("desoxyribonucleic")) is much larger than the one of P_1 (i.e. N_h^1("word")). However, due to the limited size of the dictionary, it cannot contain all the words from N_h^5("desoxyribonucleic"). Hence, it is not the size of the search space that matters, but the amount of valid matches that can actually occur in the text. Therefore, in a hypothetical worst case, any sequence alignment algorithm would end up inspecting the whole dictionary in both cases. When daunted by the combinatorial nature of the problem, one should think how large a text should be as to contain all these possibilities. It is paramount to realise that the length of the text is the real limiting factor on the search. Compared to all the combinatorial possibilities, only a few happen to be matches in practice.

As a matter of fact, the English dictionary – and reference genomes – are far from having random content equally distributed. Thus, what really affects performance of filtering algorithms is the content of the text itself and the amount of candidates that the generation step can produce for a specific pattern. To illustrate furthermore this concept, let us think of a search for a numerical pattern like $P_2 =$ "3141". Presumably, the English dictionary is bias towards alphabetical content, leaving just a few occurrences of four digits to explore as viable matches.

5.1.1 Classification of filters. Unified filtering paradigm

In this thesis, the author would like to propose a different and more general view of filtering algorithms. As we define the problem of approximate string matching, the final goal is to reduce the search space – the whole text – to presumably a few matching regions. In this way, sequence alignment algorithms are mere filters which, given a candidate and a pattern, discriminate its alignment. Differences across algorithms strike in their accuracy and complexity. Thus, all these algorithms can be understood as filters whose main purpose is to reduce the search space.

Definition 5.1.1 (Filter)**.** Given $P, e, d()$, and $C = \{w_0, ..., w_p \mid w_i \in \Sigma^*\}$, a generic filter is defined as a binary classifier of the substrings contained in C.

$$\phi_{d,e}(C, P) = \{w_0, ..., u_w \in C\}$$

Hence, we consider classical pair-wise alignment approaches (e.g. dynamic programing algorithms, based on automatas, etc) as fully sensitive and specific filters ($FP = 0$ and $FN = 0$). In other words, they never fail to identify a valid match and perform without reporting noise. We classify these algorithms as **exact filters**.

Definition 5.1.2 (Exact filter)**.** Given $P, e, d()$, and $C = \{w_0, ..., w_p \mid w_i \in \Sigma^*\}$, an exact filter is defined as a filter such that:

$$\phi_{d,e}(C, P) = R$$

$$(\ d(w, P) \leq e \iff w \in R\ ), \forall w \in C$$

As opposed to exact filters, **approximate filters** are not fully specific and often report false positives in the form of candidates that do not match the pattern (i.e. generate noise). There are many approximate filtering algorithms to choose from with different trade-offs between filtering efficiency and computational cost. But in general, these filters tend to be very lightweight and perform very well in practice. In turn, we can classify approximate filters as **lossless** or **lossy** filters; depending on whether they retrieve all valid matches (i.e. true positives) or fail to report

complete results. Note that lossy filters lead to incomplete algorithms. Because of this, and in the context of this thesis, we will place our focus on lossless filters.

Traditionally, filtering algorithms are presented in either the context of online or indexed searches. However, it is of great interest to understand the duality of filters within these two classes and that many filters can be conveniently adapted to search both modes. Note that, given a full-text index I, an online filter $\phi_{d,e}$ on I can be converted into an online filter by enumerating all substrings of the text – using the index – and filtering them individually. Conversely, given a text $T_{0..n}$, an indexed filter $\phi_{d,e}$ on T can be converted into an online filter by locally indexing each text candidate $w_i = T_{i..j}$. Obviously, the whole point of specialised filters is to take advantage of particular scenarios towards better performance. Nevertheless, there are many cases of filters that can be efficiently applied in both scenarios and continue to preform equally well without loosing their very essence. (e.g. counting filters)

Full-text indexes are data structures that basically pre-process the text to provide fast and organised access to certain regions of it. For this reason, filtering conditions using full-text indexes are suitable to being adapted to both online and indexed search. However, in the case of limited indexes (i.e. indexes for which the original text cannot be fully retrieved), liked hashes or q-gram indexes, the structure of the index may prevent applying certain filtering rules. This is because the index has been design with a specialised purpose in mind and does not contain enough information to produce the original text back. For instance, in the case of q-gram indexes, the order relation between the individual q-grams is missing.

Nevertheless, it is not desirable for an algorithm to report matches beyond the specified distant thresholds. Consequently, results from approximate filters require further filtering – by an exact filter – to remove false positives and report only valid matches. In this way, we depict a filtering algorithm as an approximate filter – candidate generation step – coupled with an exact filter – verification step. This unified vision of sequence alignment algorithms as filters is the fundamental idea behind chaining filters.

5.1.2 Approximate filters. Pattern Partitioning Techniques

In this thesis, we mainly focus on filtering techniques based on pattern partitioning. The idea behind these methods is that, no matter how the errors are distributed in a matching string, we can always focus on the highly conserved parts of the read to reduce the combinatorial number of potential matches and reduce the search space.

Let us think about the size of the search space of a 100 base long pattern up to 4 errors (i.e. mismatches and indels). The number of combinations can be overwhelming. However, the problem becomes easier once we realise that any valid match has to contain at least 96 bases matching the pattern – and in the same relative order. What is more, no matter where the errors appear in the match, for any combination, at least a contiguous chunk of 24 bases has to match the pattern exactly. Surprisingly, by means of focusing on the highly conserved parts, the approximate string search has been reduced to an exact search of parts of the read.

This idea of reducing the problem to the search of chunks of the read is commonly known as a seeded search. In the context of sequence alignment, the concept of a "seed" has been overloaded with various definitions up to the point where a seed can denote any substring of the pattern – exact or with errors. The main purpose of a seed is to limit the search space to those parts of the read which contains it – filtering out the rest of the text. We denote as **exact seeds** to those matching exactly the pattern. Correspondingly, we denote as **approximate seeds** those which allow errors to match the pattern. In essence, approximate seeds describe a set of seeds that match a portion of the read allowing certain errors. This type of seed can better constrain the search space and reach more distant matches by specifying specific seed shapes containing errors.

Attending to how these techniques partition the pattern, we distinguish between static partitioning techniques and dynamic partitioning techniques. Following, the basic partitioning techniques are presented in an incremental fashion; from the most simple to the most complex and effective filters. The reader must bear in mind that the development of these techniques has been strongly influenced by research on new index data structures. Consequently, some techniques are currently of no practical use, yet are still quite interesting from an algorithmic standpoint.

5.2 Exact filters. Candidate Verification Techniques

As explained earlier, exact filters are algorithms capable of computing whether a given string candidate w aligns a given pattern P (within a maximum distant threshold e and for a given distance metric $d()$). These filters are full, sensitive and specific; never fail to report a valid match ($FP = 0$) and never report an invalid match as so ($FN = 0$).

Most classical pairwise alignment algorithms fall into this category. Extensive research has been done in this area for many years [Needleman and Wunsch, 1970; Sankoff, 1972; Sellers, 1974; Wagner and Fischer, 1974; Navarro, 2001]. In this way, dynamic programming (DP) approaches [Kukich, 1992; Ukkonen, 1985a,b] explore the whole text and output only the matching positions (true positives). These types of algorithm should be classified as exact-filters as they return all matches with zero false positive rates. Despite being the most flexible solutions and delivering the best filtering efficiency, most of them are quite computationally expensive. This is the main reason to explore other kinds of filter more computationally efficient, though perhaps less accurate as they may report false positives.

Since the formulation of the first algorithms to compute the pair-wise edit distance using DP, some alternative formulations of the problem and many improved solutions have been proposed.

5.2.1 Formulations of the problem

Notably, some redefinitions of the problem have lead to novel theoretical results, which in turn have inspired alternative algorithms.

One of the most notorious revisions is the approach given by [Ukkonen, 1985a] where the problem is formulated as finding the **shortest path problem on a graph** built on the text and the pattern. In the paper, Ukkonen presented basic theoretical properties of the DP table still exploited in modern algorithms. He formally proves that diagonals of the DP table are monotonically increasing from the upper-left to the lower-right cells. Moreover, adjacent cells value can differ at most by one. As a result, many algorithms exploit these properties towards computing only the band of the DP table where alignments up to e edit operations are confined. As a result, they drastically reduce the amount of computations needed to calculate the edit distance between two strings.

Furthermore, these properties are exploited so as to compute edit distance e in $\mathcal{O}(e^2)$ within algorithms denoted as **diagonal transition algorithms**. These algorithms aim to compute in constant time the cells where the diagonal values are incremented. These diagonals – so-called "strokes" – model segments that match exactly between two strings. Afterwards, strokes are joined to compute the optimal path which leads to the optimal alignment. In [Landau and Vishkin, 1989] Landau and Vishkin propose one of the first algorithms of this kind. Another remarkable algorithm exploiting these properties is proposed by Myers in [Myers, 1986]. Myers' $\mathcal{O}(nd)$ algorithm – unlike many others – is able to report the optimal alignment in $\mathcal{O}(|P|e)$ time. Many other algorithms have been proposed in this line of research; the interested reader is refered to Navarro [2001] for more information.

Another formulation of the problem is given in the form of **searching with an automaton** Navarro [2001]. Using a non-deterministic automaton (NFA) transformation operations are modelled as transitions over a finite number of states. Each state is associated with a number of errors and a position of the pattern aligned. By introducing the letters of given stringd into the NFA, the final state becomes active if the string aligns the pattern. This computation can be done by

means of simulating the NFA or by transforming the NFA into the corresponding deterministic automaton (DFA).

This idea was proposed in early publications like [Ukkonen, 1985b] using a deterministic automaton. Later on, the computation of the automaton was further described and improved [Baeza-Yates and G. Navarro, 1999; Wu and Manber, 1992]. Depending on the distance metric employed, there are solutions – like the levenshtein automaton [Schulz and Mihov, 2002] – that perform fast in real scenarios. However, the challenging aspect of this approach is the exponential number of possible states that arises as the error increases. Hence, these algorithms are only practical for reasonably low error rates.

5.2.2 Accelerating computations

I Block-wise computations

One of the most widely know approaches to improving the computation of the DP table is the so-called **Four-Russians technique** [Kronrod et al., 1970] – though only one of the authors was actually Russian. Their approach pre-computes all possible combinations of small DP tables of size b (i.e. look-up table of all possible DP blocks of size b). Afterwards the algorithms proceed block-wise computing the full matrix taking advantage of precomputed blocks; thus gaining a factor of b. Since neighbouring cells differ at most by one, the DP is encoded using horizontal – or vertical – differences ($\delta_e(i,j) \in \{-1,0,+1\}$). This makes a total of $(3|\Sigma|)^{2b}$ possible input combinations to each precomputed block. Choosing $b = \mathcal{O}(\log(|P|))$ the complexity gets reduced to $\mathcal{O}(\frac{|P|*|T|}{\log(|P|)})$. In short, assuming equal text and pattern length n, the complexity drops from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^2/\log(n))$.

II Bit-parallel algorithms

Another very successful technique employed to accelerate the computation of pair-wise alignment is to exploit the intrinsic parallelism of the computer instructions. Nowadays, any computer architecture offers simple arithmetical and logical instructions that can be understood as parallel operators over a set of bits. These operations can be performed using 32-64 bit words and, depending on the architecture, using SIMD instruction from 128 to 512 bits long.

Here, the key insight is to encode the problem conveniently so as to benefit from simple bit-wise instructions. Algorithms based on this idea are able to compute several cells of the DP table – or states of the NFA – at once, enabling much faster computation of pair-wise alignments.

Most notably, Myers in [Myers, 1999] proposed the computation of the DP matrix encoding columns using bit-words and using bit-wise operations to fill the matrix. The so-called Bit-Parallel Myers (BPM) takes advantage of the intrinsic parallelism within bit-wise operations of any computer architecture (i.e logical, shift, and addition operations). Algorithm 5.3 shows the pseudo C-code of the BPM algorithm.

In this approach, columns of the DP matrix are encoded in differences among themselves. As explained before, this reduces the number of possible values of each cell to $-1, 0, +1$. Then, each cell of the matrix is modelled as a logic block. In this way, the function "advance block" takes a column of the DP and produces the next column using bit-wise operations (Algorithm 5.2). This process iterates so as to compute the whole matrix chunk-wise meanwhile the cells of each column are computed intrinsically in parallel. Low-level details of the algorithm make it one of the most quick witted bit-wise algorithms of its type [Myers, 1999; Hyyrö, 2003].

```
1  int8_t T_hout_64[2][2] = {{0,-1},{1,1}};
2  uint8_t P_hin_64[3] = {0, 0, 1L};
3  uint8_t N_hin_64[3] = {1L, 0, 0};
4  int8_t bpm_advance_block(
5      uint64_t Eq,
6      const uint64_t mask,
7      uint64_t Pv,
8      uint64_t Mv,
9      const int8_t hin,
10     uint64_t* const Pv_out,
11       uint64_t* const Mv_out) {
12   uint64_t Ph, Mh;
13   uint64_t Xv, Xh;
14   int8_t hout=0;
15   // Compute H-vectors
16   Xv = Eq | Mv;
17   Eq |= N_hin_64[hin];
18   Xh = (((Eq & Pv) + Pv) ^ Pv) | Eq;
19   Ph = Mv | ~(Xh | Pv);
20   Mh = Pv & Xh;
21   // Compute carry
22   hout += T_hout_64[(Ph & mask)!=0][(Mh & mask)!=0];
23   Ph <<= 1;
24   Mh <<= 1;
25   Mh |= N_hin_64[hin];
26   Ph |= P_hin_64[hin];
27   // Compute V-vectors
28   Pv = Mh | ~(Xv | Ph);
29   Mv = Ph & Xv;
30   // Return out-vectors and carry
31   *Pv_out=Pv;
32   *Mv_out=Mv;
33   return hout;
34 }
```

Figure 5.2: BPM Advance Block Implementation

The BPM algorithm has a worst-case complexity of $\mathcal{O}(|P||w|/b)$ – where b is the length of the computer word used – and $\mathcal{O}(e|w|/b)$ on average as cut-off conditions can be easily implemented within. What is more, this algorithm performs better than others of its kind with long patterns. For this reason, it is often the choice of many real mappers [Marco-Sola et al., 2012; Siragusa et al., 2013] based on filtering to verify candidates. Additionally, in [Hyyrö and Navarro, 2002] an improved version of the BPM algorithm was proposed so as to reduce the overall bit-wise operations performed.

III SIMD algorithms

Using general scores, some techniques have been proposed to exploit SIMD instructions in order to compute several cells of the DP matrix at once. The key idea is to allocate several cells into a computer register and compute the DP matrix by blocks [Alpern et al., 1995]. In this way, these algorithms differ in that they arrange the cells into the registers (i.e. by columns [Rognes and Seeberg, 2000], anti-diagonals [Wozniak, 1997] or interleaved [Rognes, 2011]). The main challenge of these methods is to incorporate all the dependencies in the SIMD computation without introducing mayor slowdowns. In this way, Farrar's approach in [Farrar, 2007] proves to be the most efficient, achieving from 2x-8x folds against previous approaches. In practice, this

```
uint64_t bpm_compute_edit_distance() {
  // Initialize search
  bpm_init_search(...);
  // Advance in DP-bit_encoded matrix
  uint64_t min_score = DISTANCE_INF; // -1
  uint64_t text_pos;
  for (text_pos=0;text_pos<text_length;++text_pos) {
    // Fetch next character
    const uint8_t enc_char = text[text_pos];
    // Advance all blocks
    int8_t carry;
    uint64_t i;
    for (i=0,carry=0;i<num_words;++i) {
      uint64_t* const Py = P+i;
      uint64_t* const My = M+i;
      carry = bpm_advance_block(
        PEQ[BPM_PATTERN_PEQ_IDX(i,enc_char)],
        level_mask[i],*Py,*My,carry+1,Py,My);
      score[i] += carry;
    }
    // Check match
    if (score[num_words-1] < min_score) {
      min_score = score[num_words-1];
    }
  }
  // Return edit distance
  return min_score;
}
```

Figure 5.3: BPM Implementation

method is widely used and some mappers based on SWG distance implement it [Langmead and Salzberg, 2012; Li, 2013].

5.2.3 Approximated tiled computations

In a more practical context, as the pattern gets longer, most of these algorithms become truly impractical. Despite many efforts towards reducing their complexity, the final running time gets dominated by their quadratic underlying complexity. Moreover, in order to compute the complete alignment transcript, the full DP matrix has to be stored in memory; or at least the band for a given e threshold. At the same time, note that the maximum size of the band may not be known; in which case many algorithms would make a blind estimation of it.

Because of the tight dependencies between the cells of the DP matrix, no lossless partition of the computations seems feasible. Here, we propose to give up on the idea of computing the exact edit distance, towards establishing accurate minimum and maximum edit distance bounds using a tiled partition of the problem. In this way, we tile the DP matrix in horizontal tiles of height h. Figure 5.4 shows a bounded diagram depicting the tiles and how they define the different regions of the band to compute.

In this way, we propose independently computing the region of the band overlapping each tile. Given n tiles, we will compute the optimum alignment in each one $\{a_{t_0}, ..., a_{t_{n-1}}\}$, starting and ending anywhere in the frontiers. Note that nothing guarantees that the tiled optimum alignments a_{t_i} are connected. However, we can easily establish bounds on the global-optimum alignment $\delta_e(P, T)$.

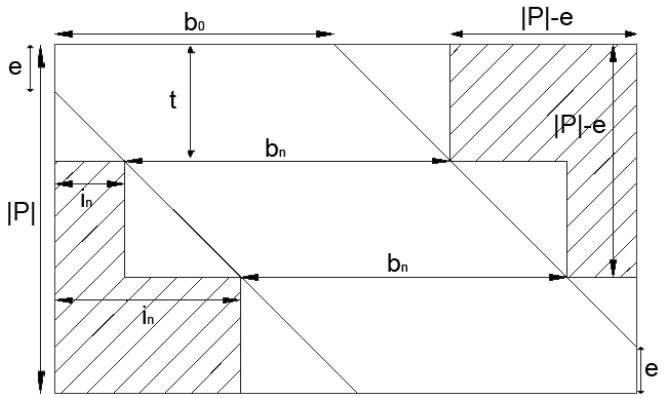


Figure 5.4: Tiled dynamic programing table and dimensions

Lemma 5.2.1 (Minimum edit distance). Given P, T, and $\{a_{t_0}, ..., a_{t_{n-1}}\}$ the optimum alignment for each tile. The global-optimum alignment $\delta_e(P,T)$ distance is greater or equal than the sum of the distances of each tile alignment.

$$\sum_{i=0}^{n-1} \delta_e(a_{t_i}) \leq \delta_e(P,T)$$

Proof. If the distance of global-optimum alignment $\delta_e(P,T)$ was lower than the sum of the optimum alignments of the individual tiles, then there is at least one a_{t_i} which is not optimal (given by the $\delta_e(P,T)$ between the boundaries of the tile). □

Lemma 5.2.2 (Maximum edit distance). Given P, T, $\{a_{t_0}, ..., a_{t_{n-1}}\}$ the optimum alignment for each tile, and $\{(i_{t_0}, f_{t_0}), ..., (i_{t_{n-1}}, f_{t_{n-1}})\}$ the corresponding initial and final position of each tiled alignment. The global-optimum alignment $\delta_e(P,T)$ distance is lower or equal than the sum of the distances of each tile alignment plus the cost of linking the initial and final positions of each tile.

$$\delta_e(P,T) \leq \sum_{i=0}^{n-1} \delta_e(a_{t_i}) + \sum_{i=1}^{n-1} |f_{t_{i-1}} - i_{t_i}|$$

Proof. Forcing to link each tile alignment with as many deletions or insertions as needed gives a valid alignment. Therefore, the global-optimum alignment $\delta_e(P,T)$ can only improve on such alignment. □

The advantage of the tiled approach to enable independent – and potentially parallel – computations of each alignment tile. Also, the overall memory requirements can be reduced to those of one single tile. But more importantly, note that if a given tile minimum edit distance bound is greater that the overall error allowed (e), a filter based on this method could directly filter out the corresponding candidate. However, the approach is no longer an exact filter – no longer guaranteeing full specific results –, in practice it serves as a very accurate bounding tool on the exact edit distance. Moreover, it establishes an accurate estimation of the maximum band that posterior full-fledged DP algorithms can take into account to derive the actual global-optimum alignment and save computations. Overall, tiled computations will not only help to improve filter performance, but will also allow one to derive distance bounds on the filtering candidates so as to rank them and consider which candidates are promising and which can be left out (section 5.5.3).

5.2.4 Embedding distance metrics

As a matter of fact, different distance metrics strongly condition the performance of these algorithms. Most of the optimisation techniques presented so far are only compatible with edit distance – or close variations of it. As a result, practical running times between fast edit distance algorithms – like BPM – and biological inspired algorithms – like Smith-Waterman-Gotoh (SWG) – differ by orders of magnitude. In spite of this, the latter are widely preferred for their flexibility and accurate modelling of the biological problem.

As opposed to the commonly held belief, using biological distances does not rule out the application of other exact filter techniques to enhance approximate matching searches. In this way, we propose embedding biological metric distances into simpler metrics – like edit distance – so as to bound distances using faster algorithms. Ultimately, biological metric based algorithms will have to be used to derive the exact distance. However, these computations can be greatly constrained using bounds derived using edit distance based algorithms.

Upper bound in embedded metric spaces

To illustrate this idea, we will assume a widely used SWG metric as the biologically inspired distance using gap opening score g_o, gap extension score g_e, matching score m_s, and mismatch score x_s (irrespectively of the mismatched nucleotide). Then, let A_e be the optimal alignment between the input strings v and w, so that $\delta_e(v, w) = e$. Note that the transformation operations (i.e. mismatches and indels) are the same for both metrics. Then, A_e constitutes a valid alignment in both metric spaces (i.e. A_{swg}). However, A_{swg} might now be optimum in SWG-space. Nevertheless, A_{swg} represents an upper bound on the distance – or lower bound on the score – of the optimum alignment using SWG.

Lemma 5.2.3 (Upper bound on the distance in embedded metric spaces). Given v, w, A_e the optimum alignment in edit-space, and another metric space Φ defined with the same transformation operations as the edit-space. Then the optimum alignment in Φ-space is upper bounded by A_e scored in Φ-space ($max_\Phi(v, w)$).

$$\delta_\Phi(v, w) \leq \delta_\Phi(A_e) = max_\Phi(v, w)$$

Proof. As edit-space and Φ-space are defined using the same transformation operations, A_e is a valid alignment in both spaces and thus establishes an upper bound on the best distance in Φ-space. □

Lower bound in embedded metric spaces

Furthermore, let us assume A_e is the optimum edit alignment between v and w ($|v| = |w| = 100$) containing 2 mismatches. Also let us assume a widely used scoring scheme $\phi = (m_s, x_s, g_o, g_e) = (1, -4, -6, -1)$. In that case, the optimum alignment in SWG-space is bounded by the score of A_{swg} in SWG-space (i.e. $\delta_{swg}(v, w) \geq \delta_{swg}(A_{swg}) = 90$). But also, it is easy to see that in SWG-space there are only two potential alignments that score better.

$$O_0 = 99 \times m_s + g_o = 93 \qquad (a)$$
$$O_1 = 98 \times m_s + g_o + g_e = 91 \qquad (b)$$

First, note that case (a) is not feasible as it involves less edit operations than the optimum A_e. Therefore we can derive a lower bound on the distance using this observation.

Lemma 5.2.4 (Lower bound on the distance in embedded metric spaces). Given v, w, A_e, and another metric space Φ defined with the same transformation operations as the edit-space. Then the optimum alignment in Φ-space is lower bounded by an alignment with at least e or more transformation operations.

$$min_\Phi(v,w) = \min\left\{\delta_\Phi(A_i) \text{ s.t. } |A_i| \geq e\right\} \leq \delta_\Phi(v,w)$$

Remark. In practice, the problem of finding this potential alignment that minimises the distance with at least e operations, can be reduced to the knapsack problem where the items are the individual transformation operations. This problem is broadly known to have a pseudo-polynomial solution using a dynamic programming. In cases like the SWG distance $\phi = (m_s, x_s, g_s)$, the computation is reduced to maximise the cost function.

$$\max_{gaps+misms \geq e}\left\{m_s(|v| - gaps - misms) - x_s(misms) - g_s(gaps)\right\}$$

In conclusion, we can bound the optimum alignment distance in a metric space Φ by computing the optimum alignment in the edit distance space.

$$min_\Phi(v,w) \leq \delta_\Phi(v,w) \leq max_\Phi(v,w)$$

Computing the effective band

Traditionally, in the context of convoluted metric spaces – like SWG –, the band used to compute the DP table is usually set beforehand attending to a subjective estimation on the worst case expected. Most of the time, it is parametrised to be set by the user according to the problem at hand. However, knowing that the optimal alignment can be bounded using edit distance computations, it seems reasonable to compute the maximum band possible for each case. Recalling previously presented cases, note that only case (b) is plausible and therefore the maximum number of transformation operations is 2. Hence, a banded SWG algorithm using $b = 2$ is guaranteed to report the optimal SWG alignment.

Lemma 5.2.5 (Maximum number of transformation operations). Given v, w, A_e, and Φ metric space defined with the same transformation operations as the edit-space. The maximum number of transformation operations b that the optimum alignment O_Φ can have is bounded.

$$b = \max\left\{|A_i| \text{ , } \forall A_i \text{ s.t. } \delta_\Phi(A_i) \leq \delta_\Phi(A_e)\right\}$$

In this way, using fast edit distance algorithms we can estimate the maximum band needed to compute the optimal alignment in other metric spaces. As a result, we can assess that the band is not a fixed parameter that needs to be determined in advanced, but a magnitude that can be effectively computed or derived as a byproduct of edit-distance based filters.

Ranking candidates and δ-strata groups

Moreover, embedding distance metrics can serve as to discard potentially distant matches uninteresting to the search. Given a set of candidates $C = \{w_0, ..., w_p\}$ and their corresponding edit distances $\{\delta_e(w_0), ..., \delta_e(w_p)\}$ – computed using any exact filter –, we can rank all the candidates according to their potential SWG score.

$$\{\max_{swg}(w_0), ..., \max_{swg}(w_p)\}$$

To illustrate the relevance of these bounds let us step back for a moment and imagine that we want to determine the δ_{swg}-strata group under the SWG distance. As mentioned before, we require a sufficiently complete set of candidates to accurately assess this magnitude. However, it only suffices to compute the SWG score of the two best matches to resolve the actual δ_{swg}-strata group. In this scenario we could align all candidates using a fast edit distance algorithm. Then, we could proceed in order, computing the actual SWG distance of each candidate. Before SWG-aligning each candidate candidate, we could test if the best possible score could improve the best two matches found so far. If not, we could directly discard that candidate and avoid the costly the computation of its SWG distance.

Furthermore, ranking leads to the natural idea that the maximum error allowed e does not have to be a fixed parameter, but can be derived as the search progresses. Basically, when two matches have been found, there is no need to explore more distant matches – but potential matches in between. In this way, the maximum e is approximated as a byproduct of the search.

This is the very essence of embedding distance metrics and bounding alignments. As the search progresses, not all candidates need to be aligned. It simply suffices to make an accurate enough bound of its distance so as to safely discard irrelevant alignments. In case we are forced to compute the actual alignment, bounds can serve to safely avoid unnecessary computations outside the band.

5.3 Approximate filters. Static partitioning techniques

Static partitioning techniques basically define a fixed partition of the pattern agnostic of the content of the text and the pattern itself. In brief, these techniques are based on counting substrings in both the pattern and the candidate to assess similarity under certain error tolerance. In this way, these techniques are able to derive fixed sets of conditions that the candidates must satisfy in order to match the pattern.

5.3.1 Counting filters

In a nutshell, counting filters work in two basic steps. First, they build a profile based on counting occurrences of q-grams in the pattern. Afterwards, they perform the same counting procedure on the candidate text. If the counting profile is similar – depending on the maximum distance allowed – then we can conclude with some confidence that the pattern and the candidate match. These techniques stand out because they are simple, yet very fast and achieve good filtration rates.

Counting characters

In the context of online filtering, the most basic version of this family is based on counting characters [Jokinen et al., 1996]. This simple filter holds the very essence of all the filters in this family. Basically, we first count the characters in the pattern and we build a histogram of the number of times that each letter of the alphabet appears in it. Intuitively, any region of the text matching the pattern has to contain a similar count of characters. Therefore, for each region of the text we count characters. In general, any text matching the pattern with e errors must have at least $|P| - e$ characters from the pattern. Note that if the pattern and the text match perfectly, the counts must be exactly the same. Hence, we can state that if the count differs in at most e characters the candidate can be a valid match, otherwise it is not possible and can be filtered out. Note that the filter does not consider the relative ordering of the characters. Therefore it is easy to see that it can potentially generate false positives.

Lemma 5.3.1 (Counting lemma). Given P, e, and $d()$, the counting filter is defined as a binary classifier, modelling a necessary condition that the candidate w must satisfy in order to align P up to error e.

$$\phi_{d,e}(w, P) = \begin{cases} 1 & \text{if } \sum_{c \in P} (max\{occ(c, P) - occ(c, w), 0\}) < e \\ 0 & \text{otherwise} \end{cases}$$

The computation of the pattern profile is $\mathcal{O}(|P|)$ and assumed to be amortised as we check multiple candidates for each single pattern. In that way, the filter complexity is dominated by the cost of checking the candidates which operates in linear time w.r.t. the candidate length ($\mathcal{O}(|w|)$).

Despite its simplicity, this simple filter can yield high filtration efficiency rates depending on the pattern length and the alphabet size. Note that a 26 character alphabet is much more diverse and restrictive than the DNA alphabet (i.e. $\Sigma = \{ACGTN\}$). In practice, this filter is rarely used and it mainly serves as an introduction to logical extensions of this idea.

Counting q-grams

At this point, it seems natural to extend the idea of counting characters to count q-grams. In fact, profiling larger components of the pattern increases the specificity of the filter. Q-gram filters are based on counting q-grams to discard candidates than contain less than a given number or q-gram occurrences [Jokinen and Ukkonen, 1991; Ukkonen, 1992]. The basic idea behind q-gram filters is depicted in figure 5.3.2. As the figure shows, the maximum number of q-grams that each potential error can modify depends on the q-gram length. Therefore, for a given maximum error e there is a minimum number of q-grams shared by the pattern and any potential match.

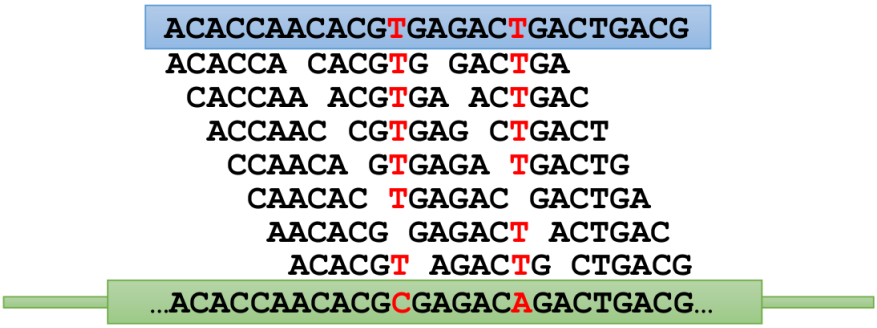


Figure 5.5: Q-gram lemma. Counting q-grams in a candidate

Lemma 5.3.2 (q-gram lemma). Given P, e, and $d()$, if the candidate w aligns the pattern P (i.e. $d(w, P) \leq e$) then P and w share at least $|w| + 1 - |q| \times (e + 1)$ common q-grams.

In their first formulation, q-gram filters were used in the context of online searches. The linear cost $\mathcal{O}(|w|)$ of counting the q-grams of the pattern (i.e. histogram or profile creating) is amortised by the cost of subsequent computations of the q-gram lemma on the candidates. For that, the algorithm applies the q-gram lemma on a sliding window that traverses every candidate and determines if the candidate is accepted or filtered out. Counters are reused between shifts of the sliding window and therefore the algorithm takes $\mathcal{O}(|w|)$ in the worst case.

e=3%	$\|P\| = 75$	$\|P\| = 100$	$\|P\| = 150$	$\|P\| = 500$
$\|q\| = 3$	85.92%	87.98%	88.84%	70.63%
$\|q\| = 4$	**89.67%**	91.76%	92.16%	79.72%
$\|q\| = 5$	89.45%	**92.23%**	**92.90%**	83.70%
$\|q\| = 6$	87.83%	91.66%	92.88%	85.07%
$\|q\| = 7$	85.48%	90.60%	92.57%	**85.37%**

Table 5.2: Q-gram filter efficiency on candidates drawn from the human genome (GRCh37) allowing up to 3% error rate

Filtration efficiency of Q-gram filters strongly depends of the distribution of the candidates (i.e. string content), the length of the pattern, the length of the q-gram, and the maximum error. As with many other filtering techniques, there is little benefit in benchmarking its efficiency for generic random texts. Instead, it is more useful to base experiments on real application data. That is, candidates generated by a real mapping tool (i.e. GEM). For that, in table 5.2 and 5.3 we present a small benchmark to capture the potential filtration efficiency of this family of filters. In general, with longer pattern lengths and longer q-gram lengths, the filter achieves better filtration efficiency. However, reaching a certain pattern length, the filtration efficiency drops as q-grams at that length are not specific enough. Also, as the error rate increases, the filtration efficiency of q-gram filters decreases as it has to account for more errors.

It is important to note that these filters are greatly limited by the g-gram length as the memory footprint increases exponentially requiring $4^{|q|}$ counters (e.g. Using 32-bit counters, for $q = 7$ the counters occupy 64KB). Also note that memory consumption is not the main limitation, but the penalties derived from randomly accessing potentially large regions of memory (e.g. cache misses, page faults, etc).

e=6%	$\|P\| = 75$	$\|P\| = 100$	$\|P\| = 150$	$\|P\| = 500$
$\|q\| = 3$	66.51%	67.17%	75.07%	81.32%
$\|q\| = 4$	**74.74%**	79.45%	87.26%	86.53%
$\|q\| = 5$	74.53%	**80.96%**	**89.72%**	88.17%
$\|q\| = 6$	70.72%	78.72%	89.57%	88.70%
$\|q\| = 7$	65.15%	74.71%	88.36%	**88.78%**

Table 5.3: Q-gram filter efficiency on candidates drawn from the human genome (GRCh37) allowing up to 6% error rate

Later on, this technique was adapted to be used in conjunction with indexes. Many q-gram specialised indexes have been proposed to accelerate computing the q-gram profile of certain regions of the reference text (e.g. overlapping blocks of text [Burkhardt and Kärkkäinen, 2003]). In any case, the basics of the approach remain the same; spotting a region of the text or candidate whose q-gram profile matches the one of the pattern as the g-gram lemma states. However, the indexed formulation of the algorithm allows skipping entire regions of the reference text at the expense of accesses to the index – and memory space devoted to the index. Note that in these cases the cost of building the index is also considered amortised.

Additionally, there have been some proposals trying to acknowledge the relative order between q-grams [Sutinen and Tarhio, 2004]. In this way, resulting filters turn out to be more accurate and report less false positives – at the cost of extra computations.

It is also worth mentioning a variation of regular q-grams which allow gaps in the q-grams (i.e. approximate seeds) and increase the overall efficiency of q-gram counting approaches [Burkhardt and Kärkkäinen, 2003]. Nevertheless, gaped q-grams have proven to be quite a theoretically challenging approach as opposed to the simplicity of ungaped q-grams [Rasmussen et al., 2006; Lee et al., 2007].

As a matter of fact, q-gram filters are often used in real tools for sequence alignment. Notable examples are SWIFT [Rasmussen et al., 2006], STELLAR [Kehr et al., 2011], RazerS [Weese et al., 2009], and RazerS 3 [Weese et al., 2012]. But also, there have been some implementations based on gaped q-grams like QUASAR [Burkhardt et al., 1999] or SHRiMP 2 [David et al., 2011]. In fact, many practical tools incorporate in different ways the basics of q-gram counting; namely Mr. Fast [Alkan et al., 2009], Mrs. Fast [Hach et al., 2010], Hobbes [Ahmadi et al., 2012], SNAP [Zaharia et al., 2011], and many others.

Counting q-samples

In the context of index searches, modern index data structures enable fast access to all q-gram occurrences in the text for an arbitrary q. In turn, these indexes allow the querying of larger q-grams which are deemed more specific than shorter ones. In this scenario, the main algorithmic bottleneck is computing the intersection of positions associated to each q-gram – towards retrieving regions that satisfy the q-gram lemma. This leads to another category of filtering algorithms that aim for larger and more specific q-grams, that produce fewer candidate positions which can be intersected easily.

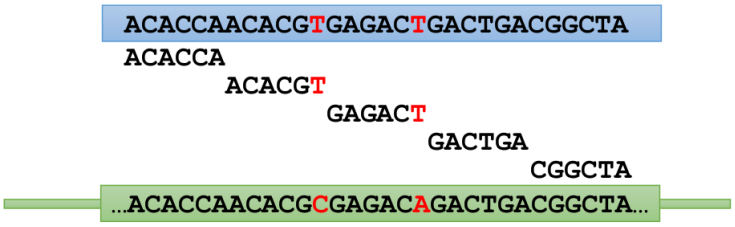


Figure 5.6: Q-sample lemma. Counting Q-samples in a candidate

In some contexts these filters are referred as h-sample filters. Basically, they extract contiguous samples of the pattern (q-samples), query them into the index and join the resulting sets of text positions so as to generate all the potentially matching candidates. As the q-samples are contiguous, each potential error in the match removes the occurrence of a single q-sample in the candidate. Figure 5.6 shows the mechanics of q-samples filtration. In this way, we can easily derive a necessary condition for a candidate to match the pattern; the q-sample lemma.

Lemma 5.3.3 (Q-sample lemma). Let e be the maximum distance allowed and $d()$ a distance metric. Given a pattern $P = f_0...f_h$ as the concatenation of h q-samples ($h > e$), if the candidate w aligns the pattern P (i.e. $d(w, P) \leq e$) then P and w share at least $h - e$ common q-samples.

In general, q-sample filters – and those derived from this idea – require $\lfloor |w|/q \rfloor$ accesses to the index plus the time needed to sort the q-sample positions $\mathcal{O}(|C| \log(|C|))$ (being C the set of all q-sample candidate positions) and filter out those that do not satisfy the q-sample lemma. Note that theoretical complexity bounds do not capture how sensitive these kind of approaches are to the total number of candidates reported by each q-sample. Depending on the length of the q-sample and the content of the reference text, the total amount of candidates involved could range from a few to large parts of the reference.

Similarly, the q-sample lemma can be formulated for online searching. In this case, an exact online search is used to lookup for a minimum number of q-samples in the candidate needed to match the pattern. In some proposals, tries or suffix-trees of the pattern are employed to accelerate the search of the q-samples. In any case, the running time is proportional to the candidate length $\mathcal{O}(|w|)$.

5.3.2 Factor filters

As q-samples increase in length, the total amount of samples decreases up to the point we reach the case where $h = e + 1$. In that case, we only need one sample occurrence in a region of the text to consider it a candidate. Filters falling in this particular case are known as factor filters. Provided a full-text index that can query arbitrarily long chunks of the pattern, these filters aim to minimise the total number of candidates produced without losing sensitivity.

For a given e and P, factor filters divide the pattern into $e + 1$ contiguous chunks (factors). Afterwards the factors are queried against the index to directly generate all candidate positions – that already satisfy the q-sample lemma. Note that in the worst case we can have matches containing e errors. Intuitively, if we partition the pattern into $e + 1$ factors, no matter how the errors are distributed, every valid match will always contain a chunk matching without errors (i.e. exact match). These filters are commonly presented as an example of how to reduce approximate searches to exact searches (i.e. without exploring all possible error events combinations on the pattern). This is the very principle behind factor filters which guarantees complete results and it

is usually explained in terms of the pigeonhole principle. Instantiated in this particular context, if m pigeons were to be distributed in $m+1$ holes, then at least one hole always remains empty.

Figure 5.7: Factor filter up to 2 errors (3 factors; at least 1 exact matching)

Lemma 5.3.4 (Factors lemma). Given P, w, e, and $d()$, such that $d(P,w) \leq e$, if P is partitioned into $e+1$ non-overlapping substrings $P = f_0, ..., f_e$, then at least one f_i occurs as a factor of w [Baeza-Yates and Perleberg, 1992].

Note that this lemma imposes no restriction on the length of the individual factors. Therefore any partition of $e+1$ non-overlapping factors will be fully sensitive up to e errors. In this way, different partitions for the same search depth will produce different numbers of candidates depending on the content of the text. In order to get a grasp on the expected number of candidates generated by these filters, let T be a random text from the alphabet Σ following a Bernoulli model. In that case, let F be the random variable modelling the number of occurrences of a given factor f. Then the probability of occurrace for a given character is $1/|\Sigma| = 1/\sigma$ (assuming i.i.d). Thus, the probability of the factor occurring in a given position of the text is given by:

$$Pr[F > 0] = \frac{1}{\sigma^{|f|}}$$

Consequently, the expected number of occurrences of the factor f in the text T is:

$$E[F] = \sum_{i=0}^{|T|-|f|} Pr[F > 0] \leq \frac{|T|}{\sigma^{|f|}}$$

In this way, it is easy to see that the number of candidates grows exponentially as the factor reduces in length. In practice, it is very important to minimise the number of false positives passed on to the next filter (e.g. a computationally expensive exact filter). In order to achieve balanced filtering algorithms, factors have to be selected appropriately as to avoid unbalanced cases (i.e. unspecific factors that potentially produce impractical amounts of candidates). Thus, if no further information is used, factors are usually configured by dividing the pattern into $e+1$ equal parts. In the case of a random text, if all factors have the same length then all factors have the same probability of occurring in the text. In turn, this can be proven to be the expected optimal partition.

$$E[Candidates] = min\left\{\sum_{f_i \in P} E[F_i]\right\}$$

However, the reader should note that assuming random distribution in the reference text is far from being a realistic assumption.

Another powerful observation is that factors have to be non-overlapping but not necessarily contiguous. Leaving gaps between factors might seem odd at first, however it turns out to be quite handy in the case of sequenced reads containing uncalled regions (e.g. stretches of Ns acting as wildcards). In this case, factors can be picked from the known regions and the factor lemma still holds. Hence, factor filters can "recover" noisy sequenced reads by applying filtering restrictions on the high quality parts of the reads and filling uncalled segments during posterior filtering verifications.

5.3.3 Intermediate partitioning filters

As the error increases, factor filters produce partitions of smaller factors. For that reason, factor filters are strongly limited by the minimum factor length for which the number of candidates produced becomes impractical. To overcome this limitation and scale up to higher error rates, factor filters are generalised towards using approximate factors. In this way, intermediate partitioning filters [Myers, 1994; Navarro et al., 2001] yield more specific seeds meanwhile performing full-sensitive searches. So as to report complete sets of candidates, these filters follow the pigeonhole principle for any number of pigeons and holes.

Lemma 5.3.5 (Pigeonhole principle). If m pigeons are in n holes, then at least one hole contains $\lceil \frac{m}{n} \rceil$ pigeons. Likewise, at least one hole contains $\lfloor \frac{m}{n} \rfloor$ pigeons.

Lemma 5.3.6 (Intermediate partitioning lemma). Given P, w, e, and $d()$ (s.t. $d(P, w) \leq e$), if P is partitioned into s non-overlapping substrings $P = f_0, ..., f_{s-1}$ s.t. $1 \leq s \leq e + 1$, then at least one f_i occurs as a factor of w with distance $\lfloor \frac{e}{s} \rfloor$

Essentially, this lemma holds the simple idea of searching for approximate matches starting from the least distant regions of the match. In other words, searching first for the better conserved parts of the sequenced read (see figure 5.8).

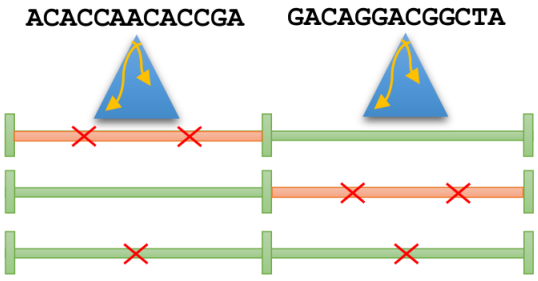


Figure 5.8: Intermediate partitioning filter up to 2 errors (2 factors; at least one with 1 or less errors)

From a more general perspective, intermediate partitioning not only poses a necessary condition for a given candidate to match the pattern, but also provides a method for preforming arbitrary deep searches by means of distributing errors across factors. In any case, approximate searches of the individual factors r_i up to a given number of errors e_i is yet another problem of approximate string matching (i.e. reduction to smaller approximate searches). However, these smaller approximate searches are often irreducible to recursive filtering searches. The even smaller factors will saturate and lead to an unfeasible number of candidates. Because of this, these approximate factor

searches are tackled by enumerating the search space of each r_i up to e_i errors (i.e. neighbourhood search). The hope is that the moderate error degree of each factor allows feasible neighbourhood searches while generating not too many candidates.

This approach offers a wide spectrum of choices to achieve a given search depth. An intermediate partitioning algorithm can vary on (a) the number of non-overlapping factors dividing the read, (b) the length of those factors, and (c) the number of errors assigned to each factor. Note that most descriptions of the method overlook the possibility of using variable length factors and assume factors equally distributed over the pattern by default. In this way, the number of factors used and the assignment of errors to each factor will determine the error partition of the filter.

Definition 5.3.1 (Error partition)**.** For a given pattern P, an error partition is a vector of positive integers

$$P^e = (e_0, ..., e_{s-1})$$

assigning errors to each factor of the pattern.

Lemma 5.3.7 (Completeness of an error partition)*.* Given the error partition

$$P^e = (e_0, ..., e_{s-1})$$

, the match space searched by the filter using P^e will be complete up to e errors.

$$e = (s-1) + \sum_{i=0}^{s-1} e_i$$

Proof. Note that every factor i in the partition will be searched up to e_i errors. Which means that all matches containing e_i errors or less on that factor will be found. Therefore, any missing matches by filtering that factor have to contain more than e_i errors. Consequently, any missed matches by filtering all factors from the partition P^e have to contain, at least, one extra error per factor.

$$\sum_{i=0}^{s-1}(e_i + 1)$$

Equivalently, the match space found using the partition P^e contains all valid matches up to one error less.

$$\left(\sum_{i=0}^{s-1}(e_i + 1)\right) - 1 = (s-1) + \sum_{i=0}^{s-1} e_i$$

□

Depending on the choice of the error partition – and the size of the factors –, the filter will generate more or less candidates. As explained before, there are many possible choices of pattern partitions to resolve the same search, all of them equivalent. For example, searching up to 4 errors, the partitions $P_0^4 = (0, 0, 0, 0, 0)$, $P_1^4 = (1, 0, 1)$, $P_2^4 = (3, 0)$ are equivalent (graphically depicted in table 5.4).

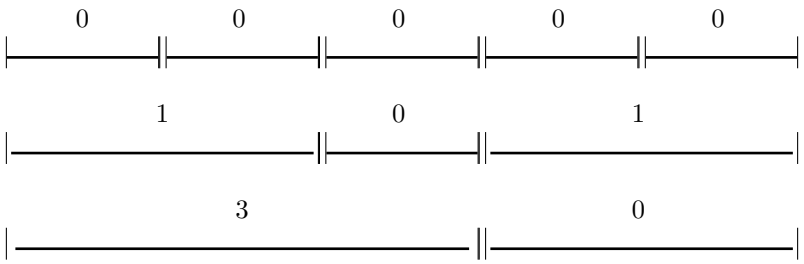


Table 5.4: Three equivalent partitions for searching up to 4 errors

However, these three partitions are expected to generate a different number of candidates and run with different complexities. For instance, P_0^4 has more factors than any other partition. Presumably, these factors will be shorter than those of P_1^4 and P_2^4. The smaller the factors, the more candidates it can potentially generate and, in the worst case, P_0^4 factors could be so small that the number of generated factors could be impractical. For that, P_1^4 and P_2^4 merge factors search each approximately. In the extreme case of P_2^4, half of the pattern will be searched up to 3 errors – which will presumably constrain the number of candidates more than any other partition. However, the performance of P_2^4 will be dominated by the time enumerating all the possibilities up to 3 errors (i.e. combinatorial search), making this error partition potentially impractical. Likewise, much experimental work on the subject [Siragusa et al., 2013; Navarro et al., 2001] concludes that the best solutions are found by balancing the number of factors and the error on each. Despite many attempts, up to this point there is no precise solution to obtaining the optimal combination.

5.3.4 Limits of static partitioning techniques

As a matter of fact, the filtering techniques presented so far represent very successful ideas towards effective approximate string matching. Nevertheless, all these techniques face the same limitations. For relatively high error rates, these setups fail to effectively reduce the search space and as result, they produce an impractical number of candidates. This can vary across different patterns and even across different seeds in the same pattern. For that reason, some seeds will produce very few candidates meanwhile others will generate impressive amounts of candidates.

In fact, this does not really depend on the filtering setup itself but on the content of the reference text. It is important to highlight the close relationship between the mappability of the reference genome at hand, and the probability that given a set of seeds, one of them will produce too many candidates (in section 3.3.3 we presented a comprehensive description of this magnitude). Here, for a given filtering approach that generates exact seeds of length l, the mappability M_l^0 gives an empirical estimation of the average number of candidates this filtering method will produce. For instance, mapping against the human genome, a factor filter up to 2 errors in a read of 108 nt will generate 3 fixed-length exact seeds of 36 nt per sequenced read. In this scenario, the mappability $M_{36}^0 = 80.12\%$ reveals that roughly 8 out of 10 seeds are expected to be unique. However, the remaining two can lead to multiple candidates and a minority (¡ 1%) will generate more than a thousand candidates. As we decrease the size of the seed, the mappability reveals that this probability of finding pathological repeats increases. Likewise, in the case of intermediate partitioning, the mappability of the error assigned to the factor can be used to measure the expected abundance of candidates.

In this way, static filtering approaches become limited by the repetitive content of the reference genome as much as the mappability score decreasing for the individual seeds. With increasing error

rates, these techniques saturate by generating an impractical number of candidates and failing to effectively filter the search space.

5.4 Approximate filters. Dynamic partitioning techniques

In general, each seed of a given pattern will produce a different number of candidates. In the worst case, a seed might belong to a repetitive region of the reference genome and produce an impractical number of candidates. As explained earlier, this is strongly linked with the content of the reference genome and the length of the seed. Hence, it seems logical to select arbitrarily large seeds so as to avoid repetitive reads.

5.4.1 Maximal Matches

This idea is not novel and it has been presented many times in the literature with different names and for different applications. Basically, a MEM (Maximum Exact Match [Gusfield, 1997]) is a substring of a pattern that matches the reference exactly (i.e. without errors) and is maximal (i.e. cannot be extended further in either direction).

Definition 5.4.1 (Maximum Exact Match)**.** Given the text T and the pattern P, then s is a MEM iff

$$(occ(s,T) > 0) \wedge (\nexists w \in P \text{ s.t. } s \in w \wedge occ(w,T) > 0)$$

Remark. In the particular case that the number of occurrences of the MEM is one, the seed is denoted **MUM (Maximal Unique Match)** [Delcher et al., 2003].

Remark. Another important subset of MEMs is the subset of **SMEMs (Super Maximal Exact Matches)** [Gusfield, 1997; Li, 2013]. That is, a MEM that is not contained in any other MEMs on the pattern. For any position of the pattern there is only one SMEM; the longest exact match covering the position.

Conceptually, extracting MUMs/MEMs/SMEMs from a given pattern is like delimiting seeds with minimum occurrences and maximal length. These types of seed encapsulate the idea of selecting seeds according to the content of the reference so as to maximise uniqueness and, in turn, minimise the number of candidates. These seeds are often found in the context of aligning very large sequences against each other (e.g. whole genome alignment). In this computationally challenging scenario, the key insight is to find significantly long exact matches that could certainly be part of the pair-wise global alignment. In other words, these unique and exact matches should almost always be aligned against the "true-match". Afterwards, the seeds are chained together and the gaps between them aligned to produce the final global alignment.

As always, the devil is in the detail and "almost" is no guarantee of accuracy. It is important to realise that the motivation behind these seeding approaches was to enable practical pair-wise alignment between sequences of the size of a genome. Looking for maximal highly conserved substrings between genomes was used to guide the alignment and reduce drastically the amount of computations needed.

However, the suitability of these approaches in the context of mapping is questionable as they do not guarantee complete searches by themselves. Note that these definitions do not prevent overlapping between seeds and therefore the pigeonhole principle no longer holds.

Nevertheless, these techniques are widely used by many popular mapping tools. This reflects the disconnection between formal approximate string matching techniques and some mapping approaches. These tools tend to favour performance – by means of drastically reducing the amount

of candidates – at the expense of the ability to preform complete searches. In practice these tools perform adequately, being very competitive. However, there is no formal guarantee that they can miss an important match in a given experiment.

5.4.2 Optimal pattern partition

At this point, the idea of selecting seeds attending to the number of candidates they produce occurs naturally. Many algorithms [Xin et al., 2013; Langmead and Salzberg, 2012] perform this selection by excluding repetitive seeds – under certain thresholds – from computations. However, in most cases this often translates into a loss of sensitivity. As opposed to discarding seeds, we could think of enlarging and shrinking non-overlapping factors to achieve a practical compromise between the number of factors, the search depth and the number of candidates generated (i.e. dynamic partitioning filters). Note that simply minimising the total number of candidates generated is not enough. Here, the key insight is to minimise the number of candidates meanwhile guaranteeing a fixed search depth. Then, for a given pattern P and a text T, the problem can be formulated in terms of finding the optimal n-factors partition.

Definition 5.4.2 (Pattern partition)**.** For a given pattern P, a pattern partition is a vector P^n of n non-overlapping substrings of P

$$P^n = (s_0, ..., s_{n-1}) \text{ s.t. } s_i \in P$$

Definition 5.4.3 (Optimal n-factors partition)**.** Given the pattern P and the text T, the optimal n-partition is a pattern partition $P^n_{opt} = (s_0, ..., s_{n-1})$ of P so that the number of candidates generated is minimum.

$$P^n_{opt} = (s_0, ..., s_{n-1}) \text{ s.t. } \min\left\{\sum_i occ(s_i, T)\right\}$$

The optimal n-partition problem can be formulated (as a recurrence by computing optimal sub-problems) and solved using a dynamic programing algorithm for $OP(n, 0)$.

$$OP(n,t) = \begin{cases} occ(P_{t..|P|-1}, T) & \text{if } n = 1 \\ \min\limits_{i=t+1}^{|P|-2}\left\{occ(P_{t..i}, T) + OP(n-1, i+1)\right\} & \text{otherwise.} \end{cases}$$

Using a precomputed table with the occurrences of every substring of the pattern P this algorithm runs in $\mathcal{O}(n|P|^2)$ worst case. Despite there having been some contributions to reduce the cost of this algorithm on the average case [Xin et al., 2016], its overall complexity makes it impractical. However, it is quite useful in establishing a lower bound on the number of candidates that a dynamic partitioning algorithm can achieve.

5.4.3 Adaptive pattern partition

For the reasons previously stated, here we present the adaptive pattern partition algorithm (APP) behind the GEM-mapper [Marco-Sola et al., 2012]; a greedy algorithm, able to approximate the n-optimal partition so that the total number of occurrences of each factor is bounded. The key insight of the method is that given a text T and a pattern P we would like to maximise the number of factors – and consequently the search depth – meanwhile minimising the total number

of candidates. Due to the inherently biased content of reference genomes, different patterns depict different content properties; some are almost unique as a whole, meanwhile others are composed of repetitive substrings. Hence, there is no fixed n that best suits the search needs of every pattern. As opposed to other partition methods, APP does not fix the total number of factors achieved in advanced. Instead, it approximates the optimum number of factors by exploring the content of the pattern (i.e. data-aware filtering). Algorithm 6 describes the APP method.

Algorithm 6: Adaptive pattern partition

```
   input  : P, T strings
   output: S set of factors approximating the |S|-optimum partition
 1 begin
 2   j ← |P| − 1
 3   i ← |P| − 2
 4   while i > 0 do
 5     occ ← occ(P_{i..j}, T)
 6     if occ ≤ Threshold then // Check occurrences
 7       p_opt ← i
 8       opt ← occ
 9       i ← i − 1
10       s ← MaxSteps
11       while s > 0 ∧ i > 0 do // Local optimisation stage
12         occ ← occ(P_{i..j}, T)
13         if occ == 0 then break
14         if occ < opt then
15           p_opt ← i
16           opt ← occ
17         end
18         s ← s − 1
19       end
20       S ← P_{i..j}  // Add factor
21       j ← p_opt − 1
22       i ← p_opt − 1
23     end
24     i ← i − 1  // Expand factor
25   end
26   if occ(P_{0..j}, T) ≤ Threshold then // Add last factor
27     S ← P_{0..j}
28   end
29   return S
30 end
```

Without loss of generality, the APP algorithm is based on the backward search of any FM-Index. Taking advantage of the FM-index backward search, the algorithm extends a factor candidate towards the beginning of the read until the number of occurrences drops below a given threshold. As a result, APP progress backwards delimiting factors.

Once a given substring $P_{i..j}$ occurs less than the threshold, a local improvement of the factor is performed. From the first eligible position, the algorithm progresses $MaxSteps$ steps in order to reduce further the number of candidates in the factor. Therefore, the algorithm selects the shortest factor $f \in P_{i-MaxSteps+1..j}, ..., P_{i..j}$ that has the least number of occurrences.

Note that given two substrings $P_{i..j}$ and $P_{i+1..j}$ occurring the same number of times in the reference, it is preferable to take the shortest so as to leave the remaining characters to consolidate other factors. In fact, extending a unique factor (i.e. one occurring once in the reference) several bases – in the fashion of a MEM – can be counterproductive. This, not only does not generate any new candidate positions, but limits the discovery of more factors – by reducing the remaining bases in the pattern – thus limiting the final search depth of the filter.

Also note that the algorithm can rarely produce zero candidate factors. Nevertheless, this is not a limitation at all, as zero candidate factors do not break the pigeonhole principle. Moreover, a zero factor indicates that at least one error is needed for the factor to match the reference.

As a result, the APP algorithm computes an approximation of the optimal partition of the pattern following the maximum number of candidates constraints. Note that by construction, the APP algorithm guarantees that all the factors generated occur less than the threshold number of times in the reference. Overall, the total number of candidates to filter is bounded and proportional to the number of factors generated. Hence, the algorithm allows the tuning of the maximum expected number of candidates generated.

In order to evaluate the results of the APP algorithm against the the optimal partition we have to run both algorithms using 5x coverage Illumina datasets against human genome. Experimental comparisons are shown in table 5.5. Notably, for the same n derived by the APP, the optimal partition reduces the number of candidates per pattern achieved by APP by just an average of 3-4 candidates. More importantly, APP correctly computes the optimal n for which the number of candidates remains practical. Compared with the optimal partition and forcing an extra factor – n+1 factors –, the total amount of candidates to verify per region increases by an order of magnitude – making any subsequent verification steps ten times slower. Also note that the optimal n varies per each read, so a fixed n for all reads could lead to even higher numbers of candidates. The goal of the APP is to properly adapt with each read and approximate the optimum partition in each case. Despite rarely achieving the optimal partition, the APP algorithm obtains an overall number of candidates very close to the optimum. At the same time, APP run in $\mathcal{O}(|P|)$ time making $\mathcal{O}(|P|)$ accesses to the index. Thus, the APP algorithm represents a practical, lightweight approach to the optimum partition (which is unfeasible in practice due to its $\mathcal{O}(n|P|^2)$ complexity).

	Candidates per read		Factors per read	
	75 nt	100 nt	75 nt	100 nt
APP(n)	12.00	10.33	3.48	4.53
OPP(n)	8.80	6.53	3.48	4.53
OPP(n+1)	147.70	59.07	4.48	5.53

Table 5.5: Experimental comparison of APP against OPP using 5x Illumina datasets at 75nt and 100nt. For this experiment APP was using ($Threshold$,$MaxSteps$) = (20,4).

It is important to realise that the fact this algorithm greedily approximates the n-optimal partition does not imply that mapping tools based on it become heuristic. On the contrary, for the same number of factors, this algorithm generates more candidates than the optimum partition does, meanwhile not limiting the match space explored. It simply misses out candidates which the optimum partition would have discarded (i.e. false positives).

The impatient reader might have noted that the APP algorithm does not guarantee a fixed search depth – it just approximates the optimum partition leading to fewer candidates. In this way, there is little space for tuning the algorithm to explore deeper strata and generate compete searches up to arbitrarily high error rates. In the following sections, complementary methods to

extend this algorithm will be proposed in a more general approximate string matching framework of chained filtering algorithms. Moreover, APP will be presented as the first stage in composed algorithms that will explore the search space progressively.

Tuning the APP algorithm

The APP algorithm allows the parametrisation of the maximum candidates allowed per factor ($Threshold$) and the maximum steps allowed to optimise a factor ($MaxSteps$). In this way, we could vary these two parameters so as to allow more candidates and further restrict each factor in length towards extracting more factors. In a more global perspective, we would like to evaluate its impact not only on the number of candidates generated, but also on the total number of mapped sequences and TPs found. Table 5.6 shows the results of this calibration benchmark. Surprisingly, variations in these two parameters do not lead to radically different results and the algorithm is moderately sensitive to this parametrisation. Variations range from the more restrictive case (1,8) leading to 1.22 candidates per read (97.05% mapped and 96.08% TP) to the more candidate tolerant case (100,2) leading to 42.18 candidates per read (99.93% mapped and 98.99% TP). In both scenarios, the overall number of candidates per read remains moderately low – with special note to (100,2) leading in most cases to just the one TP candidate – with the number of mapped reads and TP count quite high.

Threshold	MaxSteps	Factors per read	Candidates per read	FM-Index Accesses (M)	Mapped	TP
1	8	4.05	1.22	8.42	97.05%	96.08%
	6	4.05	1.22	8.42	97.05%	96.08%
	4	4.05	1.22	8.42	97.05%	96.08%
	2	4.05	1.22	8.42	97.05%	96.08%
2	8	3.77	1.53	8.60	98.24%	97.22%
	6	3.91	1.56	8.96	98.24%	97.22%
	4	4.06	1.60	9.38	98.24%	97.23%
	2	4.16	1.77	10.00	98.24%	97.23%
5	8	3.63	2.12	9.75	99.07%	98.05%
	6	3.84	2.28	10.56	99.07%	98.06%
	4	4.14	2.45	11.61	99.08%	98.07%
	2	4.29	3.19	13.61	99.08%	98.08%
10	8	3.66	2.85	11.82	99.70%	98.60%
	6	3.92	3.25	13.33	99.70%	98.61%
	4	4.25	3.50	14.70	99.70%	98.62%
	2	4.46	5.45	19.54	99.70%	98.64%
20	8	3.73	3.96	14.55	99.82%	98.74%
	6	4.10	4.92	17.57	99.82%	98.75%
	4	4.33	5.15	18.72	99.82%	98.77%
	2	4.71	10.06	30.56	99.82%	98.79%
50	8	3.87	6.93	21.57	99.90%	98.88%
	6	4.29	7.96	24.95	99.90%	98.90%
	4	4.44	10.00	29.96	99.90%	98.91%
	2	5.05	22.45	59.12	99.91%	98.93%
100	8	4.03	13.07	35.65	99.93%	98.95%
	6	4.37	12.31	34.93	99.93%	98.96%
	4	4.59	19.87	52.37	99.93%	98.98%
	2	5.19	42.18	103.50	99.93%	98.99%

Table 5.6: Mapping benchmark for different (Threshold, MaxSteps) values using Simulated Illumina-like 1M x 100nt reads (GRCh37)

5.5 Chaining filters

As presented above, many filters can be adapted to online searches – for a given reference text –, indexed searches – using a precomputed index structure –, or to filter individual candidates. Also, performance of these techniques varies from the most lightweight filters – like g-gram filters or factor filters – to the most computational intensive – like exact filters. Despite having worse performance, only exact filters guarantee full specificity. Therefore, in order to design efficient algorithms, we must aim to reduce the number of candidates passed on to an exact filter.

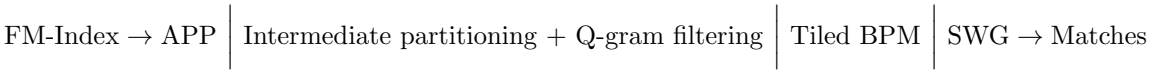
Within this unified vision of filtering algorithms, nothing prevents us from chaining several filters together so as to progressively reduce the amount of candidates to only those that actually match the pattern (true positives). Thus, every filter operates on a list of candidates or possibly an index structure and outputs a refined list of candidates. In order to completely discard false positives from the final list of candidates, an exact filter is always needed at the end of the chain.

In this section we present a general framework to progressively refine a list of candidates (horizontal chaining, section 5.5.1) and how to combine filters to perform arbitrarily deep searches without performing repeated computations (vertical chaining, section 5.5.2). Furthermore, we will reflect on how the operation of chained filtering is done and how it can be modified so as to stop the algorithms whenever satisfactory results are obtained (breadth-first vs depth-first filtering search, section 5.5.3).

5.5.1 Horizontal Chaining

Among different filtering methods one can observe differences related to their practical running times and filtration efficiencies. The idea behind horizontal chaining is to compose filters from the fastest and potentially least efficient to the most computationally intensive and fully-specific. In this context, filter efficiency will be strongly conditioned by previous filter efficiency. That is, the margin of improvement leftover as the number of false positives passed on to the next filter. Nevertheless, we should think of each intermediate filter as a fast procedure to remove false positives – compared with the running time of exact filters.

To illustrate the concept of horizontal chaining filters, here we present a simple – yet powerful – chain of filters. This filtering chain illustrates the base concepts on which the GEM-mapper filtering algorithm is based.



1. Adaptive pattern partitioning (APP)

   The first step of the chain starts with the APP algorithm. Given an index of the reference text, the APP algorithm employs the logic behind factor filters to search for exact factors of the pattern in the reference. This algorithm produces a set of candidates that will be passed on to the next stage to remove false positives.

2. Intermediate filtering + Q-gram filtering

   After the first set of candidates is generated from using APP, an online g-gram filter is applied to remove false positives. Q-gram filters perform very fast in practice. Nevertheless, they

are very sensitive to the error rate, pattern length and q-gram length. Unfortunately, the q-gram length is susceptible to being adjusted and, in turn, for longer read lengths the optimal q-gram length becomes impractical (i.e. computing simple g-grams profiles takes too much memory).

Interestingly, composing the g-gram filter with an intermediate pattern partitioning leaves room for balancing the errors among smaller chunks of the pattern. In this way, selecting an efficient choice of (l, q), subdividing the pattern in s chunks of length l – or less –, and applying the intermediate partitioning lemma, allows one to verify that at least one of the chunks matches the corresponding part of the candidate with distance $\lfloor \frac{e}{s} \rfloor$. For instance, q-gram filter with $q = 6$ and $|P| = 500$ is known from experimentation to yield quite good filtration efficiency after APP (50%). For patterns longer than 500nt, the algorithms would divide then into factors of 500nt (merging the remainder of the division with the last one). For each factor, the experimental optimum q-gram length would be used to check the g-gram lemma up to the reduced error $\lfloor \frac{e}{s} \rfloor$. As a result, the error in each factor is smaller than the overall error and the g-gram length can be selected as to maximise the filtering efficiency at each factor. Additionally, there is no dependency between q-gram filter operations at each factor so they could potentially be run in parallel.

Note that as the pattern increases in length, the number of false positives increases proportionally (i.e. with each additional factor extracted by the APP). Nonetheless, factors do not hold relationships amongst them. As a consequence, longer patterns will generate more candidates that spuriously contain a factor but greatly differ from the overall pattern. In the case of the human genome, the average factor length is 20nt. Given sequenced reads of 1000nt, candidates selected by APP are only required to match 2% contiguous bases on average. In practice, many of the candidates generated will be false positives and consequently filtered out counting q-grams.

3. Tiled BPM

   For the candidates passing the q-gram filter, a tiled BPM algorithm is applied so as to compute a lower bound of their edit distance against the pattern. Note that this is the first quadratic step in the chain towards exact filters. For patterns shorter than the tile, the filter behaves as an exact filter. After computing the BPM algorithm, all the candidates are ranked according to the resulting edit distance bounds and those over e errors are discarded.

4. SWG

   Once all candidates are ranked, a dynamic programming algorithm computing the SWG distance is used for each candidate starting from the least distant. Using the edit distance bound computed in the previous step, the SWG algorithm is banded so as to save computing large parts of the DP matrix.

One of the key ideas behind composing filters is to acknowledge their algorithmic complexity and their behaviour with increasing read lengths. Table 5.7 shows the average running time per candidate for every filter in the chain. The most important observation is that in the worst case, the q-gram filter is always 2x faster than the BPM. Moreover, as the length of the pattern increases, the q-gram filter runs comparatively much faster as its time grows linearly – as opposed to the BPM which grows quadratically. Chaining filters pays off, provided that the filtration efficiency of the individual filters is higher enough to compensate for those cases where the whole chain of filters needs to be applied. Progressive filtering of the candidates not only saves running computationally expensive algorithms like SWG on all the candidates (10x slower than q-gram filters) but also favours filtering algorithms that scale linearly with the pattern length. As shown in table 5.8,

	Q-gram $\mathcal{O}(\|P\|)$	BPM $\mathcal{O}(\|P\|^2)$	SWG $\mathcal{O}(\|P\|^2)$
75 nt	0.58	1.41	5.79
100 nt	0.66	1.96	7.08
150 nt	0.91	3.26	9.35
300 nt	1.73	8.66	16.63
500 nt	3.12	17.34	26.61
1000 nt	3.89	21.30	54.13

Table 5.7: Experimental running times in ms. for each filter in the chain using different pattern lengths (e=10%) (5x Illumina datasets for GRCh37)

the number of candidates gets reduced by the intermediate filters avoiding further computations on candidates that can be discarded using simple and cost effective filters. As the read length increases, the number of false positives removed from early stages of the filter chain increases.

	APP	Q-gram	BPM
75 nt	990.07	721.26	404.46
100 nt	808.98	593.89	308.79
150 nt	803.51	478.99	209.06
300 nt	704.79	370.44	111.62
500 nt	667.56	341.4	74.35
1000 nt	641.07	298.91	38.46

Table 5.8: Experimental number of candidates (Millions) produced by each filter in the chain (e=10%) using different pattern lengths (5x Illumina datasets for GRCh37)

5.5.2 Vertical Chaining

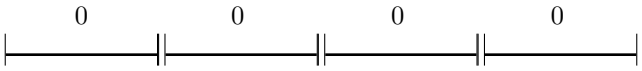
Motivation. A note on re-seeding techniques

Whenever the search depth delivered by filtering techniques is not enough, many mapping tools [Li, 2013; Langmead and Salzberg, 2012] would propose a complementary filtering stage where new seeds are generated with the aim of exploring more search space. Many re-seeding approaches have been proposed in the literature. Most of them propose generating more static seeds with a different stride, smaller in length or just in random positions in the pattern. The hope is to hit the adequate seed that leads to the "true-match". However, under the same conditions – same number of contiguous seeds and, therefore, same search depth – selecting different seed configurations iteratively with the hope of catching the expected match must be avoided. Fishing for the correct match in such a way is like looking for a black cat in a garage with the lights off and no necessarily inside. Partitioning filters must always guarantee a search depth degree. Experimental support to re-seeding techniques with no formal guarantees of solution space exploration cannot be exported beyond the limits of these experiments. Ultimately this leads to re-seeding techniques limited to precise reference organisms, read characteristics and setups.

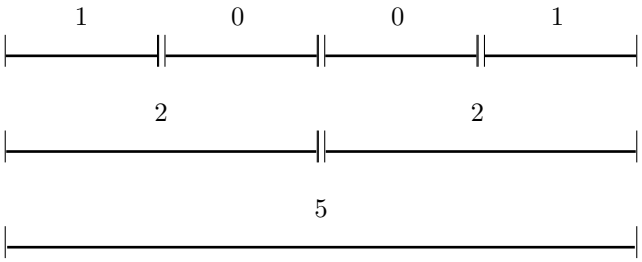
Combining candidate generation methods

As opposed to re-seeding techniques, here we propose combining filters incrementally so as to enlarge the search space. At the same time, we combine restrictions from different filters avoiding repeated computations. The key idea behind vertical chaining is to be able to combine different filters, capturing deeper and deeper strata and avoiding re-exploring strata covered by previous filters. Once a certain search depth is achieved, all the candidates generated can be passed onto any horizontal chain of filtering algorithms to report the true positive matches.

To illustrate the idea, lets assume that the APP algorithm has returned a partition $P_0^3 = (0,0,0,0)$ for a given pattern (i.e. up to 3 errors).



However, it might be the case that the search requested needs to be complete up to 5 errors. In that case, the current APP partition seems unsuitable for this purpose and other filtering approaches ought to be considered. As explained above, partitioning to accommodate 2 extra exact factors can potentially lead to an impractical number of candidates for verification. For that reason, the most effective approach would be to perform an intermediate partition. Out of many possible combinations, we depict three alternatives: $P_0^5 = (1,0,0,1)$, $P_1^5 = (2,2)$ and $P_2^5 = (5)$.

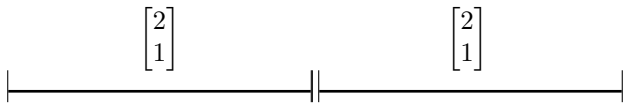


Note that the P_0^5 is bound to generate many candidates as it overloads two factors with an extra error meanwhile leaving the other two factors unused. Likewise, P_2^5 is not a filtering partition at all, as it clearly aims to generate the neighbourhood of the pattern up to 5 errors – without the need of filtering any candidate. In order to balance between factors and errors, the partition $P_1^5 = (2,2)$ seems the most suitable. However, this partition will generate candidates previously generated by the APP $P_0^3 = (0,0,0,0)$. In that case, the use of APP is pointless as its search space is already covered by $P_1^5 = (2,2)$.

However, we could think of combining the restrictions yield by P_0^3 so as to reduce the amount of computations needed for P_1^5. For that, we extend the notation of pattern partition to consider the maximum and the minimum number of errors at each factor.

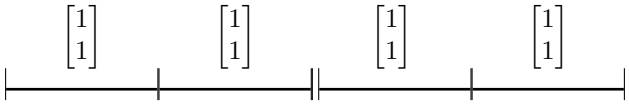
$$\begin{bmatrix}\text{Maximum errors}\\ \text{Minimum errors}\end{bmatrix}$$

Induced by P_0^3, each factor of the pattern has to contain at least 1 error so as to lead to a potential match different from all the matches considered so far. Therefore, we can restrict P_1^5 as to capture the minimum number of error required at each factor: $P_1^5 = \left(\begin{bmatrix}2\\1\end{bmatrix}, \begin{bmatrix}2\\1\end{bmatrix}\right)$



Recall that the previous scheme denotes a filtering partition of the sequence into 2 chunks. Those chunks are supposed to be searched approximately, and independently, up to 2 errors and at least 1. That means, searching for all those regions of the reference (i.e. candidates) that match any of those 2 chunks of the sequence up to 2 errors, discarding exact matches. And again, once all the candidates are found, horizontal chained techniques can be used to verify each candidate and report valid matches.

And yet, P_1^5 describes combinations that are not possible. Because at least 1 error – out of a maximum of 5 – has to be allocated in each factor of $P_0^3 = (0, 0, 0, 0)$. Thus, the maximum number of errors in each quarter of sequence can not ever reach 2. This introduces an additional restriction at the time of searching the individual factors of the partition. For that, every candidate has to match any factors (a) with at lest 1 error, (b) up to 2 errors at most, and (c) those 2 errors cannot occur in the same half of the chunk. Next figure illustrate these restrictions.



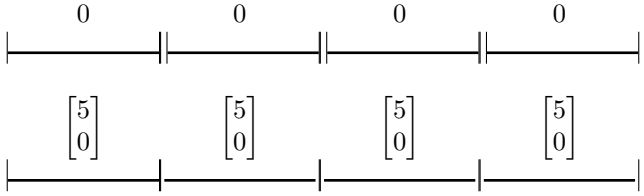
In the previous figure, note that factors are denoted using contiguous lines (i.e. 2 factors in the example), and search restrictions inside a factor are indicated using square brackets over the region of the factor to which applies.

As a result, the search of each factor has been reduced to a search up to 1 error in each half. And the overall search (allowing as much as 5 errors), to a filtering search of factors up to 1 error in each half – at most. In other words, we have reduced the intermediate filter of 2 factors up to 2 errors to the vertical chaining of 2 searches up to 1 error and 4 exact factors.

$$\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}\right) \cup \left(\begin{bmatrix}11\\11\end{bmatrix}, \begin{bmatrix}11\\11\end{bmatrix}\right)$$

This example shows that the resulting searches don't need to generate the full neighbourhood of any factor up to 2 errors. This drastically reduces the complexity of the individual factor searches and potentially leads to fewer candidates.

In order to stress this fact and illustrate the potential of the method, we could think of combining $P_0^3 = (0, 0, 0, 0)$ with the third proposed intermediate partition $P_2^5 = (5)$ (which is not really a partition, but a full neighbourhood search).



In this case, the P_2^5 constrained by P_0^3 would be reduced from searching up to 5 errors to just 2 errors. As we can see in the next scheme, imposing constraints from lightweight filtering partitions can reduce dramatically the overall complexity of very deep searches. This is true, even if they are combined on the top of a full neighbourhood search – based on combinatorial exploration of all possibilities. Nonetheless, the combination of all searches leads to the same results as directly searching for just one intermediate partition.

As with the first formulation of intermediate partitioning techniques, many possibilities arise from this method to reduce the search space. To the best of our knowledge, there is no method to determine the best combination of partitions that guarantee the least number of combinations explored. Nevertheless, vertical chaining introduces a key idea towards deep approximate string searches. The key contribution is to prove that filtering searches can be combined on the top of a combinatorial neighbourhood search to prune the search tree. Furthermore, note that in the previous scheme, all chunks of the sequence are forced to contain at least one error, but not all of them can have the maximum of 2 – it would override the global limit of 5 errors at most. These ideas lead to faster search strategies in cases where the error is very high and filtering techniques generate an unfeasible number of candidates.

5.5.3 Breadth-first vs depth-first filtering search

Up to this point, we've presented several filtering techniques towards efficient and arbitrarily deep searches. In this way, we have presented how horizontal chaining comes to prove that it's preferable to progressively filter candidates using more restrictive and computationally expensive filters each time. But also, how vertical chaining explores progressively the search space using incrementally deeper partitions. In this last section of the chapter, we want to emphasise further more on the relevance of progressive algorithms that can explore incrementally the search space allowing effective pruning techniques.

Classical literature on approximate string matching using filtering techniques presents the problem as a two-step process; generation and verification of candidates (one following the other). This can be understood as a breadth-first search where all the generated candidates are brought together and only once have all been gathered – from searching the individual seeds – can the verification begin. In most cases, the arrangement details between these two steps is neglected. As a result, many actual algorithms waste computational resources searching and gathering more candidates than strictly necessary to resolve the mapping search.

This two-step approach can be reformulated as a depth-first search by iterating the procedures of generating the candidates from a given seed and verifying them. Using partitioning methods like APP we can guarantee an incremental complete search depth each time the candidates of a seed are verified. By incrementally exploring strata we regulate the number of candidates explored at each iteration. In the context of determining the δ-strata group of a given sequenced read, we would iterate the process of generation and verification of candidates until a certain search depth is reached. In practice, most cases reach a match tie before exploring up to the maximum feasible strata e. Hence, depth-first search can implement effective cut-off techniques as to prevent the search from exploring unnecessary strata. That is, exploring matches that would not change the δ-strata group membership of a given read.

Chapter 6

Approximate string matching III. Neighbourhood search

Filtering techniques are widely known to behave very well in practice with relatively long patterns and reduced error rates. More precisely, filtering approaches perform well as long as the number of candidates generated is relatively low. However, there many other scenarios where filtering is just not good enough. In these cases, the resulting number of candidates to verify is too large. Hence, filtering algorithms become saturated and take large amounts of computational resources to verify all potential matching positions.

In contrast, neighbourhood search techniques explore all possible combinations by enumerating all sequences that can match the pattern within a given distance. In the case of indexed searches, these combinations are searched against the index, and all those occurring in the index are reported as matches. At first glance, neighbourhood techniques could seen as crude combinatorial algorithms, deemed to behave poorly in practice. Surprisingly however, they depict certain properties that cannot be mimicked by filtering techniques. Thus, in certain cases, neighbourhood techniques do outperform filtering approaches in terms of computational efficiency. Note that in practice, the more efficient an algorithm is, the deeper the searches it can perform within a reasonable time. Thus, neighbourhood algorithms can perform some searches up to a very high error rate in a short period of time, whereas no filtering approach could ever explore the same search space in a reasonable time.

In this chapter, we present the third block of approximate string matching focused on neighbourhood search techniques. We will introduce the basics of neighbourhood searches from the most basic combinatorial algorithms to the more elaborate versions towards avoiding unnecessary computations. From here, we introduce bidirectional search techniques to precondition the search and reduce the number of combinations explored. Furthermore, we introduce novel pruning techniques to further enhance the search. Finally, we introduce hybrid techniques that use filtering partitions to precondition neighbourhood searches and efficiently prune the search tree. This allows us to combine filtering and neighbourhood algorithms progressively depending on the requirement of each individual search. As a result, the final hybrid algorithms enable adaptive searches using different algorithmic approaches depending on the sequence at hand and its complexity.

6.1 Neighbourhood search

Given a distance function $d(x, y)$, neighbourhood of a string s at distance e as $N_d^e(s) = n_0, .., n_n \mid d(n_i, s) \leq e, \forall n_i$ (definition 3.2.20). In other words, the e-neighbourhood of a string s is composed by all the strings produced from applying e error operations to s in all possible ways. For example, under the Hamming distance, the 1-neighbourhood of $w =$ "$ACGT$" is $N_h^1(ACGT)$.

$$\begin{aligned} N_h^1(ACGT) = \{&ACGA, ACGC, ACGG, ACAT, \\ &ACCT, AAGT, ACGT, CCGT, \\ &GCGT, TCGT, AGGT, ATGT, ACTT\} \end{aligned} \tag{6.1}$$

Similarly, under the Levenshtein distance, the 1-neighbourhood of $w =$ "$ACGT$" is $N_{lev}^1(ACGT)$.

$$\begin{aligned} N_{lev}^1(ACGT) = \{&ACGA, ACGTA, ACGC, ACGTC, \\ &ACG, ACGG, ACGTG, ACAT, \\ &ACGAT, ACT, ACCT, ACGCT, \\ &AAGT, ACAGT, AGT, AACGT, \\ &CACGT, GACGT, TACGT, ACGT, \\ &ACCGT, CCGT, AGCGT, GCGT, \\ &ATCGT, TCGT, CGT, AGGT, \\ &ACGGT, ATGT, ACTGT, ACTT, ACGTT\} \end{aligned} \tag{6.2}$$

In the context of indexed searches, a neighbourhood search algorithm would generate all these n_i strings, and check them against the index to produce the final set of matches. Intuitively, any algorithm based on generating all these possible combinations of strings at a given distance from a pattern, quickly becomes intractable – as the number of combinations grow exponentially (we will later formalise this idea). Albeit that it might appear an intractable algorithm at first, it depicts some quite advantageous properties when searching for very repetitive sequences – or sequences containing very repetitive regions, in comparison to filtering methods.

Neighbourhood searches implicitly build a search tree which mimics the traversal of a trie. By exhaustively searching all combinations, they traverse all the paths of the tree, reporting complete sets of results. Besides this completeness, note that each node of a neighbourhood search contains all positions of the reference matching the searched path, conveniently compacted on a SA-interval. Hence, no matter how many times a specific pattern – or a neighbourhood string of it – occurs in the reference, it will be encoded as a single interval. In cases where the sequence read hits a highly duplicated region of the genome, filtering methods will have to check individually each of these matches. Whereas, a neighbourhood search could just look up the repetitive pattern and return a single interval encoding all occurrences of the duplicated sequence.

In addition, neighbourhood searches can be advantageous when the sequence depicts high divergence from the reference. Recalling filtering methods, these algorithms require an anchor seed with a low error rate – or no error at all – to spot the true locus of the sequence. Whenever the input sequence depicts a high error rate (i.e. sequencing errors), contains large variants, or the reference is of poor quality, filtering algorithms find it difficult to find such anchor seeds. Even using approximate seeds – one of the most sensitive forms of seed –, these methods would potentially spend too much time verifying all plausible candidates until the true locus is found. As opposed to filtering, neighbourhood searches can explore all the error/variant combinations without having to verify each candidate they find along the way. For that reason, neighbourhood searches are tailored to search for those matches virtually unreachable by filtering algorithms.

Additionally, in cases where searching beyond the first matching result is required (e.g. to accurately determine the δ-strata group of a sequence, CNV studies, etc), filtering methods become profoundly limited. In fact, they can only increase the search depth by increasing the number – or the error rate – of the filtering factors. This quickly leads to unfeasible amounts of candidates – ultimately forcing verification with large parts of the reference. Yet, neighbourhood searches can deal with higher error rates by exploring more nodes of the search tree. Although this may seem to lead to a combinatorial explosion of cases, we must bear in mind that not all possible string combinations occur in the reference. Indeed, neighbourhood search trees are heavily pruned by the substrings contained in the index (i.e. search paths that actually appear in the reference). Hence, neighbourhood search trees can refine the search space until only the actual matches are found, thereby avoiding the inspection of all the reference as filtering methods would do when saturated.

Another interesting case, where neighbourhood searches can potentially outperform filtering methods, occurs when the input sequence contains hazardous uncalled bases (i.e. Ns). In this case, uncalled bases can prevent filtering algorithms from extracting factors, thus limiting their length, and ultimately generating short factors with too many potential candidates to verify. Conversely, neighbourhood searches can search through uncalled bases by trial-and-error. This allows the gap of uncalled bases to be filled transparently during the search, without incurring any extra cost. What is more, uncalled bases force the allocation of errors during the search, thus reducing the overall error degree left to consider in the rest of the sequence.

On the whole, neighbourhood searches can be very powerful whenever the error rate is high, when the sequence – or any of its substrings – is very repetitive, when searching beyond the first matching strata, or in the case of sequences containing obtrusive uncalled bases. Nevertheless, it is vital to understand that under normal circumstances these algorithms are generally more expensive than filtering techniques. For this reason, these algorithms should not be used standalone. It is important to detect those cases unsuitable for filtering, and complement them using neighbourhood searches.

Size of the neighbourhood

As stated above, the main drawback to neighbourhood searches is the combinatorial explosion of cases to consider. In order to illustrate in detail the exponential nature of the approach, we can compute the size of the neighbourhood of a pattern.

Lemma 6.1.1 (Size of the neighbourhood of a string (under Hamming distance)). Given the alphabet Σ, under the Hamming distance function, the size of the neighbourhood of a string s at distance e is given by the following expression.

$$N_h^e(w) = 1 + (|\Sigma| - 1) \sum_{i=1}^{e} \binom{|s|}{i}$$

Proof. We can think of the number of strings with exactly e mismatches, as the number of combinations to select e positions from the string as to substitute the character with another $|\Sigma| - 1$ from the alphabet $((|\Sigma| - 1)\binom{|s|}{e})$. In that way, we compute the total number of strings in the neighbourhood $N_h^e(w)$ by adding up all the strings that have 0 mismatches (i.e. exact) to e mismatches. □

In the case of the Levenshtein distance, there is no known close expression of size of a neighbourhood. However, there are known bounds – and fairly tight recurrence formulas [Myers, 2013] – that reinforce the idea that the neighbourhood of a pattern quickly grows with increasing error rates.

Lemma 6.1.2 (Size of the neighbourhood of a string (under Levenshtein distance)). Given the alphabet Σ, under the Levenshtein distance function, the size of the neighbourhood of a string s at distance e is bounded by the following expression.

$$\overline{N}_h^e(w) = \binom{|s|}{e}(2|\Sigma|)^e$$

Proof. An extension to the idea shown at lemma 6.1.1, extended to consider not only mismatches but insertions and deletions of any character and length up to e errors, can be found at [Myers, 2013]. In [Covington, 2004; Myers, 2013], the reader can also find detailed methods to further bound $N_h^e(w)$. □

Brute-force neighbourhood search

The first and most straightforward algorithm to produce the neighbourhood of a string is based on using brute force to generate all string combinations. As the search progress, the alignment error of the search path must be checked against the pattern. Algorithm 7 shows how to generate the Hamming neighbourhood of a string. The algorithm keeps track and regulates the number of mismatches introduced. Note that each search path implicitly describes a traversal path over the trie of the reference, and makes up a string occurring in it. Once a tree leaf (i.e. within Hamming distance, a search path of the length of the pattern) is reached, the resulting string is added to the neighbourhood set. Note that the search is done backwards – in the spirit of the FM-index. Additionally, search paths leading to empty SA-intervals (i.e. searches of substrings not contained in the reference) force the search path to be abandoned.

Algorithm 7: Hamming Neighbourhood Index-Search

```
1  NS-Hamming(p, e_p, SA_p)
   input : I FM-Index, P string, e maximum error
           p current position, e_p current error, SA_p current SA-interval
   output: N Hamming neighbourhood of P at distance e (i.e. N_h^e(P))
2  begin
3      foreach c ∈ Σ do
4          e_{p+1} ← e_p + (c ≠ P[|P| − 1 − p]?1 : 0)  // Compute error
5          if e_{p+1} ≤ e then // Check error
6              SA_{p+1} ← FM-Index-query(I, SA_p, c)  // Query index
7              if SA_{p+1} = ∅ then continue
8              if p < |P| − 1 then
9                  NS-Hamming(p + 1, e_{p+1}, SA_{p+1})  // Next step
10             else
11                 N ← N ∪ SA_{p+1}  // Add result SA-interval
12             end
13         end
14     end
15 end
16 // Initial call
17 N ← ∅
18 NS-Hamming(0, 0, SA_T)  // SA_T = [0, |T|) full text SA interval
```

6.2 Dynamic-programing guided neighbourhood search

Despite the simplicity of algorithm 7, neighbourhood searches using other distances can become tricky. Unsurprisingly, there are still a number of recent publications [Salavert et al., 2015; Tennakoon et al., 2012] that struggle to capture all the combinations of mismatches, insertions and deletions trying to avoid repeated cases (e.g. insertion followed by deletion).

However, there is a much simpler solution to the problem, first glimpsed many years ago [Myers, 1994]. Instead of focusing on tracking the edit operations, it is easier to track the edit distance of every tree path as the search progresses. By incrementally adding characters to each path, we can compute the edit distance of the path and limit those branches that override the maximum error allowed. For that reason, we could potentially benefit from any exact filter (section 5.2) explained before.

Without loss of generality, we base both our explanations and implementations – therefore benchmarks – on the dynamic programming solution. In doing so, we take advantage of the most flexible method known to date for implementing several cut-off techniques. Although it is presumably one of the most expensive exact filters, its simplicity enables us to derive comprehensible bounds and avoid further complexities at implementation. At the same time, this enables the algorithm to benefit from all proposed exact filter enhancements (e.g. banded computations, BPM processing, SIMD processing, etc). Furthermore, this approach is suitable for easy adaptation to many other distance metrics.

Before going into further detail, there are some basic concepts that need to be addressed to fully understand the methods and optimisations presented here. The following sections are presented incrementally by adding refinements and optimisations, with the sole purpose of pruning the search tree to avoid unnecessary computations and enhance practical neighbourhood searches.

6.2.1 Full, condensed, and super-condensed neighbourhood

In spite of its exponential nature, the full set of neighbourhood strings of a pattern contains a certain degree of repetitiveness. In fact, many neighbourhood strings have as prefix, suffix or substrings other strings also contained in the neighbourhood. To find all approximate matches in a reference, it suffices to look only for those strings of the neighbourhood that are not the extension of other neighbourhood strings. Otherwise, we would just report the same locus of the reference using different search paths. Recalling the neighbourhood set of the string "ACGT" up to one error (equation 6.2), we call this the full neighbourhood. Below (equation 6.3), we can see marked in bold all those substrings that are also members of the neighbourhood.

$$
\begin{aligned}
N^1_{lev}(ACGT) = \{&\mathbf{ACG}A, \mathbf{ACGT}A, \mathbf{ACG}C, \mathbf{ACGT}C,\\
&ACG, \mathbf{ACG}G, \mathbf{ACGT}G, ACAT,\\
&\mathbf{ACGA}T, ACT, ACCT, \mathbf{ACGC}T,\\
&A\mathbf{AGT}, AC\mathbf{AGT}, AGT, A\mathbf{ACGT},\\
&C\mathbf{ACGT}, G\mathbf{ACGT}, T\mathbf{ACGT}, \mathbf{ACGT},\\
&AC\mathbf{CGT}, C\mathbf{CGT}, AG\mathbf{CGT}, G\mathbf{CGT},\\
&AT\mathbf{CGT}, T\mathbf{CGT}, CGT, AGGT,\\
&\mathbf{ACGG}T, ATGT, \mathbf{ACT}GT, \mathbf{ACT}T, \mathbf{ACGT}T\}
\end{aligned}
\tag{6.3}
$$

In this way, we remove all neighbourhood strings that have a suffix in the same neighbourhood to obtain the condensed neighbourhood (equation 6.4).

$$
\begin{aligned}
CN^1_{lev}(ACGT) = \{&ACG, ACAT, ACT, ACCT,\\
&AAGT, ACAGT, AGT, AACGT,\\
&CACGT, GACGT, TACGT, ACCGT,\\
&CCGT, AGCGT, GCGT, ATCGT,\\
&TCGT, CGT, AGGT, ATGT\}
\end{aligned}
\tag{6.4}
$$

Furthermore, we remove from the condensed neighbourhood all those strings that have a prefix in the same neighbourhood to obtain the supercondensed neighbourhood (equation 6.5).

$$
\begin{aligned}
SCN^1_{lev}(ACGT) = \{&ACG, ACAT, ACT, ACCT,\\
&AGT, CGT, AGGT, ATGT\}
\end{aligned}
\tag{6.5}
$$

As we can see, from the initial 33 strings in the full neighbourhood, the condensed neighbourhood reduces it to 20 strings, and the supercondensed neighbourhood to just 8. At the time, all the neighbourhood sets lead to the same solutions in a given reference, as all solutions are just extensions from the supercondensed neighbourhood strings. Find below the formal definition of these reduced sets.

Definition 6.2.1 (Condensed Neighbourhood)**.** Given a distance function $d(x, y)$, we define the condensed neighbourhood of a string s at distance e as

$$N^e_d(s) = \{n_0, .., n_n \mid (d(n_i, s) \leq e, \forall n_i) \wedge (\nexists n_j \text{ suffix of } n_i,\ i \neq j)\}$$

Definition 6.2.2 (Supercondensed Neighbourhood)**.** Given a distance function $d(x, y)$, we define the supercondensed neighbourhood of a string s at distance e as

$$N^e_d(s) = \{n_0, .., n_n \mid (d(n_i, s) \leq e, \forall n_i) \wedge (\nexists n_j \text{ substring of } n_i,\ i \neq j)\}$$

6.2.2 Full neighbourhood search

With these definitions in mind, we now proceed to describe the base algorithm to search the full neighbourhood of a pattern. NS-Levenshtein (algorithm 8) is a procedure that recursively expands each search node appending all possible characters in the alphabet. This procedure continues while the search SA-interval is not empty, and the search-patch allows for solutions with edit distances e at most. Note that line 12. regulates the inclusion of the current solution (i.e. SA-interval described by the search path) based on its alignment distance. Nonetheless, the search does not have to stop at that point, and line 15. allows the continuation of the search, provided that there are more feasible solutions within reach. This depends on the values of the current DP-column cells and whether these cells are active or not. *Active cells* are those that can lead to a valid alignment (i.e. e errors at most), assuming all matches from their diagonal – best case scenario. In this way, if the current DP-column contains any active cell, NS-Levenshtein will be called recursively to search for further solutions. Overall, the search mimics a trie traversal and each step expands a given search node by adding all characters in the alphabet. By induction, the search will explore all possible string combinations that, regulated by the DP-table, are contained in the full neighbourhood.

In order to keep track of the alignment distance of the search path – and prune paths –, DP-column(P, C_p, c) (algorithm 9) is used to update the dynamic programming table. This procedure computes the next column in the DP-table, using the previous one C_p and the last character used in the search path c. Note that $C[-1] = p$ so as to forbid for free-end alignments and force any search path to trace-back to position $(0, 0)$ of the DP-table.

Algorithm 8: Dynamic-Programing Neighbourhood Search

```
1  NS-Levenshtein(p, e_p, SA_p, C_p)
   input : I FM-Index, P string, e maximum error
           p current position, e_p current error,
           SA_p current SA-interval, C_p current DP-Column
   output: N Levenshtein neighbourhood of P at distance e (i.e. N^e_lev(P))
2  begin
3      foreach c ∈ Σ do
4          // Compute DP column
5          C_{p+1} ← DP-column(P, C_i, c)
6          if min{C_{p+1}} > e then continue
7          // Query FM-Index
8          SA_{p+1} ← FM-Index-query(I, SA_p, c)
9          if SA_{p+1} = ∅ then continue
10         // Keep searching (Full Neighbourhood)
11         e_{p+1} = C_{p+1}[|P| − 1]
12         if e_{p+1} ≤ e then
13             N ← N ∪ SA_{p+1}  // Add result SA-interval
14         end
15         if min{C_{p+1}} ≤ e then
16             NS-Levenshtein(p + 1, e_{p+1}, SA_{p+1}, C_{p+1})
17         end
18     end
19 end
20 // Initial call
21 N ← ∅
22 C_0[−1..|P| − 1] ← 0..|P|
23 NS-Levenshtein(0, 0, SA_T, C_0)  // SA_T = [0, |T|) full text SA interval
```

Algorithm 9: Dynamic Programing Compute Column

```
1  DP-column(P, C, c)
   input : P pattern, C_p DP-Column, c character
   output: R next DP column adding c
2  begin
3      C[−1] ← p
4      for i in 0..|P| − 1 do
5          del ← C[i − 1] + 1
6          ins ← C[i] + 1
7          match ← C[i − 1] + (c = P[i])?0 : 1
8          R[i] ← min{del, ins, match}
9      end
10     return R
11 end
```

Note that queries to the FM-Index are done assuming forward searches for the sake of simplicity. Although it is straightforward using a bidirectional index, it can easily be adapted to the backward search by simply reversing the input pattern.

6.2.3 Supercondensed neighbourhood search

However, as shown earlier, searching for the full neighbourhood produces far more search paths than strictly necessary. Our goal is to restrict the search to the supercondensed neighbourhood. For that, we need to modify the algorithm that guides the search (i.e. dynamic programing table), to discard search paths describing strings that contain a substring member of the same supercondensed neighbourhood. For that, we must forbid free-end alignments that can traceback to any other cell rather than $(0, 0)$. We denote those traceback paths as *free-end paths*. Similarly, we denote all cells of the DP-table that have a free-end patch – lead to a free-end alignment – as *free-end cells*.

$$\begin{bmatrix} & - & A & T & G & A & T \\ - & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ G & 1 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ A & 2 & 1 & \mathbf{2} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ T & 3 & 2 & 1 & \mathbf{2} & \mathbf{1} & \mathbf{0} \\ T & 4 & 3 & 2 & 2 & \mathbf{2} & \mathbf{1} \\ A & 5 & 4 & 3 & 3 & 2 & \mathbf{2} \\ C & 6 & 5 & 4 & 4 & 3 & \mathbf{3} \\ A & 7 & 6 & 5 & 5 & 4 & \mathbf{4} \end{bmatrix}$$

Figure 6.1: Dynamic programing table for P="GATTACA" and T="ATCAT" (δ_e("$GAGATA$","$GATTACA$")). Free-end cells are marked in bold and red.

Figure 6.1 shows an example of a DP-table with all free-end cells marked. All these cells can lead to free-end alignments that do not require any starting characters from the text. For example, note that the value zero on the last column simply denotes that the suffix "GAT" (i.e. the beginning of the pattern) can lead to an exact free-end match by adding the remaining "TACA". But in such a case, another search is bound to start with "GAT" at the beginning. Hence, there is no need for the first "AT" characters and any path from this cell should be discarded. Overall, in this example, as all cells from the last column of figure 6.1 are free-end cells, there is no need to further continue the neighbourhood search from this node. All solutions from this point will be explored from shallower nodes of the search tree.

In order to implement this free-cell concept, we modify algorithm 9 to keep track of free-end paths. The resulting procedure, DP-supercondensed-column (algorithm 10), embeds a flag into each cell to keep track of whether the minimum between all edit operations leads to a free-end cell or no. Additionally, initial conditions of any new column allow for free-end alignment (i.e. line 3. $C[-1].error \leftarrow 0$), properly marked as free-end cells.

Below, algorithm 11 describes the procedure NS-Supercondensed (based on algorithm 8), that finds all strings in the supercondensed neighbourhood of a pattern. Note that line 6. discards all cases where the last column of the DP-table does not contain any active cells, or all active cells are free-end cells. However, DP-supercondensed-column can only detect cases where the search path describes a string with a suffix contained in the neighbourhood. We also need to prevent the search from extending beyond search paths that already align to the pattern (line 12. and 14.). Then, if the search path aligns to the pattern, the string described is added to the results (line 13.) and the search stops. Otherwise, as in line 6. after checking that there is at least one active cell, the search continues recursively calling NS-Supercondensed. As a result, valid search paths are not extended and the search does not produce any search path that contains as a prefix another string in the neighbourhood.

As we can see, algorithm 11 will emulate a trie traversal over the neighbourhood of the pattern,

Algorithm 10: Dynamic Programing Compute Column (Supercondensed)

```
1  DP-supercondensed-column(P, C, c)
   input  : P pattern, C_p DP-Column and free-end flags, c character
   output: R next DP column adding c
2  begin
3      C[−1].error ← (0, true)
4      for i in 0..|P| − 1 do
5          del ← (C[i − 1].error + 1, C[i − 1].freeEnd)
6          ins ← (C[i].error + 1, C[i].freeEnd)
7          match ← (C[i − 1].error + (c = P[i])?0 : 1, C[i − 1].freeEnd)
8          R[i] ← min_error{del, ins, match}
9      end
10     return R
11 end
```

Algorithm 11: Supercondensed Neighbourhood Search

```
1  NS-Supercondensed(p, e_p, SA_p, C_p)
   input  : I FM-Index, P string, e maximum error
            p current position, e_p current error,
            SA_p current SA-interval, C_p current DP-Column
   output: N Supercondensed neighbourhood of P at distance e
2  begin
3      foreach c ∈ Σ do
4          // Compute DP column (Supercondensed)
5          C_{p+1} ← DP-supercondensed-column(P, C_i, c)
6          if (c_i.error > e ∨ c_i.freeEnd, ∀c_i ∈ C_{p+1}) then continue
7          // Query FM-Index
8          SA_{p+1} ← FM-Index-query(I, SA_p, c)
9          if SA_{p+1} = ∅ then continue
10         // Keep searching (Supercondensed Neighbourhood)
11         e_{p+1} = C_{p+1}[|P| − 1].error
12         if (e_{p+1} ≤ e) ∧ (¬C_{p+1}[|P| − 1].freeEnd) then
13             N ← N ∪ SA_{p+1}  // Add result SA-interval
14         else
15             NS-Supercondensed(p + 1, e_{p+1}, SA_{p+1}, C_{p+1})
16         end
17     end
18 end
19 // Initial call
20 N ← ∅
21 C_0[−1] ← (0, false)
22 C_0[0..|P| − 1] ← (1..|P|, true)
23 SA_T ← [0, |T|)
24 NS-Supercondensed(0, 0, SA_T, C_0)
```

avoiding strings that have a prefix or a suffix in the same neighbourhood. Thus, producing the supercondensed neighbourhood of the pattern.

6.3 Bidirectional preconditioned searches

So far, we have presented the basics of any neighbourhood search and some refinements to generate a succinct representation of the set (i.e. supercondensed neighbourhood) for the purpose of ASM. Nevertheless, theoretical bounds still support the neighbourhood set of string growing exponentially. Moreover, the number of possible paths to traverse, in order to perform searches with high error rates or long read lengths, quickly becomes intractable. Actually, the only fact that prevents this worst-case scenario from happening is that not all possible string combinations occur in a limited length reference. That is, SA-Intervals are bound to be reduced as the search progresses and the search space reduces. Ultimately, deep enough search paths are expected to lead to a handful of matches in the reference as they become highly specific.

Combinatorial explosion and decay of search paths

In order to strengthen this idea, we formally present a concept that will help us bound this worst case scenario. In short, for a given alphabet Σ, a *deBruijn* sequence $B(n, \Sigma)$ is a cyclic sequence in which every possible string of length n occurs exactly once [Higgins, 2012]. That means, that a deBruijn sequence is the minimum length string that contains all possible substrings of length n. In thi case, a deBruijn sequence used as a reference represents the minimum length text for which the neighbourhood search falls into in its worst case scenario. A search against such a reference would require exploring all feasible paths up to n steps, all of them being contained in the reference. However, for the step $n+1$, we would expect the search to abandon many paths as they cannot occur in the reference (by construction of the deBruijn sequence). At the same time, a deBruijn sequence $B(n, \Sigma)$ is known to have length $|\Sigma|^n$ [Higgins, 2012]. Therefore, given any reference text T and assuming the worst case scenario, in which this reference is also a deBruijn sequence, we can expect the search to start closing paths past $\log_{|\Sigma|}(|T|)$ steps.

Interestingly, the same result can be obtained by means of the proper length p_l (definition 3.3.8). The definition of proper length implies that for search paths longer than p_l, the probability of having a non-empty SA-interval is almost zero. Thus, for patterns longer than p_l, neighbourhood searches beyond p_l steps are theoretically bounded by the content of the reference and cannot expand to all possible combinatorial paths.

However, by the same principles that prevent the combinatorial explosions beyond a certain search depth, the first steps in any of the presented neighbourhood searches are bound to explore all combinations. Broadly speaking, for a relatively small k, the probability that a given reference contains almost every possible k-mer is very high. Hence, searches in their first steps will grow exponentially leading to a strong penalty because proportionally only a few paths will lead to valid matches – as shown earlier.

Distributing errors

Nevertheless, this early explosion of cases is the direct result of allowing an unrestricted number of errors (up to e) since the beginning of the search. A more thoughtful approach would divide the search into cases by distributing errors in the spirit of the pigeon hole. Recalling figure 5.7 in chapter 5, we could divide a search into several cases. All those cases have a region which has to match exactly. Within that region, a neighbourhood search is restricted to follow a single search path avoiding any node expansion. If the region is large enough, the search space is expected to be significantly reduced. Thus, extending the search from that point towards the two remaining regions, is also expected to end up in a measured number of search nodes – conditioned by the properties explained previously.

This is the essence of preconditioned neighbourhood searches. These methods exploit the fact that not all possible combinations occur in the reference, arranging searches in such a way that

combinations not contained in the reference are quickly discarded in favour of those search paths that lead to valid matches. The previous strategy implies that the search can be performed in both directions; forward and backwards – hence, the name of the method. For that reason, a bidirectional index is required (e.g. bidirectional FM-Index) or the use of two indexes (e.g. the SA of the forward and reverse reference text). In this section, bidirectional neighbourhood searches (BNS) are presented as an effective method to precondition searches and mitigate combinatorial explosions throughout neighbourhood searches. Here we present the basics of the method and, taking advantage of the flexibility of the method, we will show novel bounds that will further constrain the cost of neighbourhood searches.

6.3.1 Bidirectional search partitioning. Upper bounds

To illustrate the idea, we will use a search up to four errors as an example. As explained above, instead of doing a neighbourhood search up to four errors from the beginning, we divide the problem into two searches. Then, we divide the pattern into two **search regions** and, for each search operation, we only allow for two errors on the first region searched. In this way, we aim to control the number of paths explored in the first half of each neighbourhood search. In more detail, figure 6.2 shows an scheme of both searches. Note between parenthesis the global limits on the error imposed to each search region. In this way, the first search S_{00} starts from the left, searching forward for the first half of the pattern. During this region's search, the neighbourhood search has to follow the error limits specified between parenthesis, and consider as valid only those paths whose distance goes from zero to two errors. Once a search path jumps to search the other half of the pattern (i.e. S_{01}), the algorithm is allowed to introduce two more errors – so that the search paths can align using from zero to four errors (note that a search path can proceed to simply add four errors in the second half). Similarly, the second search S_{10} and S_{11} does the symmetric search starting from the right (i.e. backwards). By simple enumeration of cases, it is straightforward to see that these two searches cover all possible distributions of four errors in two regions. Thus, the results retrieved by these two searches are equivalent to a single unrestricted search up to four errors. The overall effect of distributing the errors is to regulate the number of nodes explored in each search. Moreover, as the search increases in depth, SA-intervals are expected to shrink, limiting the search space and preventing an explosion of cases to explore when the maximum error is allowed.

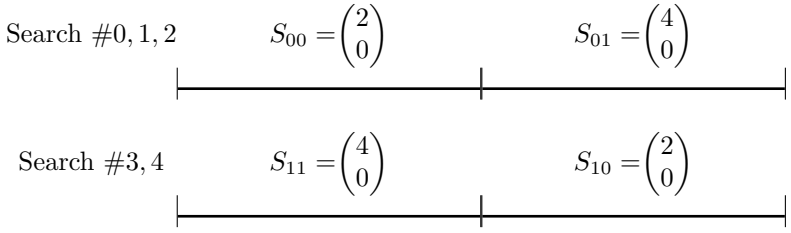


Figure 6.2: Generation of the BNS-partition. First step splitting four errors into two search regions. Global error bound of each region are specified between parenthesis (i.e. $\binom{max}{min}$).

Now, we proceed to apply recursively the same idea to the first searched region of each search partition. We repeat until the starting search region is reduced to an exact search – or a region

shorter than the number of errors assigned. Figure 6.3 shows this recursive process until search #0 is derived (out of a total of five searches). Note that search #0 is annotated with the direction and order in which each region search has to be performed. Moreover, each region is bounded by the maximum and minimum errors allowed – global error of the search path – when performing that region's search. As a result, the first region search performs an exact search that can only produce one search node. As the search progresses, the error allowed is doubled looking for the pattern length specificity to limit the number of paths explored.

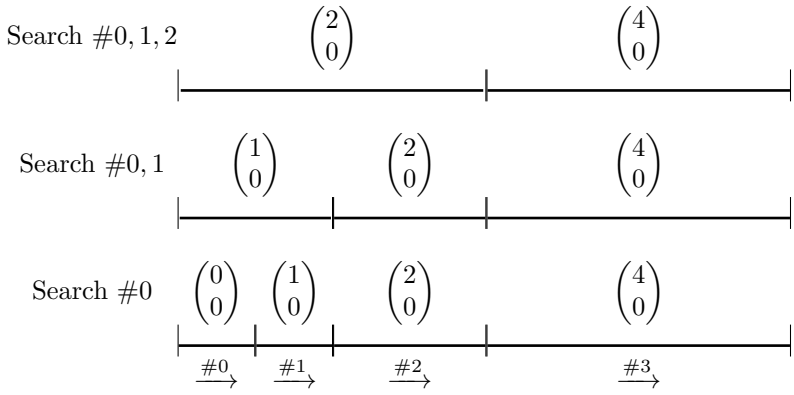


Figure 6.3: Generation of a single BNS-partition (e=4). The resulting search partition (no. 0) is annotated with the order in which the regions must be searched and the direction of these searches

In the same manner, figure 6.4 shows the full set of BNS-partitions searching up to four errors. Against all intuition, searches are allowed to start from the middle of the pattern and progress in both directions exploring the neighbourhood.

Lemma 6.3.1 (Equivalence and completeness of search partitions). Given a search S and e maximum error degree allowed, the set of partitions produced by the bidirectional search is equivalent to the search S and returns all possible matches up to e errors.

Proof. According to the pigeon principle, we hold that distributing e errors into two search partitions – (S_{00}, S_{10}) and (S_{11}, S_{10}), as shown before – is equivalent to a single search S up to e errors. In all cases, one of the two regions has to have $\lceil e/2 \rceil$ errors at most. As the second half of the search allows up to e errors, both search partitions cover all possible arrangements of e errors into two regions. Furthermore, each SS_{i0} can be divided recursively and, by induction we can prove that the resulting search partitions are equivalent to a single search of S_{i0} up to $\lceil e/2 \rceil$ errors. In the base case, an exact search of a region is performed reporting all occurrences of the region. In this way, the method is complete – does not miss any matches – and partitions in figure 6.4 are equivalent to a neighbourhood search up to four errors. □

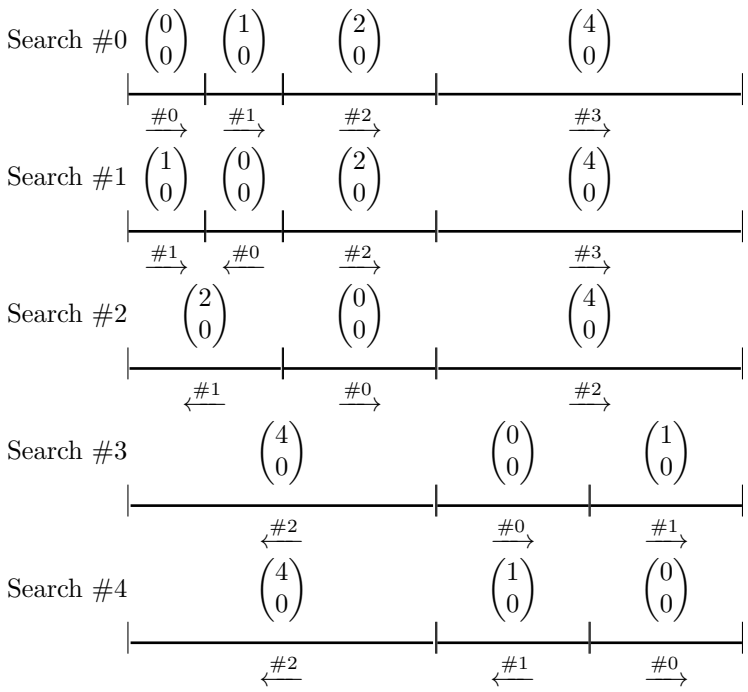


Figure 6.4: Full set of BNS-partitions (e=4)

Bidirectional search algorithm

Having proven that the method is correct and yields a complete set of matches. Here, we proceed to present the algorithm to generate BNS-partitions. Algorithm 12 shows the recursive procedure that derives all search partitions SP equivalent to a search up to e errors. The recurrence stops whenever the maximum error allowed is zero or the region to be searched is smaller than the error (line 3.). Otherwise, the algorithm splits the region into two equal parts (line 10.), sets the error limits of each part (line 11.), and composes each partition by recursively calling BiSearch-Partitions (lines 14,17.). For each partition, the region searched with the highest maximum error limits are enqueued to be searched in the proper direction (lines 13,16.).

For the sake of completeness, and because of its relevance to future sections, we present below the Partition-Split-Region (algorithm 13) and Partition-Split-Error (algorithm 14) algorithms. The former, divides the input region into two equal parts and the latter distributes the errors across the resulting region partitions.

Algorithm 12: Bidirectional search partition generation algorithm

```
1  BiSearch-Partitions(SP, R^p, R^l, e_min, e_max)
   input : SP search partition,R^p region starting position, R^l region length,e_min minimum
           global error,e_max maximum global error
   output: Computes all search partitions equivalent to a neighbourhood search from e_min up
           to e_max errors of the pattern in [R^p, R^p + R^l)
2  begin
3      if (e_max = 0 ∨ R^l ≤ e_max) then
4          // Stop partitioning
5          S ← (R^p, R^l, e_min, e_max)
6          SP ← SP · (S, →)
7          NS-Neighbourhood-Search(SP)
8      else
9          // Compute partition and error bounds
10         p ← Partition-Split-Region(R^p, R^l)
11         Partition-Split-Error(p, e_min, e_max)
12         // First Search Partition
13         SP ← SP · (S_01, →)
14         BiSearch-Partitions(SP,S^p_00,S^l_00,S_00.e_min,S_00.e_max)
15         // Second Search Partition
16         SP ← SP · (S_11, ←)
17         BiSearch-Partitions(SP,S^p_10,S^l_10,S_10.e_min,S_10.e_max)
18     end
19 end
```

Algorithm 13: Algorithm to split search into two equal regions for the partition

```
1  Partition-Split-Region(R_p,R_l)
   input : R_p region starting position, R_l region length
   output: p region partition
2  begin
3      (R^p_0, R^l_0) ← (R_p, R_l/2)
4      (R^p_1, R^l_1) ← (R_p + R_l/2, R_l/2)
5      S_00 ← R_0
6      S_01 ← R_1
7      S_10 ← R_1
8      S_11 ← R_0
9      p ← {S_00, S_01, S_10, S_11}
10     return p
11 end
```

Algorithm 14: Algorithm to compute the error bounds assigned to each region in each search partition

```
1  Partition-Split-Error(p, e_min, e_max)
   input  : p = S_00, S_01, S_10, S_11 region partition, e_min minimum global error,e_max maximum
            global error
   output: Assignment of error bounds
2  begin
3      // Split error
4      e_search ← e_max/2
5      e_extend ← e_max
6      // First partition
7      S_00.e_min ← 0
8      S_00.e_max ← min{e_search, S_00^l}
9      S_01.e_min ← 0
10     S_01.e_max ← min{e_extend, S_01^l}
11     // Second partition
12     S_01.e_min ← 0
13     S_01.e_max ← min{e_search, S_01^l}
14     S_11.e_min ← 0
15     S_11.e_max ← min{e_extend, S_11^l}
16 end
```

6.3.2 Bidirectional region search chaining

Once all bidirectional search partitions are generated, one of the trickiest parts of the algorithm is to connect neighbourhood searches across regions. Figure 6.5 shows an schematic diagram of the DP-table and the paths explored during a BNS-partition search. In this case, three searches are performed: two forward and one backward. Beyond the inherent complexities involved in an actual implementation of this method, there are some details that need to be taken into account.

First, the whole DP-table should not be considered from the beginning. Only the square delimited by the pattern regions searched so far should be computed. Note in figure 6.5 how the areas coloured in yellow get bigger and paler as the search considers more search regions. In this way, the effective size of the DP-table increases each time a new region is added to the search. It is key to note that invalid paths at previously searched regions – possibly due to exceeding the number of errors – may lead to valid ones when another region search is added to the whole search.

Also, we should not forget that we aim to search for the supercondensed neighbourhood of the pattern. For this reason, not all squares of the DP-table should be computed identically. While it is clear that the last search – computing the whole DP-table – should prevent free-end paths – depicted in red in figure 6.5) –, using the supercondensed algorithm to compute intermediate squares of the table can lead to missing solutions. Due to the fact that new valid paths may appear as the effective DP-table gets larger, internal squares in it must be computed using the full neighbourhood search. This guarantees that no valid path is ruled out prematurely.

Equally important is to detect when to switch to the next region and expand the search (i.e. chaining). Premature chaining would defeat the purpose of the bidirectional search – allowing errors too soon –, and late chaining could lead to missed valid search paths or the generation of redundant computations. For that, we consider the search of a region to be satisfied once there is

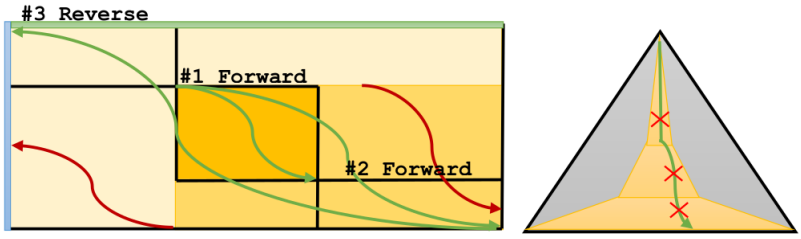


Figure 6.5: Bidirectional region search chaining using dynamic programing and depiction of the search space explored by a single search partition (against a full neighbourhood search).

a valid alignment path (i.e. the current search path aligns to the region of the pattern within the error boundaries). Computations past this point would generate paths that either are computed afterwards, or lead to free-end paths at the end of the table. As a consequence, the ϵ character should be consider in the main loop of the neighbourhood search, to allow chaining to more than one search region without adding any character to the search.

Additionally, note that the search may eventually change direction. In this case, the DP-table must be reversed to consider search paths in the opposite direction.

Figure 6.5 also shows a depiction of the search space explored by a single search partition. In the spirit of [Navarro et al., 2001], this figure tries to capture the effect of regulating the maximum number of errors allowed on each search region. In this manner, the search steadily allows for more solutions while constraining the paths explored and discarding string combinations not appearing in the reference. While this is a powerful technique to enhance the effectiveness of neighbourhood searches, in the next section we argue how forcing a minimum number of errors on these regions also has a very powerful effect on pruning the search tree.

6.3.3 Bidirectional search vertical chaining. Lower bounds

So far, we have emphasised the relevance of regulating the maximum number of errors throughout a neighbourhood search, and the effect this has on the number of paths explored in a neighbourhood search. However, searches based on BNS-partitions can be further constrained by imposing conditions on the minimum number of errors at each region. Note that, the first and last search partitions from figure 6.4, both allow for an exact search of the whole pattern. This implies that some search paths are explored more than once and we could reduce the computations of the search. Here, we propose constraining search partitions using the error bounds of other partitions. In this way, we apply a vertical chaining technique over the search partitions (section 5.5.2) to reduce the overall number of paths explored, and ultimately enhance neighbourhood searches based on this technique.

In the example, search #0 and # 1 were generated as the last step partitioning the search. Specifically, the region searched $S = (\binom{1}{0})$ was partitioned into $S_0 = (\binom{0}{0}, \binom{1}{0})$ and $S_1 = (\binom{1}{0}, \binom{0}{0})$. Although S_0 already covered for all searches up to one error that matched exactly the first region. Therefore there was no need to repeat this search path, and we could force S_1 to add an error in the first search region (i.e. $S_1 = (\binom{1}{1}, \binom{0}{0})$). In this way, each recursive partition imposes constraints on the next searches increasing the lower error bound. Figure 6.6 completes the BNS-partition of figure 6.4 (new minimum error bounds are marked in bold).

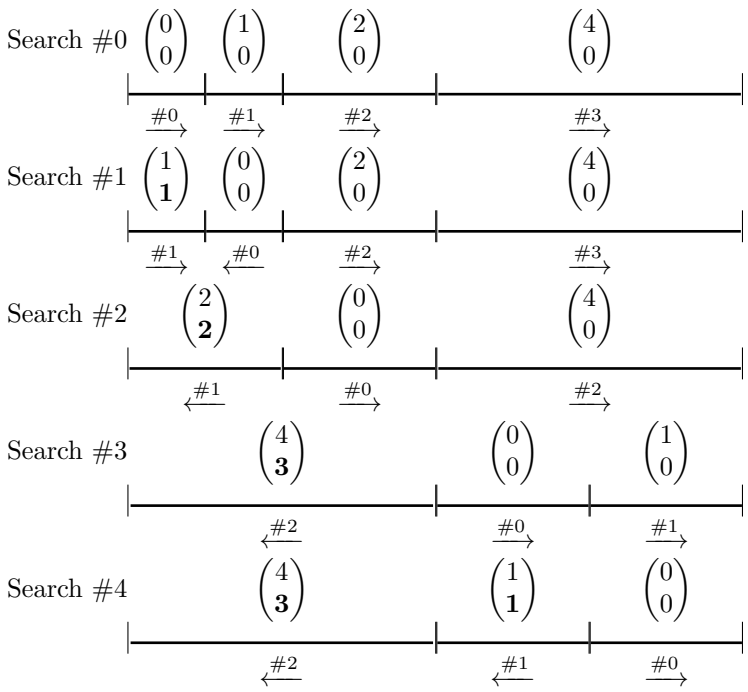


Figure 6.6: BNS-partitions imposing lower bounds (i.e. vertical chaining restrictions among BNS-partitions) (e=4)

6.3.4 Pattern-aware region size selection

Note that there is no mayor reason why regions have to be divided into equal chunks so as to produce recursive partitions. Moreover, practical experiments clearly show that such partitions are not necessarily guaranteed to explore less search nodes than unevenly distributed partitions. Furthermore, there no evidence that suggest that it is the best choice if applied to enough randomly generated patterns against a random text. The only reference to this topic [Kucherov et al., 2016] comes to prove this observation. However it is limited to Hamming distance with results at a very early stage.

As happens with filtering partitions, previous work on the topic has a fundamental flaw. As it is stated throughout this manuscript, not all patterns have the same structure and properties. For this reason, there is no reason to believe that a single partition scheme – oblivious to the content of the pattern and reference –, can lead to the optimal neighbourhood search in every case (i.e. spanning the least number or search paths).

For that reason, we propose to select regions with a similar number of occurrences in the reference. In this way we guarantee that the first region searched in each search partition is equally preconditioned. We heuristically approximate this equally preconditioned partition by counting k-mer frequencies and dividing across the regions required by the bidirectional search. In the experimental evaluation (section 6.5) we show the effect of this approach on the overall number of search nodes explored.

6.4 Hybrid techniques

The first appearance of the term hybrid search came in the paper [Navarro et al., 2001] to sketch the idea behind intermediate partition filters (section 5.3.3). That paper already acknowledged the difficulty in selecting the optimal partition of approximate factors and explained the limitation of the method in cases of searches with high degrees of error. As with many other contributions in this thesis, we try to select terms close to those accepted in the field for the sake of simplicity. In this case, the ideas presented below can be thought of as an extension of the hybrid model.

In short, the essence of a hybrid search is to combine filtering and neighbourhood search algorithms. As with most filtering methods, the goal is to balance the work between these two stages (thoroughly explained in section 5.1). As a counter to this approach, we propose a more elaborate – and efficient – way of combining both types of algorithm.

The hybrid techniques presented in this section will allow neighbourhood searches to benefit from filtering techniques in two ways. Firstly, by constraining the number of search paths by imposing bounds derived from previous filtering searches (section 6.4.1). And secondly, by preventing searches from reaching deep search nodes by filtering SA-intervals (i.e. candidates of the search so far), whenever the number of candidates is low (section 6.4.2).

6.4.1 Combining neighbourhood searches with filtering bounds

As mentioned several times previously, filtering techniques largely outperform other ASM techniques in most scenarios. However, for that minority of cases where filtering approaches do not attain the required results (e.g. deep searches), neighbourhood searches can be a suitable replacement – or may be the only alternative. Nevertheless, both methods are not incompatible and solutions considering a first filtering stage can be complemented with a neighbourhood search. In actual fact, the space of solutions explored by a filtering partition impose error boundaries on the search paths of a neighbourhood search.

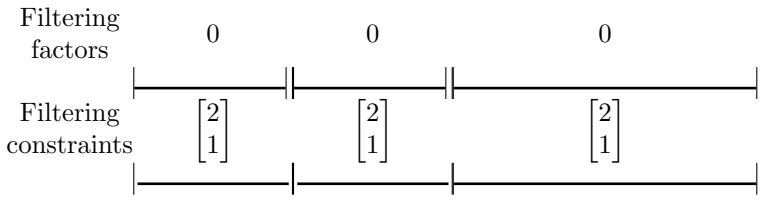


Figure 6.7: Search constraints from filtering 3 exact factors

To illustrate this idea, figure 6.7 shows a hypothetical scenario where the APP algorithm (section 5.4.3) has produced three factors – not necessarily of equal length. These three factors have been filtered exactly – without introducing errors in the seeds. As explained earlier, the search depth reached by the APP algorithm in this case is guaranteed to be complete up to two errors (section 5.3.2). Assuming we want to capture complete results up to four errors, we complement the results found so far with a bidirectional neighbourhood search. However, any match in the reference that matches exactly any of the factors from the APP partition has already been already found. Therefore, any complementary neighbourhood search should avoid generating paths leading to these matches. In other words, the neighbourhood search is forced to introduce errors in those

three regions. Figure 6.7 shows the local search restrictions imposed on each region of the pattern. Recall that these restrictions on the error are local to the region and we denote them using square brackets (section 5.5.2). Conversely, global restrictions within neighbourhood searches apply to the whole search – at the time the region is added to the search –, and are annotated using curly brackets.

Once the error constraints produced by the APP algorithm are known, using the same partition dimensions given by the APP factors, we produce the BNS-partitions – equivalent to a search up to four errors. Then, we override the lower bounds of the search regions taking into account local error constraints derived from the first filtering search. As a result, some search regions get their minimum number of errors raised. In some cases this can simply lead to discarding a whole search partition as the resulting error bounds are inconsistent.

Figure 6.8 shows the resulting BNS-partitions and error bounds. Note that search #3 has been forced to introduce an error on the first searched region. Yet, this search is not allowed to introduce any error at all. In fact, matches retrieved by APP already cover this search partition and, therefore, the whole of search #2 is discarded.

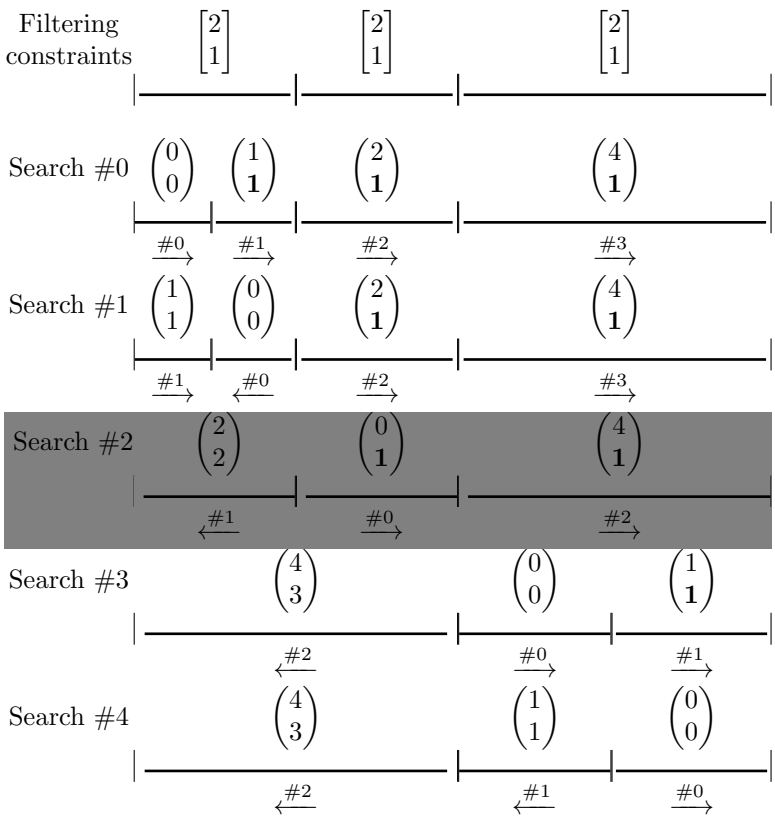


Figure 6.8: BNS-partitions imposing constraints from searching 3 exact factors

Note that figure 6.8 depicts both local and global error bounds on the search partitions of the neighbourhood search. That is, the actual algorithms that perform these region searches take into account both sets of constraints: from BNS-partitions (global), and from APP factors (local).

In short, this hybrid approach employs previously generated filtering partitions to precondition bidirectional searches. In this way, we apply the ideas behind vertical chaining (section 5.5.2) to further constrain BNS-partitions. The main benefit of this method is the possibility of combining filtering and neighbourhood searches, avoiding repeated computations. Furthermore, this enables us to perform deeper searches preventing filtering methods from saturating with the number of candidates.

6.4.2 Early filtering of search branches

Another powerful observation is the fact that any valid search path – leading to a valid match – has to be extended from the beginning of the search until the end. Yet, whenever a search path corresponds to a handful of matches in the reference (i.e. the SA-interval of the search is small enough), we could directly verify those candidates one by one – using any online filter (section 5.2) or a chain of online filters (section 5.5.1).

This is what we denote as early filtering of search branches and it exploits the decay property of search branches once the search path is large – and specific – enough (section 6.3). This method allows us to quit neighbourhood searches before reaching a leaf of the tree. In this way, it skips potentially expensive computations at the end of search partitions – which contain the search regions that allow most errors to happen. Furthermore, this method favours fast and simple verification procedures (i.e. online filters).

In practical terms, a heuristic algorithm very similar to the one used by the APP (algorithm 6) is used to determine the best point in the search path to stop and verify candidates. Specifically, the so-called "Local optimisation stage" (line 11.) is applied as the search progresses. In this way, a search path only nominates for early filtering if its SA-interval has less candidates than a given hard threshold. If that happens, the algorithm lets the search carry on a fixed number of steps before stopping the search – and all paths it could have spawned in the meantime –, and leaving the rest of the search for verification.

Despite it sharing similarities with other hybrid methods, the approach presented adapts to the needs of each search – and the structure of the pattern. As opposed to classic hybrid methods, our approach has no hard boundaries to switch from neighbourhood search and filtering. Hence it self-adapts to each scenario and applies each method whenever it fits best.

6.5 Experimental evaluation

In this section, experimental results for the previous methods are presented. In order to justify and quantify the benefits of the proposed techniques, we have benchmarked several algorithms and compared them against each other. Actual implementations of these algorithms have been included in the GEM Mapper version 3 (chapter 7). This allows us to extract practical conclusions from a fully functional tool which generates production-ready results. Throughout the section we focus on measuring the total wall-clock time spent by the tool as a whole, the number of search nodes spawned during the neighbourhood search, and the average depth of the search paths for all reads mapped.

All benchmarks have been run on a Intel i7-6500U (2.5GHz) equipped with 16GB of RAM using a Linux Ubuntu 16.04 (kernel 4.8.0). All tests have been run on a single thread and core in order to avoid biases from multiple threads interactions. In order to avoid memory latencies that could bias performance results, chromosome-1 of the human genome was used as a reference. Using a smaller index mitigates memory penalties and allows a focus on times strictly due to the neighbourhood search. Likewise, Illumina-like simulated reads – for chromosome-1 – of different lengths, have been used as inputs in order to keep the tests as realistic as possible.

We have selected seven algorithms, each representative of a neighbourhood search technique. In short, `Brute-force-NS` is a brute-force, full-neighbourhood search as presented in section 6.1. `Supercondensed-NS` is a refinement of the first to search for only the supercondensed neighbourhood (section 6.2.3). Next, `BNS` stands for bidirectional neighbourhood search and implements the basic bidirectional algorithm presented in section 6.3 (using equal-length search regions and imposing both upper and lower bounds). Based on `BNS`, we benchmark `BNS + PA` so as to select pattern-aware region size (section 6.3.4), and `BNS+EF` to early filter search branches (section 6.4.2). Finally, we include in the benchmark the hybrid algorithm `Hybrid-BNS(APP)` – which combines the APP algorithm with bidirectional neighbourhood searches (section 6.4) –, and the hybrid algorithm `Hybrid-BNS(APP)+EF` – which builds on the previous while allowing early filtering of search branches.

Method	Time	Search Nodes	Average Search Depth
`Brute-force-NS`	7.5h*	20.3G*	99.00*
`Supercondensed-NS`	154m*	8.08G	96.54
`BNS`	5.23s	12.74M	97.77
`BNS + PA`	3.11s	7.60M	98.48
`Hybrid-BNS(APP)`	3.00s	7.31M	98.56
`BNS + EF`	2.54s	6.20M	52.80
`Hybrid-BNS(APP) + EF`	1.58s	3.85M	55.13

Table 6.1: Benchmark results for neighbourhood methods (10K Illumina reads of 100nt, e=4%). Searches report all valid matches (i.e. complete set of results). Results marked with an asterisk have been estimated from computing a smaller sample of 1K reads.

Table 6.1 shows the results from searching 10K reads of 100nt up to four errors. Note that the time accounts for the total running time including APP filtering and verification of candidates if the method employs them. Also note that the results for `Brute-force-NS` and `Supercondensed-NS` algorithms have been estimated from a smaller sample of 1K (a full run would have taken an impractical amount of time).

As expected, the supercondensed search spawed considerably fewer search nodes than the full neighbourhood search (2.5 times less), effectively reducing the the total running time. However,

compared to the bidirectional search (BNS), the supercondensed search spawned >600x more nodes. The preconditioning strategy is strikingly effective at reducing the overall number of search paths explored. Ultimately, this enables mapping orders of magnitude larger datasets in a reasonable amounts of time.

Interestingly, the BNS+PA algorithm comes to prove the difference a good choice of region size can make in the total number of search nodes explored. In the results, the number of nodes spawned is reduced to 60% the number of nodes spawned by BNS – which uses equal-length search regions. As we can see, proper search preconditioning has a high impact on the search, leading to large improvements in the running time.

Similarly, the hybrid algorithm Hybrid-BNS(APP) reduces both the number of searched nodes and the running time – compared to the BNS algorithm. In actual fact, the strategy of Hybrid-BNS(APP) implicitly exploits the properties of selecting an adequate region size by using the factors produced by the APP algorithm. In this way, the effect of preconditioning the search path is very similar to that of BNS+PA. Additionally, it further reduces the number of search paths by discarding combinations already covered by the first stage of APP filtering.

In contrast to the previous methods, approaches based on early filtering avoid exploring paths until the end of the pattern (i.e. on average 98nt long paths). In this way, both methods which use this technique – BNS+EF and Hybrid-BNS(APP)+EF – search for just half of the pattern before switching to BPM verification of the candidates. This implies that previous algorithms carry out half of the search extending search paths leading to few matches. In these cases, there is no point in exploring by trial-and-error all the neighbourhood combinations; candidate verification proves to be a much more efficient approach. Additionally, we observe that the number of searched nodes explored by BNS+EF is reduced by almost half in Hybrid-BNS(APP)+EF. Again, this is the result of selecting more adequate region size partitions and imposing filtering conditions derived from APP.

Scaling with the read length

Traditionally, neighbourhood methods have been relegated to searching for very short sequences – in favour of filtering methods. In this way, these type of searches have always been deemed impractical for read mapping of long reads. The methods proposed in this thesis come to prove that this is not entirely true. Table 6.2 complements the results shown before with tests for increasing read length.

The first result to notice is that the basic BNS algorithm does not scale with the read length. Compared to the case of 100nt reads, the running time gets increased by a factor >27X when searching for 1000nt long reads. In contrast, the Hybrid-BNS(APP)+EF algorithm only increases the running time by 5.6X, meanwhile the sequences used are 10X larger. This is quite an impressive result taking into account that these searches are done up to e=4% in 1000nt long reads; that is, complete searches up to 40 errors.

This result heavily relies on the fact that the Hybrid-BNS(APP)+EF algorithm quickly shrinks the search space of each search partition and quickly switches from neighbourhood searching to filtering verification. Interestingly, the Hybrid-BNS(APP)+EF algorithm searches on average just 34nt of the pattern (i.e. less than the 4% of the read) by means of neighbourhood search. In contrast, all other methods that do not use the early filtering technique have to continue the search up to almost 1000nt. In the same way, the BNS + EF method avoids inspecting the majority of the sequence so as to find all matches up to 40 errors. However, improving the preconditioning and constraining the search with complementary filtering methods (i.e. Hybrid-BNS(APP)+EF) reduced the average number of bases inspected by half.

	Method	Time	Search Nodes	Average Search Depth
50nt	`BNS`	2.23s	5.25M	49.27
	`BNS + PA`	1.52s	3.58M	49.56
	`Hybrid-BNS(APP)`	1.47s	3.48M	49.59
	`BNS + EF`	1.76s	4.15M	39.51
	`Hybrid-BNS(APP) + EF`	1.23s	2.90M	42.1
100nt	`BNS`	5.23s	12.74M	97.77
	`BNS + PA`	3.11s	7.60M	98.48
	`Hybrid-BNS(APP)`	3.00s	7.31M	98.56
	`BNS + EF`	2.54s	6.20M	52.8
	`Hybrid-BNS(APP) + EF`	1.58s	3.85M	55.13
200nt	`BNS`	12.07s	26.65M	195.2
	`BNS + PA`	7.16s	15.81M	196.83
	`Hybrid-BNS(APP)`	6.85s	15.13M	197.01
	`BNS + EF`	3.30s	7.29M	63.02
	`Hybrid-BNS(APP) + EF`	2.24s	4.95M	64
1000nt	`BNS`	2.23m	153.59M	970.62
	`BNS + PA`	53.66s	56.46M	983.46
	`Hybrid-BNS(APP)`	51.85s	54.55M	984.82
	`BNS + EF`	11.42s	12.02M	67.91
	`Hybrid-BNS(APP) + EF`	8.89s	10.03M	34.9

Table 6.2: Benchmark results for neighbourhood methods using different read length (10K Illumina reads, e=4%). Searches report all valid matches (i.e. complete set of results).

It is also important to notice that these methods do not have a large impact on very short reads (e.g. 50nt). The reason why we do not see a real improvement in performance is that most reads of 50nt searched up to e=4% have a huge number of matches in the reference. Hence, the number of valid search paths in each search is very high and most lead to a valid solution. As a result, little effect can be expected from these pruning techniques as they generally have a small margin to reduce the number of cases considered.

Scaling with the error

In the same manner, we have benchmarked how the hybrid algorithms – `Hybrid-BNS(APP)` and `Hybrid-BNS(APP)+EF` – behave with increasing error rates. Both perform better than the rest at high error rates. However, in order to compare to a base line and emphasise the exponential nature of the problem, we have included estimated times for the `Brute-force-NS` and `Supercondensed-NS` algorithms using smaller inputs.

Table 6.3 shows the results of this benchmark. As we can see, estimated times for the basic neighbourhood algorithms grow exponentially in time with the error rate (i.e.) up to unaffordable costs. In contrast, both hybrid algorithms grow in time much slower. In the case of the `Hybrid-BNS(APP)` algorithm, the number of nodes increases 6.8x from e=4% to e=6%, and 6.4x from e=6% to e=8%. In the case of `Hybrid-BNS(APP) + EF`, these increments are 9.6x and 6.2x. Overall, in both cases the time spent and nodes explored in the case of e=8% remains reasonable enough to consider their use on a real tool.

Considerations towards practical use of neighbourhood searches

It is extremely important to bear in mind that, besides all the optimisations presented in this chapter, filtering methods greatly outperform neighbourhood searches in the majority cases.

	Method	Time	Search Nodes	Average Search Depth
e=4%	`Brute-force-NS`	7.5h*	–	–
	`Supercondensed-NS`	154m*	–	–
	`Hybrid-BNS(APP)`	3.00s	7.31M	98.56
	`Hybrid-BNS(APP) + EF`	1.58s	3.85M	55.13
e=6%	`Brute-force-NS`	52.5h*	–	–
	`Supercondensed-NS`	31h*	–	–
	`Hybrid-BNS(APP)`	20.66s	45.67M	97.53
	`Hybrid-BNS(APP) + EF`	15.18s	24.49M	50.71
e=8%	`Brute-force-NS`	639d*	–	–
	`Supercondensed-NS`	137.5h*	–	–
	`Hybrid-BNS(APP)`	1.74m	220.51M	96.95
	`Hybrid-BNS(APP) + EF`	1.35m	127.11M	43.79

Table 6.3: Benchmark results for hybrid neighbourhood search methods increasing error rate (10K Illumina reads of 100nt). Searches report all valid matches (i.e. complete set of results). Results marked with an asterisk have been estimated from computing smaller samples

Moreover, neighbourhood searches have a very high computational cost. Their use should be limited to those cases where filtering methods cannot fulfil the requirements of the search – in most cases with regard to search depth. In practice, neighbourhood searches should not be used standalone, but complementing a filtering algorithm (hybrid algorithms). Additionally, one must recall that neighbourhood searches prove their effectiveness in cases where it is not feasible to effectively determine seeds leading to valid matches, or where intrinsic repetitiveness of the pattern – and its substring – saturates filtering approaches with an unfeasible number of candidates to verify.

In the same way, we must highlight that throughout this whole chapter we have focused our attention and tests on cases where all the feasible matches within a given distance are to be retrieved (i.e. all-matches). This case represents the most demanding scenario and the most computationally expensive; hence our interest in evaluating algorithms for it. Nevertheless, neighbourhood searches can be used efficiently in other scenarios (e.g. "true-match") by means of adapting the finishing conditions of the search. Furthermore, they can be complemented with the notion of δ-strata groups (section III) to effectively stop the search of the true-match whenever a tie is found, or the mapping accuracy derived from the classification of the matches found in a δ-strata group is high enough.

In actual fact, this idea is implemented within the GEM Mapper version 3, when the mapping algorithm falls back to neighbourhood searches. Ultimately, this avoids excessive and unproductive computations when mapping "hard" sequences that require this kind of search to obtain accurate results.

Chapter 7

High-throughput sequencing mappers. GEM-mapper

This chapter is devoted to explaining the practical implementation of a production-ready mapping tool: the GEM mapper. This aligner implements the techniques and algorithms presented in previous chapters. In this chapter, we want to emphasise the practical aspects that allowed us to successfully turn theoretical insights into a practical tool. In particular, we present a detailed description of how the GEM mapper is structured and how its components embody the algorithmic techniques previously presented. Then, we examine some engineering issues affecting performance in a real implementation. Additionally, we present a GPU implementation of the GEM Mapper which exploits the massive parallelism capabilities of modern graphic processing units to enhance the performance of the most critical mapping routines.

In the last section of this chapter we present a benchmark comparing the GEM mapper with other cutting-edge mappers currently available. We aim to give experimental results and a practical comparison between the GEM mapper and other widely used mappers. We mainly focus on comparing performance and the quality of the results. In accordance with published literature, experiments show that GEM consistently outperforms other mappers in both speed and accuracy.

7.1 Mapping tools

Since the advent of the first mapping tools in the context of genomics and bioinformatics there has been constant work towards improving these tools and producing better results for the available sequencing technologies and experimental protocols. Since the scale of sequencing has been

increasing more than exponentially in recent years, the need for better tools that can cope efficiently with the analysis of this data has become paramount. In the past decade, there has been massive interest in mapping tools, mainly focused on performance (often at the cost of accuracy). Here we present a brief introduction to the mapping tools considered here for benchmarking purposes. The interested reader is referred to [Fonseca et al., 2012] for a full review of mapping tools available and their main characteristics.

CPU Mappers

From the multiple mapping tools available that run using general purpose processors (i.e. CPU), we have selected those that stand out from the crowd because of their performance, the quality of their results, or both.

First, **BWA-MEM** is the latest mapping tool released from the BWA family. Based on the BWT (FM-Index) implementation of the tool BWT-SW [Lam et al., 2008], this tools has a small memory footprint and performs very fast in practice. Its alignment algorithm is based on extracting maximal exact matches (MEMs) and verification using an SIMD SWG algorithm. In case the algorithm does not find any matches with high MAPQ, it resorts to a re-seeding process considering smaller seeds and more candidates. In practice, it is a very accurate mapper which scales adequately with read length up to relatively long sequences. As a result, it is widely used by many researchers in the field.

Another very popular mapper is **Bowtie2**. It is a mapping tool designed to quickly report a first good mapping location. Nevertheless, it can retrieve multimaps at the cost of expensive running times. Bowtie2 is based on an custom FM-Index implementation, designed to have a small memory footprint. As for its mapping algorithm, Bowtie2 is based on a seed-and-extend technique; it extracts fixed-length, partially overlapping seeds and verifies them using a SIMD striped vectorial dynamic programming [Farrar, 2007]. In case the first round of generated seeds is not sensitive enough, it resorts to a second round of re-seeding – with different seeds – as a best effort to map the read. Despite being a widely used mapper, its effectiveness is limited to relatively short reads. In this way, experiments show that for reads beyond 300nt its performance severely deteriorates.

An equally relevant mapping tool is **Novoalign**. Although not particularly fast, this mapping tool produces very accurate results. In fact, it is often selected when performance is not an issue and high quality results are required. Novoalign is a commercial software package and therefore, little is known about its internal implementation details.

Yet another interesting mapper is **SNAP**. This mapping tool relies heavily upon a big hash table to exploit larger seeds. In fact, SNAP takes advantage of ever-increasing sizes of RAM to justify the high memory footprint of its index. Nevertheless, this tool delivers very good performance in practice.

The last CPU mapper selected for the benchmarks is **HPG-aligner**. Unlike the others, the HPG-aligner employs a suffix array to support a seed-and-extend algorithm. In practice, this mapper performs relatively fast and produces results of a quality similar to BWA-MEM and others.

GPU Mappers

In addition to the mappers mentioned, some others have been designed to exploit modern GPU architecture and implement faster mapping tools. Though some of these tools are still prototypes, great advances have been made to prove the suitability of GPU towards accelerating sequence mapping.

nvBowtie is a GPU implementation of the widely used mapper Bowtie2. This GPU-mapper has been completely rewritten from scratch to adapt to the GPU architecture while retaining the same spirit as the original Bowtie2. In this way, it reproduces many of the features of the original tool while delivering remarkable speed-ups compared to it. In addition to this, we have included **Soap3-dp** and **CUSHAW2** – both GPU based mappers – in the benchmarks as they are both widely used and best representatives of the performance achieved by using modern GPUs.

7.2 GEM Mapper

Since the first release of the GEM-mapper, its main purpose has been to deliver high-quality results with predictable accuracy. In fact, the GEM project was born to deliver a mapping tool that could produce alignments with a guaranteed level of search depth and accuracy. Because of that, it has always offered the user a controllable way of specifying the required search depth. Historically, the design has always relied on some form of filtering stage, with a fallback neighbourhood search algorithm in case the requested search-depth could not be achieved by means of filtering only. Its reliance on these two complementary string matching approaches to reach a predictable level of accuracy represents one of the most important characteristics of the GEM mapper.

The following subsections describe in detail the structure and algorithmic workflows of the GEM-Mapper. Many experimental figures are presented throughout this sections, focusing on the reference use case of resequencing human reads, both real and simulated, of different lengths. The examples we present aim to showcase design decisions and illustrate how they impact the internal workings of the GEM-Mapper.

7.2.1 GEM architecture

In this section we would like to present the overall architecture of the GEM-mapper (a schematic illustration can be found in Figure 7.1), with its main modules and their purpose. The goal is to serve as an introduction to the following sections, where each part is presented in detail.

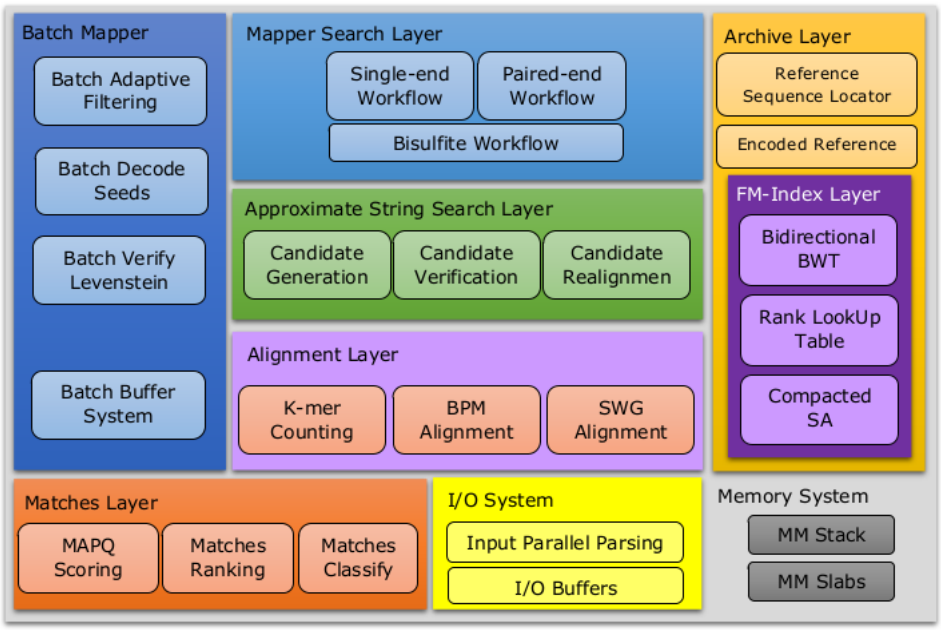


Figure 7.1: Architecture of the GEM-Mapper (Version 3)

The GEM architecture is structured as several abstraction layers, with each layer playing a precise role. Firstly, the search modules represent the most high-level routines of the GEM-Mapper. Besides building and handling GEM reference files, these modules are in charge of performing the different algorithmic workflows used when mapping.

Second come the approximate string matching modules, which constitute the main part of the mapper. They are organised according to the classical steps involved in filtering algorithms – i.e. candidate generation, verification and realignment. In turn those modules make use of the alignment modules, that is the algorithms and routines in charge of performing pairwise alignment and implementing individual filters.

In the end, intermediate results of the mapping process are gathered and handled by the matches modules. They rank, classify, and score all the matches produced by a given search.

As a backbone to all the routines of the mapper, the GEM Archive and FM-Index modules are encapsulated in the archive layer. This is the place where GEM implements the encoded reference, the BWT, the FM-Index and other related data structures and functions. Last but not least, GEM implements its own I/O modules to enable faster parallel parsing of FASTQ/FASTA files, and its own memory management subsystem to avoid overheads derived from excessive use of library malloc/free functions.

In general, all the GEM algorithmic workflows are designed in such a way that they can be reused by higher level workflows. Note that the candidate verification module constitutes the basic workflow of alignment and hence other modules can call it several times during their operation. For instance, the single-end workflow uses candidate verification to resolve SE alignments, and in turn is used by the paired-end workflow.

7.2.2 Candidate verification workflow

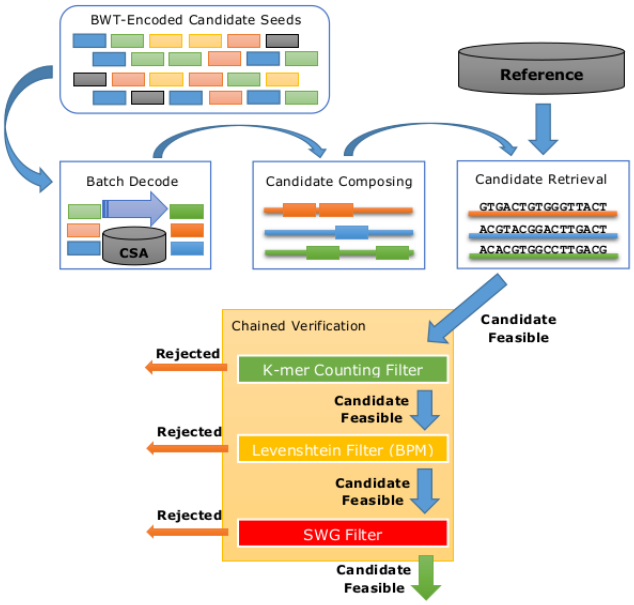


Figure 7.2: GEM candidate verification workflow

Candidate decoding and composition

Candidate verification starts with a set of positions to be verified against a sequenced read (i.e. aligned). The first difficulty is translating the SA-space encoded positions to text-space. For that, and depending on the number of candidate positions to decode, we use a batch decoding procedure

to prefetch both accesses to the FM-Index and the CSA – and therefore alleviate slowdowns due to page-faults accessing the GEM Archive.

The next step is to put all the decoded candidate positions together so as to derive the candidate region of the genome likely to align to the sequence read (compose candidates). It is important to note that more than one candidate position can lead to the same genomic region. This is due to the fact that multiple seeds can derive to the same locus on the genome. To put it in numbers, using the simulated human dataset of 100nt, the number of candidates shrinks by 20% when all the positions are arranged together. At the length of 1000nt the effect is even higher reaching a reduction of 60%.

Note that other mapping algorithms would perform the same operation but with the purpose of chaining the seeds onto those nearby locations and derive an alignment via extension (i.e SWG alignment). Seed information can come as a handy hint at the time of verification. Nevertheless, there is nothing that guarantees that the placements of the seeds in the region at hand is optimum. For some cases, a region of the read can generate multiple seeds that match the candidate region several times. In case of conflict, greedy seed-chaining policies do not guarantee optimality and thus, some candidate regions might be mistakenly discarded.

In our case, the GEM mapper aims to analyse the resulting candidate region without the information given by the seeds to check if it aligns against the read within a given error. Moreover, we must extend the scope of the candidate region by the maximum error allowed to consider worst-case alignments.

Before any verification can be performed, the candidate region must be retrieved from the reference. In the case of the GEM-mapper, the reference is stored as plain ASCII text in the GEM archive. However, only the forward strand is stored as the reverse complement can be easily derived from it. Other approaches employ the BWT of the index so as to decode the candidate region text character-wise doing LF-mappings. However, these approaches are much slower and generate more sparse memory accesses – generating inefficient memory accesses – than just having the plain reference stored explicitly in memory. The only downside is extra memory consumption, nevertheless modern computer architecture allows the integration of large memory banks and this extra memory cost should not represent a limitation nowadays. Additionally, we should consider cases where the candidate region in the reference is not large enough to cover the read and dangling ends appear. In these cases, the reference module must generate a padded candidate region so as to enable regular verification procedures transparent to the reference trims.

Chained candidate region verification

At this point, the candidate region is ready to be verified. This is done by means of a chain of verification algorithms. First, a q-gram filter (also denoted k-mer counting) algorithm quickly discards the most divergent candidates from the read. Experimental results show that on read lengths of 100nt, k-mer counting filter filters-out almost 30% of the candidates. Second, an implementation of Myer's Bit-Parallel algorithm (BPM) is used to compute the edit distance between the remaining candidates and the read.

Until this point, all the initial candidates are verified so as to rank them together before a re-alignment step. During re-alignment, from the least distant candidate to the most divergent, candidate regions are fully aligned against the read using a banded Smith-Waterman-Gotoh (SWG) algorithm. This process can prematurely stop if the maximum matches to report – specified by the user – is reached and no subdominant match still to realign can achieve a better SWG-score than the current realigned matches.

Interestingly, at this point it is easy to derive a minimum and maximum edit distance bound of each region candidate. The minimum bound will help to determine if a region candidate can obtain a better SWG-score than previously re-aligned candidates. Meanwhile the maximum bound will help to reduce the effective band used within the SWG-alignment without losing optimality.

Candidate realignment is one of the most important bottlenecks of modern read mappers. That is not only because of its quadratic complexity, but also due to the need for storing the whole alignment matrix – or just the band – in memory and back-tracing it afterwards. As the read length increases, so does the stress in the realignment step. Therefore it is important to limit this stage to the smallest number of candidate regions possible.

Re-alignment: A note on SWG based mappers

In recent years, there has been a marked trend for mappers to use SWG as an alignment metric. At the end of the day, this is forcing mappers to acquire SWG metrics for no clear reason. One should not forget the main purpose of a mapper and its place in the analysis pipelines. A genomic mapper is ultimately designed to recover the true position a read was generated from at sequencing. That is, a mapper is expected to maximise the number of TP reported mappings with reduced, if not null, FP (noise). Progressively, computing the true biological shape of the alignment (i.e. CIGAR) has become a second requirement for mappers. For this reason, there has been a marked trend towards mappers implementing SW-based algorithms with some predefined scoring weights. Ultimately, this has silently become a requirement towards compliant downstream analysis in many genomic pipelines. As a result, many mappers have been forced to add this computationally intensive stage into their internal workflow becoming, in many cases, the main bottleneck of the mapper. The task of computing the most likely biological CIGAR might be misplaced and the true aim of performing a SWG alignment may have been misunderstood.

A genomic mapper usually operates at the level of single reads. This severely restricts the overall vision a mapper can have of the whole genome and how all the reads map against it. It is quite common for many analysis pipelines to have a re-alignment stage after mapping. Reads mapping to the same locus are considered as a whole (pileup) and then, aligned together to derive from the consensus the most likely shape of the real biological alignment (multiple re-alignment). This approach benefits from the aggregated information coming from all sequenced reads at a given locus of the genome. Not only that, but it is also robust to spurious reads falsely mapped as the consensus as the pileup can easily reveal any discordance.

Therefore, we should re-think the real purpose of performing an SWG alignment at the stage of mapping. It might not be to determine the real biological CIGAR but to better rank all the mappings found for a certain read and to determine the most plausible locus of the read. In other words, to improve the accuracy of the mapper. As a matter of fact, many aligners and other tools do not rely upon the use of the SWG metric and interestingly, it does not affect the quality of their results.

7.2.3 Single-end alignment workflow

Single-end alignment workflow follows the principle that not all reads are equal and some of them require more computational effort to be properly aligned. In this way, the workflow is structured in a modular and incremental fashion (figure 7.3). Depending on the structure of the read, the mapper will perform different searches until the read is considered properly aligned or the mapper can assess that no plausible result can be found. In this way, this workflow is composed of three major search stages: Adaptive Filtering Searches (AFS), Neighbourhood Searches (NS)

and Local Alignment Searches (LAS). Each search stage employs the information derived from the previous search stages. The goal is to further explore the search space, avoiding repeated and unnecessary computations.

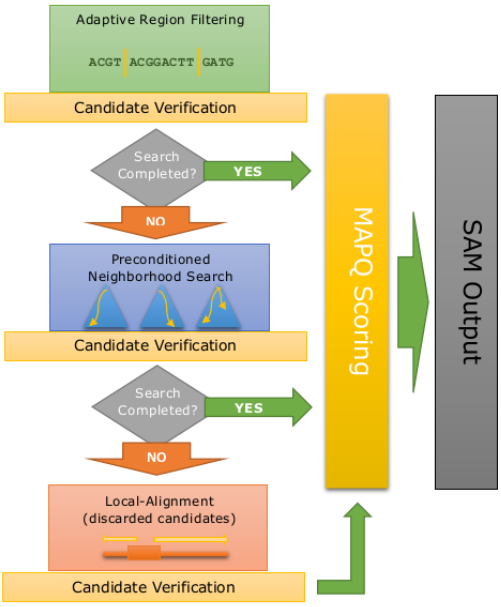


Figure 7.3: GEM single-end alignment workflow

Classifying reads according to their difficulty in being aligned

Interestingly, input reads can be classified with respect to the computational difficulty of aligning them. In section 3.3 Genome mapping accuracy, we explained that for a given read there is a certain search depth needed to fulfil the search accurately. In most cases, this search space can be explored easily by means of simple filtering techniques. However in some other cases, there is need for more aggressive search algorithms, as filtering approaches are simply not good enough. Experimental numbers show that at the length 100nt using AFS, for more than the 80% of the reads, the search explores more solutions than strictly necessary. The remaining 20% corresponds to reads that cannot be mapped or where there is no certainty that there are no more matches close to the ones found (i.e. possibility of finding ties).

Attending to this, the "easy" reads will only perform the AFS, meanwhile the "hard" reads will have to continue through NS. If this last stage in not able to find any global alignment within given error limits, all the discarded candidates generated so far will feed to the LAS for it to look for local-alignments. Therefore, local alignments will be output only if no good global alignment can be found. This approach aims to exhaust the search of global alignment – first with fast algorithms and them with more computationally intensive and accurate ones – and then resort to local-alignment.

Adaptive Filtering Search

The first stage for all reads is based on an adaptive pattern partitioning which splits the read into variable-length regions with a minimum number of occurrences in the reference. Afterwards,

all the encoded candidates generated from these regions are verified (i.e candidate verification workflow).

As stated above, for the majority of reads, this step will reveal unambiguously the nature of the read (i.e. unique, tie, etc). For the remaining cases, the search will have to go further so as to better resolve the mapping. Classically, when the initial filtering search could not deliver accurate enough results, a re-seeding stage would take place. During this stage, filtering thresholds would be increased so as to tolerate more candidates during verification at the cost of more computation. The aim would be to extract more seeds/regions to explore further the space of solutions. Nevertheless, filtering has its limitations on the number of candidates that it is feasible to verify just to increase one error deep in the search.

Preconditioned Neighbourhood Search. Further tree pruning

As opposed to re-seeding, the GEM-Mapper employs a different approach based on a bidirectional neighbourhood-search preconditioned with the search bounds derived from the AFS (chapter 6). This algorithms searches further the search space until it finds a valid match for the sequence (always exploring complete strata).

Local Alignment Search

Were the previous methods to fail when searching for a valid match, a local alignment step if perform as to report high quality local matches. For this, all seeds failed to align are extended using a SWG algorithm in other to delimit the part of the alignment that locally scores better.

Matches re-alignment, MAPQ scoring and output

Afterwards, all found matches are sorted and classified. Depending on the parameter options, the GEM-mapper (re)-aligns the most relevant matches to the search, and produces a complete alignment (i.e. CIGAR). The distance function can be configured via parameters. Nonetheless, by default a SWG alignment algorithm is performed.

In practice, due to the repetitive nature of many genomes, several of the reported matches share the exact same CIGAR. That is, many reported matches belong to repetitive copies of the same locus (section 3.3.3). In other to exploit this observation, the GEM-mapper implements an alignment cache that check the reference sequence of every match. In this way, we avoid duplicated re-alignment calls to expensive SWG routines.

Finally, alignments are MAPQ scored attending to the δ-strata group (section III) they belong to, and reported to the user.

7.2.4 Paired-end alignment workflow

The paired-end mapping workflow of GEM is heavily based on the single-end mapping workflow. In this way, both ends are mapped independently and the resulting matches for both ends, cross-paired, take into account feasible insert sizes (according to the distribution of pairs observed so far, and the parameters provided by the user). Figure 7.4 shows an scheme of the workflow.

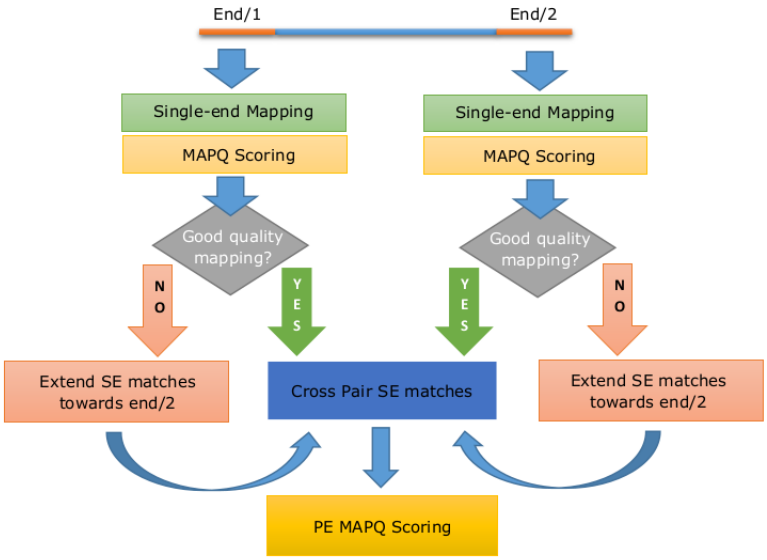


Figure 7.4: GEM paired-end alignment workflow

7.3 GEM-GPU Mapper

In addition to the CPU standard mapper, GEM has also been ported to the GPU. To that end, the most critical stages of mapping have been implemented in the GPU to offload these computations and relieve the CPU. Figure 7.5 shows a scheme of the modular interaction of the CPU mapper with the GPU kernels. In this way, the adaptive pattern partition – or adaptive region filtering module –, the decoding of positions from SA-space to text-space, and the BPM verification algorithms are computed using one or many GPUs. In this way, the results output by both versions of the mapper are "diff" identical.

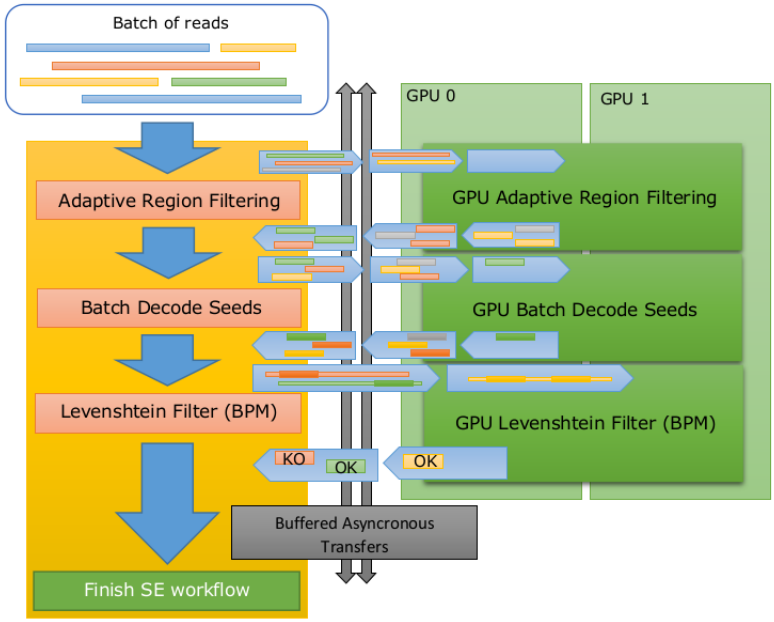


Figure 7.5: GEM batch mapping

Using high performance implementation of these modules (published in [Chacón et al., 2015, 2013, 2014b,a]) the CPU mapper gathers batches of searches and sends them to GPU. Once computed, these batches are returned to the CPU where the regular work-flow for each search is re-established. Note that to ensure that the process of sending and retrieving searches to the GPU pays off, the batches have to be large enough to guarantee that the speedup achieved at the GPU compensates for the transfers and overheads involved in the communication. Furthermore, transfers to the GPU are done asynchronously to enable multiple threads to use the GPU and avoid waiting and synchronisation overheads.

7.4 Experimental results and evaluation

In order to evaluate the GEM mapper and compare it with other state-of-the-art mappers, this section presents several benchmarks performed on both real and simulated data. The purpose of the benchmark is to present practical performance results of the GEM mapper and all the optimisation techniques implemented within. We aim to prove the performance benefits of the GEM optimisation in real case scenarios. In addition, we analyse the quality of the results delivered by different mappers and for different datasets. We want to put special emphasis on comparing the different trade-offs between speed and accuracy that current mapping tools can offer in practice.

For this purpose, as mentioned in section 1.3, we have selected real datasets that accurately represent current sequencing technologies (table 1.2), and simulated corresponding datasets to perform further analysis on the quality of the results (table 1.1). We aim to give a complete description of the behaviour of the different tools with different real datasets and technologies. Besides this, we want to point out how each of these mappers scale with the read length – as long read technologies are evolving and becoming ever more present in regular experiments.

For the benchmark, we have selected a series of mappers relevant because of their competitive performance, quality, or because they are widely used and relevant in the field. From all the available CPU mappers, we have selected: Bowtie2 [Langmead and Salzberg, 2012] because it is a widely used mapper, BWA-MEM [Li, 2013] because of its popularity, performance and quality, SNAP [Zaharia et al., 2011] and HPG [Tárraga et al., 2014] due to their great performance, and Novoalign [Hercus, 2012] because of its high quality results. On the GPU side, we have selected those practical mappers which could be run successfully with different real datasets without crashing and which have high enough performance to match the rest of the mapper. In summary, we selected CUSHAW2 [Liu and Schmidt, 2014], SOAP3DP-GPU [Luo et al., 2013], and NVBowtie2 (unpublished). Additionally for each mapper, different execution parameters have been used in an effort to show the different compromises between quality and performance that each of these mappers can offer in practice.

All the benchmarks have been executed within the same computing node equipped with an Intel(R) Xeon(R) CPU E5-2640 v3 at 2.60GHz (8 cores, 16 threads). This node is equipped with 128 GB of RAM available and runs a Red Hat Enterprise Linux Server release 6.7 (Kernel 2.6.32). Note that all executions have been performed in the computing node in exclusive mode to reduce any noise from other processes and jobs. In order to measure computing scalability of the mapping tools and analyse their behaviour in a high-performance computing node, all mappers have been executed using 16 threads (i.e. using command line options provided by each mapper). Furthermore, many measurements have been repeated several times to ensure stability and avoid mayor running time variations between executions.

7.4.1 Benchmark on simulated data

Simulated single-end

In simulated datasets, figure 7.6 shows the speedups achieved by GEM compared to other mappers – and for different read lengths (table 7.1 shows all times).

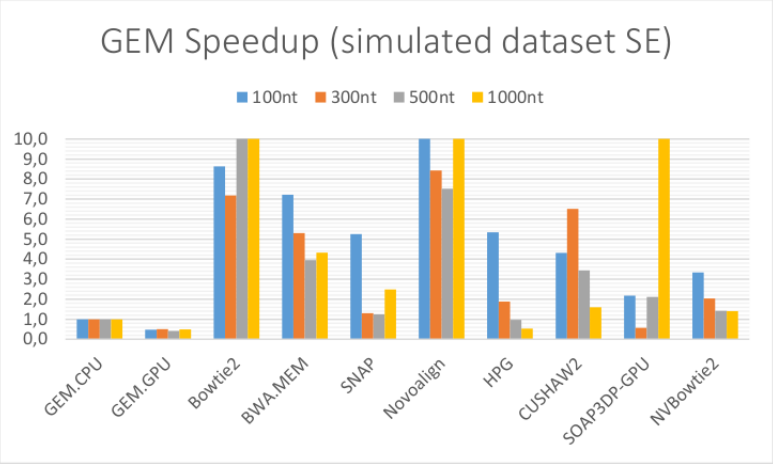


Figure 7.6: Comparative GEM CPU Speed-up for different simulated read length SE. All mappers run using defaults

In general, GEM consistently outperforms the rest of the mappers achieving better running times. For instance, at 100nt GEM obtains an speedup of 7.2x compared to BWA-MEM, 8.6x compared to Bowtie2, 15.8x compared to NovoAlign, and 5.3x compared to SNAP – selected because of its remarkable performance. As the read increases in length, some mappers cannot scale and take a large amount of time to finish (e.g. Bowtie2, Novoalign and SOAP3DP). In the case of GEM, times increase linearly with the read length. For instance, at 1000nt reads GEM is about 2 times slower compared to mapping 100nt reads. In this way, GEM – like BWA-MEM – takes advantage of long reads, being more efficient per base aligned with longer reads. In contrast, other mappers like HPG improve the running time with longer reads. However, this is at the expense of accuracy as the total number of true positives found decreases with respect to other mappers (table 7.2).

From the GPU standpoint, GEM-GPU achieves an speedup of 2x – with respect to the CPU version – consistently in all read lengths, making it the fastest among all mappers both CPU and GPU. In particular, GEM obtains an speedup of 15x compared to BWA-MEM, 9x compared to the fastest mode of Bowtie2, and 4x compared to SNAP. Note that it produces the same results as the CPU version, so no loss of quality is produced – compared to the CPU mode. It is worth mentioning that other mappers achieve quite good performance results. This is the case of SOAP3DP-GPU, HPG, and SNAP, which show remarkable speed up compared to widely used mappers like BWA-MEM. Though GEM outperforms all of them, they come to prove that "de facto" tools in the field can be largely improved in term of performance.

Nevertheless, performance should not be evaluated alone. It is crucial to take into consideration accuracy results. In table 7.2 we show the total number of true positives reported by each mapper for simulated single-end datasets. The reader must keep in mind the relevance of small margins in the total number of true positives found and the effects this can have on the downstream analysis

Wall-clock time	100nt	300nt	500nt	1000nt
GEM.fast	393	566	812	898
GEM.sensitive	671	620	885	920
GEM.fast.GPU	188	302	342	439
GEM.sensitive.GPU	466	430	405	456
Bowtie2.veryFast	1753	1866	8830	25405
Bowtie2.default	3393	4061	20535	80832
Bowtie2.verySensitive	7411	9750	40311	127756
BWA.MEM.default	2836	3005	3207	3885
BWA.MEM.c500	2843	2995	3224	3868
BWA.MEM.c1000	3104	3124	3366	4004
BWA.MEM.c16000	7057	5125	4332	4189
BWA.MEM.c20000	12728	9138	7271	6269
SNAP.h50	811	540	523	811
SNAP.default	2065	736	1015	2237
SNAP.h1000	2162	3200	1843	3564
SNAP.h2000	4662	5722	2924	3849
Novoalign	6221	4770	6102	9611
HPG.fast	2030	827	568	399
HPG.default	2098	1065	782	474
HPG.sensitive	2092	1433	1154	690
CUSHAW2.default	1697	3687	2787	1437
CUSHAW2.sensitive	1693	3650	2843	1421
SOAP3DP-GPU	854	326	1718	18640
NVBowtie2.veryFast	987	816	770	n/a
NVBowtie2.default	1312	1151	1156	n/a
NVBowtie2.verySensitive	2045	1995	1907	n/a

Table 7.1: Running time in seconds for simulated single-end datasets

[Hwang et al., 2015; Li, 2014]. In this case, mapping high quality Illumina sequences (i.e. low average error rate) against a well known reference genome, allows reporting more than 90% of the true-positives for all mapper. However, the devil is in the details and a margin of ¿2% can lead to the discovery of potentially interest variants [Laurie et al., 2016].

In this case, GEM consistently outperforms all the other mappers. Nonetheless, we must point out that HPG at 100nt achieves better percentage of true-positives than GEM by 0.23% – though running 5x slower. As the table shows, the number of true-positives found increases with the read length for the majority of mappers. This an expected result as long reads are more specific and mapping tools can extract more seeds – increasing the probability of finding the true locus of the sequence. HPG and SNAP are the exception to this; probably due to their inability to handle reads of 1000nt and larger. It is also important to highlight that GEM spends a non negligible amount of time in the sensitive mode to report little extra true positives. Ultimately, at length of 500nt and 1000nt, this search can be deemed as unproductive as the total number of true positives remains unchanged. In most cases, this is due to the fact that the true positives are found in a deep stratum, and the search is abandon as less distant matches are found before. Note that, even if we force the mapper to explore this strata, no sensible MAPQ scoring function is likely to score these matches better than the ones found before.

True positives	100nt	300nt	500nt	1000nt
GEM.fast	98.76	99.77	99.87	99.94
GEM.sensitive	99.01	99.79	99.87	99.94
Bowtie2.veryFast	94.93	96.82	97.62	98.37
Bowtie2.default	96.06	98.07	98.47	98.90
Bowtie2.verySensitive	96.25	98.15	98.52	98.92
BWA.MEM.default	98.07	99.19	99.33	99.51
BWA.MEM.c500	98.07	99.19	99.33	99.51
BWA.MEM.c1000	98.07	99.19	99.33	99.51
BWA.MEM.c16000	98.07	99.19	99.33	99.51
BWA.MEM.c20000	98.06	99.18	99.32	99.51
SNAP.h50	95.83	98.15	98.26	90.30
SNAP.default	96.41	98.21	98.28	90.30
SNAP.h1000	96.57	98.22	98.29	90.31
SNAP.h2000	96.62	98.23	98.29	90.31
Novoalign	95.23	95.23	95.23	95.23
HPG.fast	98.99	99.05	98.58	97.72
HPG.default	98.99	99.17	98.67	97.87
HPG.sensitive	98.99	98.97	98.31	97.41
CUSHAW2.default	96.14	97.85	97.73	97.71
CUSHAW2.sensitive.K40	96.36	98.24	98.12	98.28
SOAP3DP-GPU	98.44	99.00	99.07	99.39
NVBowtie2.veryFast	95.23	97.97	98.43	n/a
NVBowtie2.default	95.77	98.13	98.52	n/a
NVBowtie2.verySensitive	95.86	98.14	98.53	n/a

Table 7.2: Percentage of true positives found for simulated single-end datasets

In regard to quality, GEM clearly shows one of the best ACC to FPR ratios, certainly as good as BWA-MEM and Novoalign. Meanwhile mappers – like CUSHAW, nvBowtie, or Bowtie2 – introduce more FPs than the others studied (Figure 7.7 shows the ROC curves for 100nt). We observe that BWA-MEM surpasses GEM at high FPR (i.e. between 10^{-3} and 10^{-2}, corresponding

to MAPQ values below 20). In those cases, it becomes very difficult to accurately solve mapping conflicts and score matches. In case of conflict, GEM usually chooses to score those matches as ambiguous setting MAPQ=0. In contrast, BWA-MEM's scoring formula behaves well in practice for those cases and obtains a small advantage over GEM.

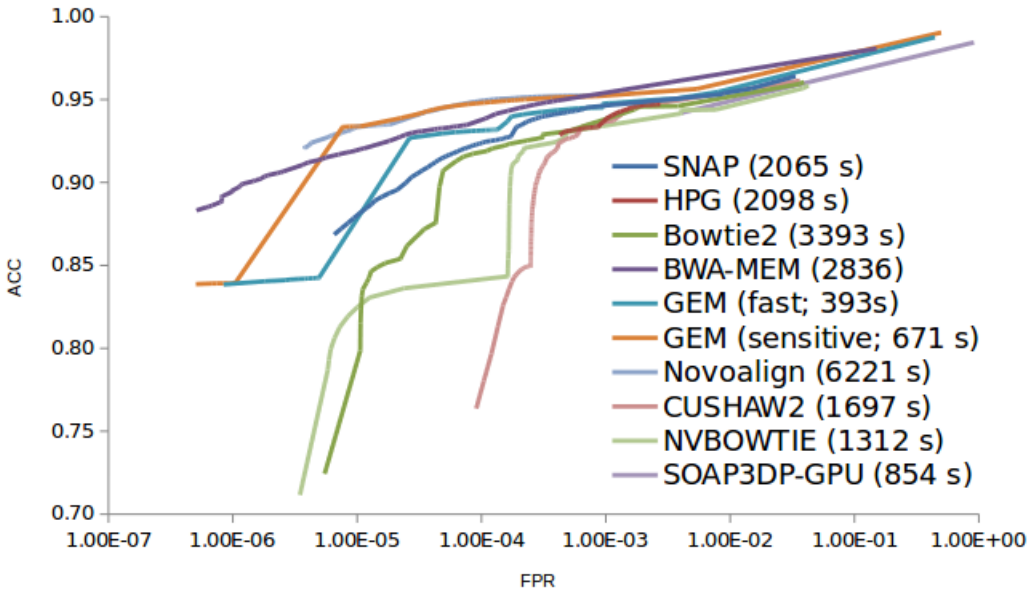


Figure 7.7: ROC curves for simulated SE datasets (default parameters)

It is usually the case that downstream analysis tools select only a subset of the reported read attending to its MAPQ score. Accordingly, tables 7.3 and 7.4 show the percentage of mappings reported – with respect to all reads in the dataset – with certain MAPQ scores or greater. In this way, we benchmark how much usable data is reported to any tool using these mapping results. As the tables show, compared to BWA-MEM, the GEM mapper consistently reports more usable mappings at the different levels of MAP score.

Beyond these results, it is more important to highlight that at the end of the day not all mappings reported are used. In this way, only those scored as highly confident are used. Thus, it is crucial for a mapper to accurately evaluate all reported mappings and assess which can be used with confidence.

	100nt			300nt		
MAPQ ≥	40	30	20	40	30	20
GEM.fast	92.48%	92.66%	93.64%	96.51%	96.52%	97.21%
GEM.sensitive	93.28%	93.31%	94.40%	96.47%	96.48%	97.22%
BWA.MEM.default	90.60%	91.85%	93.21%	95.01%	95.48%	95.98%
BWA.MEM.c500	90.60%	91.85%	93.21%	95.01%	95.48%	95.98%
BWA.MEM.c1000	90.62%	91.86%	93.22%	95.03%	95.50%	95.99%
BWA.MEM.c16000	90.65%	91.87%	93.23%	95.05%	95.52%	96.00%
BWA.MEM.c20000	90.64%	91.87%	93.23%	95.05%	95.51%	96.00%

Table 7.3: Percentage of reported true-positives – with respect to all reads – with a certain MAPQ score or greater. Simulated human reads at 100nt and 300nt.

	500nt			1000nt		
MAPQ ≥	40	30	20	40	30	20
GEM.fast	97.11%	97.11%	97.68%	97.72%	97.72%	98.20%
GEM.sensitive	97.10%	97.10%	97.68%	97.72%	97.72%	98.20%
BWA.MEM.default	95.84%	96.16%	96.64%	96.55%	96.82%	97.16%
BWA.MEM.c500	95.84%	96.16%	96.64%	96.55%	96.82%	97.16%
BWA.MEM.c1000	95.86%	96.17%	96.65%	96.55%	96.82%	97.17%
BWA.MEM.c16000	95.87%	96.17%	96.66%	96.56%	96.82%	97.17%
BWA.MEM.c20000	95.87%	96.17%	96.66%	96.56%	96.82%	97.17%

Table 7.4: Percentage of reported true-positives – with respect to all reads – with a certain MAPQ score or greater. Simulated human reads at 500nt and 1000nt.

Simulated paired-end

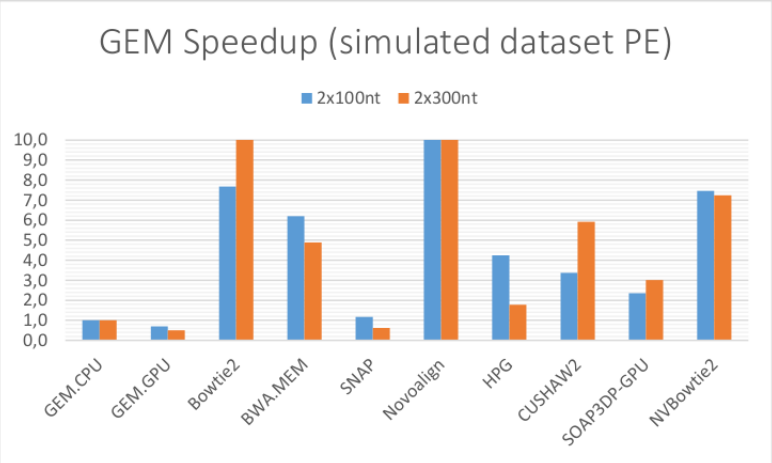


Figure 7.8: Comparative GEM CPU Speed-up for different simulated read length PE. All mappers run using defaults

For paired-end data, table 7.5 shows the timing results. Once again, GEM outperforms the rest of the mappers with the exception of SNAP in its fastest mode (i.e. h50 mode at 2x300nt). For most of the mappers, the paired-end mode adds a small overhead compared to single-end mapping.

If we observe the number of true positives found by each mapper, we find that most mappers increase their results using paired-end data. Despite GEM obtains very high results, mappers like HPG or SOAP3DP-GPU find additional true positives (i.e. about a 1% more). In contrast, SNAP and CUSHAW2 fail to find a comparable number of true positives in the paired-end mode.

As the ROC curves show (figure 7.9), GEM achieves one of the best ACC to FDR ratios; comparable to those of Novoalign and BWA-MEM. Other mappers like NvBowtie, SNAP, HPG, and SOAP3DP-GPU report significantly more false positives. In this case, Novoalign followed by BWA-MEM produce the best ROC curve.

Wall-clock time	2x100nt	2x300nt
GEM.fast	505	621
GEM.sensitive	1221	1312
GEM.fast.GPU	351	423
GEM.sensitive.GPU	1072	1102
Bowtie2.veryFast	2796	6903
Bowtie2.default	3882	7584
Bowtie2.verySensitive	6950	11801
BWA.MEM.default	3131	3040
BWA.MEM.c500	3132	3040
BWA.MEM.c1000	3439	5274
BWA.MEM.c16000	7360	5165
BWA.MEM.c20000	13015	9124
SNAP.h50	630	472
SNAP.default	589	384
SNAP.h1000	1929	3180
SNAP.h2000	2029	3576
Novoalign	12149	178403
HPG.fast	2062	854
HPG.default	2143	1102
HPG.sensitive	2142	1478
CUSHAW2.default	1700	3675
CUSHAW2.sensitive	1683	3708
SOAP3DP-GPU	1190	1872
NVBowtie2.default	3768	4501
NVBowtie2.veryFast	2764	2981
NVBowtie2.verySensitive	5619	8023

Table 7.5: Running time in seconds for simulated paired-end datasets

True positives	2x100nt	2x300nt
GEM.fast	98.38	99.64
GEM.sensitive	98.97	99.69
Bowtie2.veryFast	97.48	96.95
Bowtie2.default	97.72	97.29
Bowtie2.verySensitive	97.78	97.35
BWA.MEM.default	98.41	99.08
BWA.MEM.c500	98.41	99.08
BWA.MEM.c1000	98.41	99.08
BWA.MEM.c16000	98.41	99.08
BWA.MEM.c20000	98.41	99.08
SNAP.h50	96.47	97.73
SNAP.default	96.92	97.88
SNAP.h1000	97.85	98.14
SNAP.h2000	97.85	98.14
Novoalign	96.80	98.06
HPG.fast	99.61	99.06
HPG.default	99.61	99.06
HPG.sensitive	99.61	99.06
CUSHAW2.default	96.20	97.80
CUSHAW2.sensitive.K40	96.31	97.88
SOAP3DP-GPU	99.39	99.40
NVBowtie2.default	97.70	98.24
NVBowtie2.veryFast	97.72	98.27
NVBowtie2.verySensitive	97.71	98.26

Table 7.6: Percentage of true positives found for simulated paired-end datasets

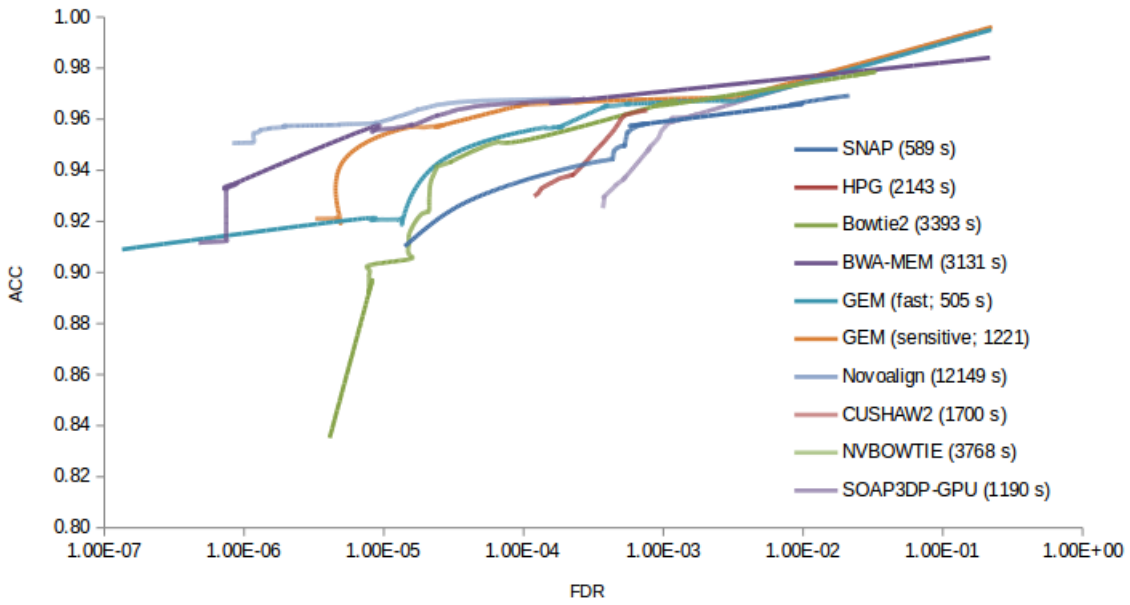


Figure 7.9: ROC curves for simulated PE datasets (default parameters)

7.4.2 Benchmark on real data

Next, a benchmark on real data was preformed. Table 7.7 shows the timing results. In short, GEM depicts better times than the rest of the mappers. As before, GEM-GPU increases the performance of the CPU mapper by a speed-up of 2x. Compared to Novoalign – one of the most accurate mappers available –, GEM obtains a speedup of 150x. Only SNAP in its fastest mode and SOAP3DP-GPU obtains similar performance results.

In terms of scalability with increasing read lengths, note that Bowtie2 and Novoaling show difficulties scaling with read length when mapping datasets like Moleculo. As opposed to SNAP which strongly depends on the mapping mode and seems to have better performance with longer reads. However, as table 7.8 shows, SNAP fails to map the majority of these long reads. As for GPU mappers, nvBowtie2 and SOAP3dp have problems scaling with longer reads, being unable to map very long reads like those of Moleculo. However, they perform very well with shorter reads (e.g. nvBowtie2 is almost 2x faster than its CPU homologous).

Wall-clock time	HiSeq.se	MiSeq.se	Ion.se	Moleculo	HiSeq.pe	MiSeq.pe
GEM.fast	423	562	467	711	620	626
GEM.sensitive	774	2613	2793	744	1552	2747
GEM.fast.GPU	211	260	499	487	305	285
GEM.sensitive.GPU	501	590	2861	505	410	1221
Bowtie2.veryFast	1771	1466	1311	134251	2972	3919
Bowtie2.default	3469	3162	3667	380706	4203	5854
Bowtie2.verySensitive	7310	6829	7146	794802	7263	7937
BWA.MEM.default	2889	3766	3461	2304	3389	4006
BWA.MEM.c500	2876	3766	3451	2295	3394	4007
BWA.MEM.c1000	3012	3867	3541	2308	3552	4088
BWA.MEM.c16000	3415	3921	3733	2327	3962	4170
BWA.MEM.c20000	4877	6897	10432	3120	5542	7318
SNAP.h50	811	1126	1103	547	1126	923
SNAP.default	3610	1617	4762	975	1617	3470
SNAP.h1000	6485	5553	4098	1436	5553	14770
SNAP.h2000	10268	5798	6100	2509	5798	14870
Novoalign	65072	45296	1464887	51932	190661	220322
HPG.fast	2198	1058	886	366	n/a	974
HPG.default	2278	1230	999	325	n/a	1172
HPG.sensitive	2257	1471	1019	353	n/a	1448
CUSHAW2.default	1582	1645	194	310	1645	4180
CUSHAW2.sensitive	1550	1603	193	318	1603	52786
SOAP3DP-GPU	503	1401	n/a	n/a	857	2408
NVBowtie2.default	1360	1156	n/a	n/a	3917	2287
NVBowtie2.veryFast	1034	716	n/a	n/a	2858	1622
NVBowtie2.verySensitive	2165	2279	n/a	n/a	5721	3752

Table 7.7: Running time in second for real datasets

Note that the majority of mappers take slightly more time on real data compared to simulated data – for the same amount of reads. As an actual fact, a small percentage of the real sequences don't map to the genome – due to potential errors during sequencing or perhaps contamination. In those cases, mapper would choose to explore matches with high error rate or switch to local

alignment algorithms in order to map chunks of those sequences. The latter option is significantly more computationally expensive, and not always guaranteed to lead to better results. For that reason, some mappers could spend an extra time compared to the results for simulated data.

Mapped reads	HiSeq.se	MiSeq.se	Ion.se	Moleculo	HiSeq.pe	MiSeq.pe
GEM.fast	96.80	98.70	57.80	99.66	95.89	97.73
GEM.sensitive	97.60	98.70	57.80	99.72	96.78	97.48
Bowtie2.veryFast	96.03	60.79	28.32	99.11	96.03	98.16
Bowtie2.default	97.00	62.23	29.49	99.25	97.00	98.89
Bowtie2.verySensitive	97.28	62.54	29.71	98.14	97.28	98.05
BWA.MEM.default	99.40	98.87	57.42	100.00	98.32	98.41
BWA.MEM.c500	99.40	98.87	57.42	100.00	98.32	98.41
BWA.MEM.c1000	99.40	98.87	57.42	100.00	98.32	98.41
BWA.MEM.c16000	99.40	98.87	57.42	100.00	98.32	98.41
BWA.MEM.c20000	99.44	98.93	57.70	100.00	98.34	98.52
SNAP.h50	99.64	33.46	24.20	81.81	98.50	20.82
SNAP.default	99.64	33.48	24.20	81.86	99.99	20.85
SNAP.h1000	99.64	33.48	24.20	81.89	99.98	20.91
SNAP.h2000	99.64	33.48	24.20	81.90	99.00	20.91
Novoalign	99.02	96.03	100.00	99.02	89.15	97.02
HPG.fast	99.14	94.32	39.51	99.42	n/a	94.12
HPG.default	99.14	94.54	39.60	99.44	n/a	95.04
HPG.sensitive	99.14	94.23	39.61	99.43	n/a	94.70
CUSHAW2.default	96.63	52.84	11.76	99.29	96.58	54.97
CUSHAW2.sensitive	98.83	97.92	51.31	99.98	98.61	99.06
SOAP3DP-GPU	97.16	85.69	n/a	n/a	96.52	100.00
NVBowtie2.default	96.99	53.69	n/a	n/a	96.99	71.26
NVBowtie2.veryFast	97.22	53.97	n/a	n/a	97.22	71.34
NVBowtie2.verySensitive	97.26	54.02	n/a	n/a	97.26	71.35

Table 7.8: Percentage of mapped reads for real datasets

In terms of mapped reads (table 7.8), most of the mappers achieve similar percentages. However, for the MiSeq dataset, some mappers fail to map most of their reads (i.e. Bowtie2, nvBowtie2, and SNAP). In the case of paired-end data, it seems that SNAP, CUSHAW2, and nvBowtie2 have difficulties mapping pairs. In contrast, BWA-MEM stands out from the rest as it achieves almost 100% mapping with many datasets outputting local alignments.

7.4.3 Benchmark on non-model organism

It is widely accepted that many mapping tool base their algorithms on the results of experimental tests. Moreover, some mappers focus specifically on the human genome reference. Ultimately, there are cases in which developers strongly couple their tool focusing only in this genome and even generate statistical models based on an specific version of it [Zaharia et al., 2011]. Nevertheless, mapping against the human genome is by far not the only case of use. Mapping tools are expected to deliver the same performance and accuracy when working with non model organisms too.

For that reason, here we give more thorough overview of the capabilities of the GEM-mapper within other cases of use. We aim to evaluate the tool when mapping against a non model organism and assess that the benefits of the methods behind the GEM-mapper still deliver good results in these scenarios.

	SE			PE		
Rainbow trout	Time	% Mapped	Matches	Time	% Mapped	Matches
Bwa-mem (default)	465	100.00	24.92M	529	99.99	14.58M
GEM (fast)	98	99.98	32.78M	130	99.97	15.94M
GEM (sensitive)	169	100.00	34.42M	212	99.97	15.63M

Table 7.9: Results from mapping against a non model organism (Oncorhynchus mykiss)

First, we present the results from mapping against the genome of the rainbow trout (Oncorhynchus mykiss). In this case, we have limited the comparison to GEM and BWA-MEM – the most competitive mapper up to date. Table 7.9 shows the result of this benchmark. In short, we can see that both tools reach a high percentage of mapped read – being BWA-MEM slightly superior. However, GEM clearly delivers better performance (i.e 2.7x - 4.7x speedup). At the same time, GEM report more matches than BWA-MEM.

	SE			PE		
Wheat	Time	% Mapped	Matches	Time	% Mapped	Matches
Bwa-mem (default)	1221	100.00	32.99M	1613	99.99	22.28M
GEM (fast)	153	99.91	72.47M	280	99.84	25.90M
GEM (sensitive)	415	100.00	80.13M	546	99.93	23.12M

Table 7.10: Results from mapping against a highly repetitive and large reference (Triticum Aestivum)

Additionally, we presents the results from mapping against the genome of wheat (Triticum Aestivum). This hexaploid genome is highly repetitive. Despite its total size is estimated to be 17 Gbp long, current assemblies only contain about 10Gbs of scaffolds – like the version used for this tests. Nevertheless, this is comparatively larger than the human genome and can potentially affect mapping tools performance (e.g. increasing memory penalties). Table 7.10 summarises the result of the benchmark. As before, both tools reach a high percentage of mapped read. However, both take longer than the previous benchmark to map the same amount of reads – with the same length. Note that the times of BWA-MEM have increased by 3x compared to those mapping against the Trout (2x in the case of GEM). As we can see, increasing the index size has less impact on the performance of GEM. Of equal importance is the number of matches reported. Note that GEM reports furthermore matches than BWA-MEM for the single-end. This highlights the repetitive nature of the genome and the sensitivity of mapping tools to it. Moreover, we can see that in the paired-end mode, BWA-MEM and GEM report similar number of matches as the paired reads constrain better the number of feasible matches, and the potential number of conflicts.

Chapter 8

Conclusions and contributions

In summary, this thesis has presented both (1) methodological contributions towards alignment quality assessment, and (2) effective techniques to perform practical sequence alignments in the context of bioinformatics and HTS data analysis. The contributions described here could ultimately be applied to most parts and modules of any modern sequence mapper.

From a methodological standpoint, this thesis offers a comprehensive analysis of *mapping quality assessment* and a practical tool – the GEM-tools – for quality analysis of mapping results (section 3.3). In particular, it analyses the potential sources of *mapping conflicts* and the properties of the reference genomes that can result in low-quality alignments, offering a well defined model to classify and compare mapping results (section 3.3.2). The proposed methodology is based on δ-*strata* groups (section III). It allows sequence mappers to distinguish potentially ambiguous reads, thus empirically estimating the expected accuracy error for each group. Additionally, this thesis presents a detailed analysis of the practical limits encountered while aligning sequencing reads, which is based on *genome mappability* – originally introduced in [Derrien et al., 2012] – (section 3.3.3). Mappability turns out to be a fundamental and recurrent concept when analysing filtering algorithms in the context of approximate string matching.

In regard to data structures used for index generation, this thesis offers a contribution centred on how to practically implement FM-indexes, and how to improve performance of algorithms based on such indexes. To that end, our work introduces a thorough analysis of a variety of *practical FM-Index design techniques* and *space-efficient layouts* (section 4.3). In the same spirit, it details many of the trade-offs between different approaches and how they impact the performance of downstream mapping tools. In addition, this work presents novel techniques to improve the performance of approximate search algorithms based on the FM-Index. These include *efficient FM-Index operators* (section 4.3.3) tailored for specific search cases and a practical *FM-Index lookup table* (section 4.3.4) which allows algorithms to skip a sizeable number of search steps. Finally, this work proposes modifications to the input reference in order to allow faster and easier alignment in application-specific situations – such as the *homopolymer compacted index* (section 4.4.2) for technologies prone to homopolymer errors during sequencing.

From an algorithmic standpoint, this thesis offers an important contribution to filtering algorithms for approximate string matching. Firstly this work contributes, alongside a novel dynamic partition filtering algorithm (published in [Marco-Sola et al., 2012]): the *adaptive pattern partitioning (APP)* (section 5.4.3). This greedy technique approximates the theoretical optimum partition while running in a time proportional to the length of the pattern. As opposed to widely used dynamic partition algorithms – like MEMs or periodical seeding techniques –, APP can formally guarantee a search depth that allows complete searches. Second, we offer a new *unified vision of*

filtering algorithms (section 5.1.1) to allow *chaining filters* (section 5.5) and exploit their properties towards better filtration ratios, efficiency and scalability. Moreover, we present a framework to *embed distant metrics* (section 5.2.4) and apply these chaining techniques to other metrics rather than editing distance. This result allows mapping tools based on biological metrics to benefit from the performance of deeply improved algorithms, tailored to simpler distance metrics. In this way, we reinforce the idea of progressive filtering by *ranking and deriving bounds* on the candidates as a *cascade of filtering algorithms* are applied (section 5.5.3). Finally, we present a technique to compose restrictions from different filtering partitions and prune incrementally deeper searches (i.e. *vertical chaining filters*, section 5.5.2). In this manner, we allow *progressive searches* increasing the error rate and avoiding computations of previous steps.

Additionally, this thesis presents neighbourhood search algorithms as complementary methods to filtering when further accuracy is required. Specifically, we extend bidirectional neighbourhood searches to the Levenshtein distance, and propose efficient pruning techniques to reduce the search spawn of these algorithms (section 6.3). Furthermore, we propose novel hybrid approaches (section 6.4) in order to combine filtering and neighbourhood search algorithms. We experimentally quantify the benefits of the methods proposed and how they successfully scale with long sequence inputs (section 6.5).

In the last chapter, this thesis presents its most notable and practical contribution in the form of a full-fledged production-ready sequence mapper: *the GEM-mapper* in its third iteration (section 7.2). Our mapping tool consistently outperforms other mappers over a wide range of parameters. At the same time, it achieves a comparable – if not better – quality than other leading mappers (section). Furthermore, the *GPU mapper implementation* (section 7.3) achieves a further speedup of 2x compared to to the CPU mapper. This enables even faster searches generating exactly the same results as the regular mapper – and achieving the same quality.

Besides all of its theoretical advances, perhaps one of the most relevant contributions of this thesis is to offer an efficient tool that can be employed by other members of the research community in order to generate high quality results in their own research. In this spirit, these two mappers – GEM CPU and GPU – have been successfully deployed within several data analysis pipelines and used within many real genome studies [Lappalainen et al., 2013; Liao et al., 2014; Suez et al., 2014; Thaiss et al., 2014; Balbás-Martínez et al., 2013; Letourneau et al., 2014].

However, it would be naive to consider this work completed. In this ever-changing field, we must confront the challenges arising from new developments in biotechnology. We are well aware that in short time we will probably have to process orders of magnitude more data with different characteristics. More specifically, we wonder whether our algorithmic approaches will remain suitable when sequence length surpasses more than a few kilobases. In the same way, we are well aware that a single linear reference will not suffice in the future. We strongly believe that mappers will naturally evolve to map sequences against populations of genomes and their multiple variants. In the future we hope to continue maintaining and improving the GEM-mapper so as to enlarge its community of users. Ultimately, we look forward to the development of new features that can tackle existing and new problems in the field of sequencing technologies and bioinfomatics research.

8.1 Journal publications

S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, 9(12):1185–1188, 2012

S. Marco-Sola and P. Ribeca. Efficient alignment of illumina-like high-throughput sequencing reads with the genomic multi-tool (gem) mapper. *Current Protocols in Bioinformatics*, pages 11–13, 2015

T. Derrien, J. Estellé, S. M. Sola, D. G. Knowles, E. Raineri, R. Guigó, and P. Ribeca. Fast computation and applications of genome mappability. *PloS one*, 7(1):e30377, 2012

A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Boosting the fm-index on the gpu: effective techniques to mitigate random memory access. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 12(5):1048–1059, 2015

A. Chacón, S. M. Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Fm-index on gpu: a cooperative scheme to reduce memory footprint. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 1–9. IEEE, 2014b

A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112. ACM, 2014a

S. Laurie, M. Fernandez-Callejo, S. Marco-Sola, J.-R. Trotta, J. Camps, A. Chacón, A. Espinosa, M. Gut, I. Gut, S. Heath, et al. From wet-lab to variations: Concordance and speed of bioinformatics pipelines for whole genome and whole exome sequencing. *Human Mutation*, 37(12):1263–1271, 2016

B. Rodríguez-Martín, E. Palumbo, S. Marco-Sola, T. Griebel, P. Ribeca, G. Alonso, A. Rastrojo, B. Aguado, R. Guigó, and S. Djebali. Chimpipe: accurate detection of fusion genes and transcription-induced chimeras from rna-seq data. *BMC Genomics*, 18(1):7, 2017

Bibliography

A. Ahmadi, A. Behm, N. Honnalli, C. Li, L. Weng, and X. Xie. Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic acids research*, 40(6):e41–e41, 2012.

D. Aird, M. G. Ross, W.-S. Chen, M. Danielsson, T. Fennell, C. Russ, D. B. Jaffe, C. Nusbaum, and A. Gnirke. Analyzing and minimizing pcr amplification bias in illumina sequencing libraries. *Genome biology*, 12(2):1, 2011.

C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061–1067, 2009.

B. Alpern, L. Carter, and K. Su Gatlin. Microparallelism and high-performance protein matching. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 24. ACM, 1995.

A. Apostolico. The myriad virtues of subword trees. In *Combinatorial algorithms on words*, pages 85–96. Springer, 1985.

Baeza-Yates and R. G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

J. A. Bailey and E. E. Eichler. Primate segmental duplications: crucibles of evolution, diversity and disease. *Nature Reviews Genetics*, 7(7):552–564, 2006.

M. Baker. De novo genome assembly: what every biologist should know. *Nature methods*, 9(4): 333, 2012.

C. Balbás-Martínez, A. Sagrera, E. Carrillo-de Santa-Pau, J. Earl, M. Márquez, M. Vazquez, E. Lapi, F. Castro-Giner, S. Beltran, M. Bayés, et al. Recurrent inactivation of stag2 in bladder cancer is not associated with aneuploidy. *Nature genetics*, 45(12):1464–1469, 2013.

A. K. Bansal and T. E. Meyer. Evolutionary analysis by whole-genome comparisons. *Journal of Bacteriology*, 184(8):2260–2272, 2002.

A. F. Bardet, Q. He, J. Zeitlinger, and A. Stark. A computational pipeline for comparative chip-seq analyses. *Nature protocols*, 7(1):45–61, 2012.

J. M. Berg, J. L. Tymoczko, and L. Stryer. Biochemistry. *The Citric Acid Cycle*, 5, 2006.

C. Berthelot, F. Brunet, D. Chalopin, A. Juanchich, M. Bernard, B. Noël, P. Bento, C. Da Silva, K. Labadie, A. Alberti, et al. The rainbow trout genome provides novel insights into evolution after whole-genome duplication in vertebrates. *Nature communications*, 5, 2014.

S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta informaticae*, 56(1-2):51–70, 2003.

S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (quasar). In *Proceedings of the third annual international conference on Computational molecular biology*, pages 77–83. ACM, 1999.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

A. Chacón, J. C. Moure, A. Espinosa, and P. Hernández. n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18:70–79, 2013.

A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112. ACM, 2014a.

A. Chacón, S. M. Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Fm-index on gpu: a cooperative scheme to reduce memory footprint. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 1–9. IEEE, 2014b.

A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Boosting the fm-index on the gpu: effective techniques to mitigate random memory access. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 12(5):1048–1059, 2015.

C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.

M. Chimani and K. Klein. Algorithm engineering: Concepts and practice. In *Experimental methods for the analysis of optimization algorithms*, pages 131–158. Springer, 2010.

P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38 (6):1767–1771, 2010.

R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *International Colloquium on Automata, Languages, and Programming*, pages 358–369. Springer, 2006.

R. Cordaux and M. A. Batzer. The impact of retrotransposons on human genome evolution. *Nature Reviews Genetics*, 10(10):691–703, 2009.

T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

M. A. Covington. The number of distinct alignments of two strings. *Journal of Quantitative Linguistics*, 11(3):173–182, 2004.

F. Crick et al. Central dogma of molecular biology. *Nature*, 227(5258):561–563, 1970.

F. H. Crick. On protein synthesis. In *Symp Soc Exp Biol*, volume 12, page 8, 1958.

M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno. Shrimp2: sensitive yet practical short read mapping. *Bioinformatics*, 27(7):1011–1012, 2011.

D. Deamer, M. Akeson, and D. Branton. Three decades of nanopore sequencing. *Nature biotechnology*, 34(5):518–524, 2016.

A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

A. L. Delcher, S. L. Salzberg, and A. M. Phillippy. Using mummer to identify similar regions in large sequence sets. *Current Protocols in Bioinformatics*, pages 10–3, 2003.

M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, et al. A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature genetics*, 43(5):491–498, 2011.

T. Derrien, J. Estellé, S. M. Sola, D. G. Knowles, E. Raineri, R. Guigó, and P. Ribeca. Fast computation and applications of genome mappability. *PloS one*, 7(1):e30377, 2012.

J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. Substantial biases in ultra-short read data sets from high-throughput dna sequencing. *Nucleic acids research*, 36(16):e105–e105, 2008.

E. K. Donald. The art of computer programming. *Sorting and searching*, 3:426–458, 1999.

M. Eisenstein. Big data: The power of petabytes. *Nature*, 527(7576):S2–S4, 2015.

B. Ewing and P. Green. Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome research*, 8(3):186–194, 1998.

M. Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13:12, 2009.

L. Feuk, A. R. Carson, and S. W. Scherer. Structural variation in the human genome. *Nature Reviews Genetics*, 7(2):85–97, 2006.

N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, page bts605, 2012.

J. L. Freeman, G. H. Perry, L. Feuk, R. Redon, S. A. McCarroll, D. M. Altshuler, H. Aburatani, K. W. Jones, C. Tyler-Smith, M. E. Hurles, et al. Copy number variation: new insights in genome diversity. *Genome research*, 16(8):949–961, 2006.

U. H. Frey, H. S. Bachmann, J. Peters, and W. Siffert. Pcr-amplification of gc-rich regions:'slowdown pcr'. *Nature protocols*, 3(8):1312–1317, 2008.

Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107–118, 1989.

A. S. Gargis, L. Kalman, D. P. Bick, C. Da Silva, D. P. Dimmock, B. H. Funke, S. Gowrisankar, M. R. Hegde, S. Kulkarni, C. E. Mason, et al. Good laboratory practice for clinical next-generation sequencing informatics pipelines. *Nature biotechnology*, 33(7):689–693, 2015.

S. Goodwin, J. D. McPherson, and W. R. McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, 2016.

D. J. Griffiths. Endogenous retroviruses in the human genome sequence. *Genome biology*, 2(6):1, 2001.

R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

Y. Guo, F. Ye, Q. Sheng, T. Clark, and D. C. Samuels. Three-stage quality control strategies for dna re-sequencing data. *Briefings in bioinformatics*, page bbt069, 2013.

Y. Guo, X. Ding, Y. Shen, G. J. Lyon, and K. Wang. Seqmule: automated pipeline for analysis of human exome/genome sequencing data. *Scientific reports*, 5, 2015.

D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsfast: a cache-oblivious algorithm for short-read mapping. *Nature methods*, 7(8):576–577, 2010.

P. A. Hall and G. R. Dowling. Approximate string matching. *ACM computing surveys (CSUR)*, 12(4):381–402, 1980.

R. W. Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2): 147–160, 1950.

E. C. Hayden. Genome researchers raise alarm over big data. *Nature*, 2015.

S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

S. Henikoff and J. G. Henikoff. Performance evaluation of amino acid substitution matrices. *Proteins: Structure, Function, and Bioinformatics*, 17(1):49–61, 1993.

C. Hercus. Novoalign. *Selangor: Novocraft Technologies*, 2012.

P. M. Higgins. Burrows–wheeler transformations and de bruijn words. *Theoretical Computer Science*, 457:128–136, 2012.

M. Holtgrewe. Mason–a read simulator for second generation sequencing data. *Technical Report FU Berlin*, 2010.

M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert. A novel and well-defined benchmarking method for second generation read mapping. *BMC bioinformatics*, 12(1):210, 2011.

N. Homer, B. Merriman, and S. F. Nelson. Bfast: an alignment tool for large scale genome resequencing. *PloS one*, 4(11):e7767, 2009.

W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *ALENEX/ANALC*, pages 31–38, 2004.

W. Huang, L. Li, J. R. Myers, and G. T. Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

S. Hwang, E. Kim, I. Lee, and E. M. Marcotte. Systematic comparison of variant calling pipelines using gold standard personal exome variants. *Scientific reports*, 5, 2015.

H. Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nord. J. Comput.*, 10(1):29–39, 2003.

H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 203–224. Springer, 2002.

P. Jokinen and E. Ukkonen. Two algorithms for approxmate string matching in static texts. In *International Symposium on Mathematical Foundations of Computer Science*, pages 240–248. Springer, 1991.

P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software: Practice and Experience*, 26(12):1439–1458, 1996.

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer, 2003.

H. H. Kazazian. Mobile elements: drivers of genome evolution. *science*, 303(5664):1626–1632, 2004.

B. Kehr, D. Weese, and K. Reinert. Stellar: fast and exact local alignments. *BMC bioinformatics*, 12(9):S15, 2011.

M. Kronrod, V. Arlazarov, E. Dinic, and I. Faradzev. On economic construction of the transitive closure of a direct graph. In *Sov. Math (Doklady)*, volume 11, pages 1209–1210, 1970.

G. Kucherov, K. Salikhov, and D. Tsur. Approximate string matching using a bidirectional index. *Theoretical Computer Science*, 638:145–158, 2016.

K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439, 1992.

S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29 (13):1149–71, 1999.

T. W. Lam, W.-K. Sung, S.-L. Tam, C.-K. Wong, and S.-M. Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24(6):791–797, 2008.

T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional bwt. In *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on*, pages 31–36. IEEE, 2009.

G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of algorithms*, 10(2):157–169, 1989.

E. S. Lander and M. S. Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–239, 1988.

E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

B. Langmead. Aligning short sequencing reads with bowtie. *Current protocols in bioinformatics*, pages 11–7, 2010.

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

B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

T. Lappalainen, M. Sammeth, M. R. Friedländer, P. AC't Hoen, J. Monlong, M. A. Rivas, M. Gonzalez-Porta, N. Kurbatova, T. Griebel, P. G. Ferreira, et al. Transcriptome and genome sequencing uncovers functional variation in humans. *Nature*, 501(7468):506–511, 2013.

A. H. Laszlo, I. M. Derrington, B. C. Ross, H. Brinkerhoff, A. Adey, I. C. Nova, J. M. Craig, K. W. Langford, J. M. Samson, R. Daza, et al. Decoding long nanopore sequencing reads of natural dna. *Nature biotechnology*, 32(8):829–833, 2014.

S. Laurie, M. Fernandez-Callejo, S. Marco-Sola, J.-R. Trotta, J. Camps, A. Chacón, A. Espinosa, M. Gut, I. Gut, S. Heath, et al. From wet-lab to variations: Concordance and speed of bioinformatics pipelines for whole genome and whole exome sequencing. *Human Mutation*, 37(12): 1263–1271, 2016.

T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.

H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proceedings of the 33rd international conference on Very large data bases*, pages 195–206. VLDB Endowment, 2007.

W.-P. Lee, M. P. Stromberg, A. Ward, C. Stewart, E. P. Garrison, and G. T. Marth. Mosaik: a hash-based algorithm for accurate next-generation sequencing short-read mapping. *PloS one*, 9 (3):e90581, 2014.

A. Letourneau, F. A. Santoni, X. Bonilla, M. R. Sailani, D. Gonzalez, J. Kind, C. Chevalier, R. Thurman, R. S. Sandstrom, Y. Hibaoui, et al. Domains of genome-wide gene expression dysregulation in down/'s syndrome. *Nature*, 508(7496):345–350, 2014.

M. J. Levene, J. Korlach, S. W. Turner, M. Foquet, H. G. Craighead, and W. W. Webb. Zero-mode waveguides for single-molecule analysis at high concentrations. *Science*, 299(5607):682–686, 2003.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

H. Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

H. Li. Towards better understanding of artifacts in variant calling from high-coverage samples. *Bioinformatics*, page btu356, 2014.

H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

H. Li and R. Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008a.

H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16): 2078–2079, 2009.

R. Li, Y. Li, K. Kristiansen, and J. Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008b.

Y. Liao, G. K. Smyth, and W. Shi. featurecounts: an efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics*, 30(7):923–930, 2014.

Y. Liu and B. Schmidt. Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing. *IEEE Design & Test*, 31(1):31–39, 2014.

N. J. Loman, R. V. Misra, T. J. Dallman, C. Constantinidou, S. E. Gharbia, J. Wain, and M. J. Pallen. Performance comparison of benchtop high-throughput sequencing platforms. *Nature biotechnology*, 30(5):434–439, 2012.

R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, et al. Soap3-dp: fast, accurate and sensitive gpu-based short read aligner. *PloS one*, 8(5):e65632, 2013.

C. A. Mack. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24 (2):202–207, 2011.

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

S. Marco-Sola and P. Ribeca. Efficient alignment of illumina-like high-throughput sequencing reads with the genomic multi-tool (gem) mapper. *Current Protocols in Bioinformatics*, pages 11–13, 2015.

S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, 9(12):1185–1188, 2012.

E. R. Mardis. A decade/'s perspective on dna sequencing technology. *Nature*, 470(7333):198–203, 2011.

V. Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162. ACM, 2004.

A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. CRC Press, 2004.

D. W. Mount. Comparison of the pam and blosum amino acid substitution matrices. *Cold Spring Harbor Protocols*, 2008(6):pdb–ip59, 2008.

E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4): 345–374, 1994.

G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

G. Myers. What's behind blast. In *Models and Algorithms for Genome Evolution*, pages 3–15. Springer, 2013.

G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33 (1):31–88, 2001.

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

G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

K. E. Nelson, R. A. Clayton, S. R. Gill, M. L. Gwinn, R. J. Dodson, D. H. Haft, E. K. Hickey, J. D. Peterson, W. C. Nelson, K. A. Ketchum, et al. Evidence for lateral gene transfer between archaea and bacteria from genome sequence of thermotoga maritima. *Nature*, 399(6734):323–329, 1999.

S. B. Ng, E. H. Turner, P. D. Robertson, S. D. Flygare, A. W. Bigham, C. Lee, T. Shaffer, M. Wong, A. Bhattacharjee, E. E. Eichler, et al. Targeted capture and massively parallel sequencing of 12 human exomes. *Nature*, 461(7261):272–276, 2009.

S. Ohno. Repetition as the essence of life on this earth: music and genes. In *Modern Trends in Human Leukemia VII*, pages 511–519. Springer, 1987.

S. Ohno. Patterns in genome evolution. *Current opinion in genetics & development*, 3(6):911–914, 1993.

R. K. Patel and M. Jain. Ngs qc toolkit: a toolkit for quality control of next generation sequencing data. *PloS one*, 7(2):e30619, 2012.

D. Pratas, A. J. Pinho, and J. M. Rodrigues. Xs: a fastq read simulator. *BMC research notes*, 7 (1):40, 2014.

K. Prüfer, K. Munch, I. Hellmann, K. Akagi, J. R. Miller, B. Walenz, S. Koren, G. Sutton, C. Kodira, R. Winer, et al. The bonobo genome compared with the chimpanzee and human genomes. *Nature*, 486(7404):527–531, 2012.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4, 2007.

K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ε-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006.

J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. *Molecular cell*, 58(4):586–597, 2015.

A. Rhoads and K. F. Au. Pacbio sequencing and its applications. *Genomics, proteomics & bioinformatics*, 13(5):278–289, 2015.

P. Ribeca and G. Valiente. Computational challenges of sequence classification in microbiomic data. *Briefings in bioinformatics*, page bbr019, 2011.

J. C. Roach, C. Boysen, K. Wang, and L. Hood. Pairwise end sequencing: a unified approach to genomic mapping and sequencing. *Genomics*, 26(2):345–353, 1995.

B. Rodríguez-Martín, E. Palumbo, S. Marco-Sola, T. Griebel, P. Ribeca, G. Alonso, A. Rastrojo, B. Aguado, R. Guigó, and S. Djebali. Chimpipe: accurate detection of fusion genes and transcription-induced chimeras from rna-seq data. *BMC Genomics*, 18(1):7, 2017.

T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011.

T. Rognes and E. Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

Ł. Roguski and P. Ribeca. Cargo: effective format-free compressed storage of genomic information. *Nucleic acids research*, page gkw318, 2016.

M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe. Characterizing and measuring bias in sequence data. *Genome biology*, 14(5):1, 2013.

N. Rusk. Torrents of sequence. *Nature Methods*, 8(1):44–44, 2011.

H. Sakai, K. Naito, E. Ogiso-Tanaka, Y. Takahashi, K. Iseki, C. Muto, K. Satou, K. Teruya, A. Shiroma, M. Shimoji, et al. The power of single molecule real-time sequencing technology in the de novo assembly of a eukaryotic genome. *Scientific reports*, 5, 2015.

J. Salavert, A. Tomás, J. Tárraga, I. Medina, J. Dopazo, and I. Blanquer. Fast inexact mapping using advanced tree exploration on backward search methods. *BMC bioinformatics*, 16(1):18, 2015.

P. Sanders. Algorithm engineering–an attempt at a definition. In *Efficient Algorithms*, pages 321–340. Springer, 2009.

F. Sanger, S. Nicklen, and A. R. Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

K. U. Schulz and S. Mihov. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85, 2002.

S. L. Schwartz and M. L. Farman. Systematic overrepresentation of dna termini and underrepresentation of subterminal regions among sequencing templates prepared from hydrodynamically sheared linear dna molecules. *BMC genomics*, 11(1):1, 2010.

P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

E. Siragusa. *Approximate string matching for high-throughput sequencing.* PhD thesis, Freie Universität Berlin, 2015.

E. Siragusa, D. Weese, and K. Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic acids research*, 41(7):e78–e78, 2013.

J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *International Symposium on String Processing and Information Retrieval*, pages 164–175. Springer, 2008.

G. S. Slater and E. Birney. Automated generation of heuristics for biological sequence comparison. *BMC bioinformatics*, 6(1):31, 2005.

J. Suez, T. Korem, D. Zeevi, G. Zilberman-Schapira, C. A. Thaiss, O. Maza, D. Israeli, N. Zmora, S. Gilad, A. Weinberger, et al. Artificial sweeteners induce glucose intolerance by altering the gut microbiota. *Nature*, 514(7521):181–186, 2014.

E. Sutinen and J. Tarhio. Approximate string matching with ordered q-grams. *Nord. J. Comput.*, 11(4):321–343, 2004.

M. N. Szmulewicz, G. E. Novick, and R. J. Herrera. Effects of alu insertions on gene function. *Electrophoresis*, 19(8-9):1260–1264, 1998.

J. Tárraga, V. Arnau, H. Martínez, R. Moreno, D. Cazorla, J. Salavert-Torres, I. Blanquer-Espert, J. Dopazo, and I. Medina. Acceleration of short and long dna read mapping without loss of accuracy using suffix array. *Bioinformatics*, 30(23):3396–3398, 2014.

C. Tennakoon, R. W. Purbojati, and W.-K. Sung. Batmis: a fast algorithm for k-mismatch mapping. *Bioinformatics*, 28(16):2122–2128, 2012.

C. T. B. Tennakoon. *Fast and Accurate Mapping of Next Generation Sequencing Data*. PhD thesis, National University of Singapore, 2013.

C. A. Thaiss, D. Zeevi, M. Levy, G. Zilberman-Schapira, J. Suez, A. C. Tengeler, L. Abramson, M. N. Katz, T. Korem, N. Zmora, et al. Transkingdom control of microbiota diurnal oscillations promotes metabolic homeostasis. *Cell*, 159(3):514–529, 2014.

T. Thomas, J. Gilbert, and F. Meyer. Metagenomics-a guide from sampling to data analysis. *Microbial informatics and experimentation*, 2(1):1, 2012.

U. H. Trivedi, T. Cézard, S. Bridgett, A. Montazam, J. Nichols, M. Blaxter, and K. Gharbi. Quality control of next-generation sequencing data without a reference. *Frontiers in genetics*, 5: 111, 2014.

A. M. Turing. Biological sequences and the exact string matching problem. *Introduction to Computational Biology-Springer*, 2006.

E. H. Turner, C. Lee, S. B. Ng, D. A. Nickerson, and J. Shendure. Massively parallel exon capture and library-free resequencing across 16 genomes. *Nature methods*, 6(5):315–316, 2009.

E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985a.

E. Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985b.

E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, 92(1):191–211, 1992.

J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, et al. The sequence of the human genome. *science*, 291 (5507):1304–1351, 2001.

R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert. Razers—fast read mapping with sensitivity control. *Genome research*, 19(9):1646–1654, 2009.

D. Weese, M. Holtgrewe, and K. Reinert. Razers 3: faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599, 2012.

P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, 13(2):145–150, 1997.

S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10): 83–91, 1992.

W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan. Accelerating read mapping with fasthash. *BMC genomics*, 14(1):S13, 2013.

H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu. Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, 32(11):1632–1642, 2016.

X. Yang, S. P. Chockalingam, and S. Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66, 2013a.

X. Yang, D. Liu, F. Liu, J. Wu, J. Zou, X. Xiao, F. Zhao, and B. Zhu. Htqc: a fast quality control toolkit for illumina sequencing data. *BMC bioinformatics*, 14(1):33, 2013b.

M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with snap. *arXiv preprint arXiv:1111.5572*, 2011.

L. Zhang, M. F. Miles, and K. D. Aldape. A model of molecular interactions on short oligonucleotide microarrays. *Nature biotechnology*, 21(7):818–821, 2003.