FACULTY OF ENGINEERING
AND ARCHITECTURE

# Approximate Sequence Alignment Using Search Schemes

**Luca Renders**

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

**Supervisors**
Prof. Jan Fostier, PhD - Prof. Kathleen Marchal, PhD
Department of Information Technology
Faculty of Engineering and Architecture, Ghent University

March 2025

GHENT
UNIVERSITY

ISBN 978-94-6355-961-4
NUR 980, 923
Wettelijk depot: D/2025/10.500/21

## Members of the Examination Board

### Chair

Prof. Filip De Turck, PhD, Ghent University

### Other members entitled to vote

Prof. Peter Dawyndt, PhD, Ghent University
Prof. Veerle Fack, PhD, Ghent University
Giles Miclotte, PhD, Ghent University
Prof. Sven Rahmann, PhD, Universität des Saarlandes, Germany

### Supervisors

Prof. Jan Fostier, PhD, Ghent University
Prof. Kathleen Marchal, PhD, Ghent University

# Dankwoord

In april 2020 besloot ik dat ik mijn masterproef verder wou uitwerken in een doctoraat. De pandemie had me doen inzien dat ik 'nog niet klaar was om dag te zeggen tegen de universiteit'. Ik contacteerde Jan - die al mijn promotor was tijdens de masterproef - die met enthousiasme instemde om mij in dit nieuwe project te begeleiden. Bedankt Jan, voor de talloze meetings en waardevolle feedback. Dankzij jouw intense begeleiding in het eerste jaar kon ik vlot aan het doctoraat starten. De kritische blik op mijn teksten heeft van mij een betere schrijver gemaakt. Al heeft de introductie van publiek toegankelijke large language models daar misschien ook iets mee te maken. Naast Jan kreeg ik nog een copromotor, Kathleen. Haar had ik al leren kennen tijdens de verdediging van mijn masterproef, waarin ze, na een uiteenzetting over cache-optimalisaties, oprecht vroeg "Wat is de cache?". Speciaal voor haar bevat deze thesis dan ook een sectie rond de cache, mét metafoor, zodat ook computer-science-leken kunnen volgen. Kathleens impact reikte natuurlijk verder dan dat. Bedankt Kathleen, om mee na te denken over allerhande use cases waarin onze software gebruikt kon worden. Dankzij jou kan deze thesis beginnen met een meeslepend voorbeeld en dat maakt een wereld van verschil.

I would also like to express my sincere gratitude to the members of the jury. Thank you to Chair prof. Filip De Turck, prof. Peter Dawyndt, prof. Veerle Fack, dr. Giles Miclotte and prof. Sven Rahmann, for your engaging discussions during the internal defense and for dedicating your valuable time and effort to reading this book. Your thoughtful remarks helped shape this work into an even more coherent whole. Ein besonderer Dank gilt Professor Rahmann für die inspirierende Zusammenarbeit, die in einer Publikation und einem Vortrag auf der RECOMB-Konferenz mündete.

Lockdowns en een gescheurde achillespees zorgden ervoor dat het even duurde voor ik mijn collega's goed kon leren kennen. Gelukkig werd dat daarna ruimschoots goedgemaakt, met spelletjesavonden, International Food Evenings, en uitstapjes naar het lichtfestival en de Gentse Feesten. Thank you to all the colleagues from the BioIT office: Aranka, David, Dries, Giles, Jan, Kathleen, Lore, Louise, Maarten, Marie, Marija, Pieter-Paul, Razgar, Simon and TaeWoo. Daarnaast wil ik ook graag het ondersteunend en administratief personeel van IDLab, Martine, Bernadette, Joke, Davinia, Karen, Nathalie, Joeri, Sai, Vicent en Sabrina, bedanken. An extra special note to Sai for her invaluable support, which extended far beyond her official duties. Thank you to the bar team at the International Food Evening, Diego, Gargya, Giulio, Maria, Marie-Sophie, and Sai, I had a blast.

Het leven van een doctoraatsstudent is gelukkig niet beperkt tot enkel het academische. Vrienden, hobby's en familie zorgden voor de nodige afleiding en motivatie. Specifiek bedankt aan de nalezers uit mijn omgeving, die onder andere hielpen in het wegwerken van mijn gênante spellingsfouten: Ellen, Ina, Lia, Liza, mama en Ruben. Ik zou graag de leden van handbalclub Don Bosco Gent en Amai Comedy Club - in het bijzonder de fijne maandaggroep - bedanken om mij in hun rangen te sluiten, en om mij het gevoel te geven dat ik toch íets kan, wanneer het op academisch vlak even tegenzat.

Hou u vast, beste lezer, dit wordt een hele lange paragraaf. Ik zou heel graag mijn vriendengroep, "de ir-puntjes", ook wel gekend als de luide groep rechts vooraan in auditorium A (Boeykens, 2025), uitgebreid bedanken. Zij hebben me mee door mijn studies en dit doctoraat gesleurd. Bedankt Ruben, voor de vele lange babbels, langs de Plateau, op een kot of op restaurant. In 2015 wist ik al dat het altijd een beetje thuiskomen is bij jou. Dankjewel Pieter-Jan, om vanaf dag één ontegensprekelijk jezelf te zijn, om uitermate vrijgevig je voorbeeldoefeningen en structuur te delen en om die structuur in de groep te brengen. Merci Simon, mijn allergrootste vriend, voor de vele knuffels, het onbevooroordeeld luisterend oor en je inspiratievolle doorzettingsvermogen. Danku, Karolien, om mijn blik op de wereld van TikTok te zijn en vooral om steeds je met hart en ziel te smijten voor het organiseren van activiteiten. Mariska, ook jij bedankt om mij steeds met engelengeduld te herinneren dat ik de poll nog niet had ingevuld. Eveneens bedankt om een voorbeeld te zijn in het nemen van rust en afzondering wanneer dat nodig is. Dankje, András, om me te laten zien hoe je met zoveel gratie je plek in de wereld vindt en inneemt, en om het improvuur in mij terug aan te wakkeren. Dankuwel, Mathias, om zoveel flauwe moppen te vertellen en vooral om te lachen met de mijne. Julie, dankjewel voor je eeuwige vrolijkheid en nog langerdurende geloof in al het goede van de mens. Danku, Leander, om zo sterk te zijn en je toch zo kwetsbaar te kunnen opstellen. Ook bedankt voor de fijne kijkdag in de Efteling. Dankje, Margje, om te tonen dat je je eigen pad kan en mag maken. Lieve Xavier, dankuwel om hier te zijn, om te luisteren, om je begrip en om ook bij tegenslag anderen op te beuren met je warmte.

Ook al kan ik voor geen meter zingen, voel ik toch een bijzondere affiniteit met de studenten- en alumnikoren van onze universiteit. Dankzij kookouder zijn en het scannen van tickets kunnen ook bijzondere banden opgebouwd worden. Specifiek bedankt aan Anne, Ewoud en Greet voor alle momenten samen en aan die eerste twee om, weliswaar in andere disciplines, doctoraatslotgenoten te zijn.

Natuurlijk wil ik eveneens graag mijn familie bedanken. Mijn ouders, en bij uitbreiding de rest van de familie, voor de steun en trotsheid die ik steeds heb mogen ervaren. Dankje, Liza en Nils, om mijn grote zus en schoonbroer te zijn, waar ik altijd op kan rekenen. Trix, dankjewel dat jij pas twee dagen ná mijn deadline op de wereld kwam, heel flink gedaan! Hilde, Dimitri, Lia, James en oma, danku voor het omarmen van deze onhandige computerwetenschapper, én de vele geïnteresseerde vragen. Ook bedankt aan zij die er niet meer zijn, maar er wel altijd waren.

Tot slot rest mij nog mijn grafisch ontwerper te bedanken. Ina, mijn lief, die bij me kwam wonen na de gescheurde achillespees en daarna nooit meer is weggegaan. Jouw hulp bij het ontwerp van alle figuren en presentaties was van onschatbare waarde — iets wat telkens opnieuw bevestigd werd door de complimenten die ik op conferenties mocht ontvangen. Esthetiek is slechts een deel van jouw invloed. Je nam niet alleen de tijd om mijn werk visueel helder te maken, maar ook om rust en balans in mijn leven te brengen. Je neemt enorm veel ruimte in mijn hart in, en je leerde me dat figuren – net als mensen – soms gewoon wat meer witruimte nodig hebben.

*Gent, februari 2025*
*Luca Renders*

# Table of Contents

# Nederlandstalige Samenvatting – Summary in Dutch –

Desoxyribonucleïnezuur (DNA) vormt de drager van genetische informatie die essentieel is voor het leven en bestaat uit sequenties van vier nucleotidenbasen — Adenine (A), Cytosine (C), Guanine (G) en Thymine (T) — die samen de bouwstenen vormen van alle organismen. Het menselijk genoom omvat ongeveer 3 miljard basenparen, georganiseerd in 23 chromosomenparen. Hoewel het genoom voor 99,9% identiek is tussen menselijke individuen, bevindt de immense genetische diversiteit zich in de resterende 0,1%. Deze kleine variaties beïnvloeden uiteenlopende cruciale biologische eigenschappen, zoals immuunreacties, neuronale functies, metabolische processen en de vatbaarheid voor diverse ziekten. Eén van de meest variabele gebieden in het genoom is de regio van het humaan leukocytenantigeen (HLA). Deze regio speelt een centrale rol in de werking van het immuunsysteem, door het lichaam in staat te stellen onderscheid te maken tussen het "lichaamseigene" en het "lichaamsvreemde". De enorme omvang en complexiteit van het genoom, gecombineerd met de variabiliteit van regio's zoals de HLA-regio, vormen echter aanzienlijke computationele uitdagingen voor een nauwkeurige genomische analyse.

Nauwkeurige sequentiealignering is fundamenteel voor genomische studies. Het maakt onder andere de identificatie van genetische varianten, genoomassemblage en toepassingen zoals HLA-typering, waarbij precisie van essentieel belang is, mogelijk. Heuristische aligneringsprogramma's, waaronder veel gebruikte programma's zoals BWA-MEM en Bowtie, bieden traditioneel hoge snelheidsprestaties door gebruik te maken van slimme algoritmes om snel kandidaataligneringen te identificeren. Alhoewel deze tools in veel standaardscenario's effectief zijn, rapporteren ze soms niet alle alternatieve, even waarschijnlijke aligneringen.

Deze beperking is bijzonder kritisch in situaties zoals HLA-typering, waar de hoge variabiliteit van allelen en kleine sequentieverschillen tussen deze allelen een uitgebreide *read*-alignering vereisen om nauwkeurige genotyperingsresultaten te garanderen. Als kandidaataligneringen ontbreken, kunnen DNA-fragmenten onvolledig worden toegewezen aan specifieke allelen. Dit kan leiden tot fouten of onnauwkeurigheden bij de uiteindelijke genotypering van het individu. Ook in pan-genomische studies, waarin meerdere nauwverwante genomen worden geanalyseerd, kan het over het hoofd zien van even waarschijnlijke aligneringen leiden tot het missen van belangrijke informatie en verkeerde conclusies over welke soort of stam er in het monster zit.

Verliesloze aligneringsmethoden pakken deze tekortkomingen aan door exhaustief alle mogelijke *matches* binnen een gegeven foutdrempel te rapporteren. Hoewel deze

methoden volledige en nauwkeurige resultaten garanderen, kampen ze traditioneel met trage verwerkingssnelheden, waardoor ze minder geschikt zijn voor grootschalige genomische toepassingen. Het overbruggen van deze kloof tussen de exhaustiviteit van verliesloze methoden en de efficiëntie van heuristiek-gebaseerde tools vereist de ontwikkeling van innovatieve computationele strategieën die zowel snel als geheugenefficiënt zijn.

Dit werk introduceert Columba (Latijn voor 'duif'), een geoptimaliseerd verliesloos aligneringsprogramma dat is ontworpen om deze uitdagingen te overwinnen door geavanceerde algoritmen en innovatieve methodologieën te integreren. Columba combineert bidirectionele FM-indexering met verfijnde zoekschema's. Hierdoor verbetert Columba de snelheid en schaalbaarheid van verliesloze alignering, terwijl een volledige dekking behouden blijft. De bidirectionele FM-index maakt efficiënte patroon*matching* mogelijk in zowel voorwaartse als achterwaartse richtingen, een essentiële eigenschap voor het effectief implementeren van flexibele zoekschema's. Zoekschema's bieden een gestructureerd kader om alle mogelijke *matches* binnen gedefinieerde foutdrempels te identificeren. In de literatuur zijn er ontworpen zoekschema's die foutdrempels tot $k \leq 4$ ondersteunen, terwijl voor willekeurige waarden van $k$ alleen minder efficiënte schema's (zoals die gebaseerd op het duivenhokprincipe) beschikbaar zijn. In dit werk introduceren we ook Hato (Japans voor 'duif'), een aanvullend hulpmiddel dat is ontwikkeld om zoekschema's te verfijnen en optimaal te ontwerpen voor hogere waarden van $k$. Hato maakt gebruik van *Integer Lineair Programming*(ILP) om wiskundig geoptimaliseerde zoekschema's te creëren. Met Hato hebben we optimale zoekschema's voor maximaal 7 toegestane fouten ontworpen. Bovendien kan Hato bestaande inefficiënte zoekschema's voor willekeurige waarden van $k$ verbeteren via toepassing van een gulzige heuristiek (*greedy heuristic*). Hiermee werden verbeterde zoekschema's tot $k = 13$ gepresenteerd. De met Hato ontworpen zoekschema's werden geïntegreerd in Columba. Door foutdrempels tot 13 te ondersteunen, zorgt Columba voor verbeterde prestaties en schaalbaarheid in complexe aligneringsscenario's.

Columba bereikt de hogere snelheidsprestaties door een combinatie van kerninnovaties die de computationele efficiëntie verbeteren, terwijl de aligneringscompleetheid behouden blijft. Dynamische partitionering van de patronen past de zoekprocedure aan aan de specifieke *read*- en referentie-inhoud., waardoor de zoekruimte en uitvoeringstijd aanzienlijk krimpen. Deze techniek is universeel toepasbaar op elk zoekschema.

Een hybride strategie combineert in-index *matching* met bit-parallele in-tekst verificatie. Afhankelijk van de verdeling van kandidaatvoorkomens wordt dynamisch de meest geschikte verwerkingsmethode gekozen. Deze aanpak minimaliseert de computationele overhead voor zeldzame voorkomens via in-text verificatie en gebruikt batchverwerking binnen de index voor frequente voorkomens. Dit leidt tot een meer dan tweevoudige reductie van de verwerkingstijd vergeleken met pure in-index methoden.

Verdere prestatieverbeteringen in Columba omvatten technieken voor redundantievermindering bij berekeningen van de bewerkingsafstand en geheugenoptimalisaties die zijn afgestemd op genomische aligneringstaken, zoals *interleaved* bitvectoren. Deze *interleaved* bitvectoren zijn specifiek ontworpen om cachefouten te verminderen door de geheugentoegangspatronen tijdens het aligneringsproces te optimaliseren, wat

de verwerkingstijd verder verbetert.

Bovendien verbetert de ondersteuning van Columba voor een *run-length* gecomprimeerde (RLC) bidirectionele index de geheugenefficiëntie voor sterk repetitieve referentiegenomen, waardoor Columba RLC bijzonder geschikt is voor pan-genomische toepassingen. Terwijl bidirectionele indexing vaak de geheugeneisen verhoogt, compenseert de RLC-implementatie van Columba dit door compressie.

Daarnaast ondersteunt Columba *paired-end* alignering, waarbij informatie over de beide uiteinden van een DNA-fragment wordt benut om aligneringen te filteren op basis van verwachte fragmentgroottebeperkingen. Door de ruimtelijke relatie tussen gepaarde *reads* te gebruiken, vermindert deze functie ambiguïteiten en verhoogt het de *mapping*-precisie. Met multithreading-mogelijkheden parallelliseert Columba efficiënt taken, waardoor effectieve verwerking van grootschalige datasets in *high-throughput sequencing* toepassingen mogelijk wordt.

In uitgebreide benchmarktests tegen meerdere *state-of-the-art* verliesloze tools zoals RazerS3, Yara en Bwolo, overtrof Columba deze tools aanzienlijk qua snelheid, met tijdsreducties tot 38 keer in complexe scenario's. Ondanks deze snelheidsvoordelen bleef de nauwkeurigheid zo goed als identiek met de bestaande methoden. Opmerkelijk genoeg was de prestatie ook concurrerend met heuristiek-gebaseerde tools zoals BWA-MEM en Bowtie2. Dit benadrukt de mogelijkheid om de langdurige prestatiekloof tussen verliesloze en verliesgevende methoden te overbruggen. De efficiëntie van Columba strekt zich uit tot zowel enkel-genoom als pan-genoom analyses, waardoor het een krachtig en veelzijdig hulpmiddel is voor een breed scala aan genomische studies.

Columba kan succesvol worden geïntegreerd in bioinformatica-pijplijnen in de praktijk, zoals blijkt uit de implementatie ervan in de OptiType-pijplijn voor HLA-genotypering. In deze praktijktoepassing verbeterde Columba de verwerkingstijd aanzienlijk, terwijl de hoge nauwkeurigheid behouden bleef, wat aantoont dat Columba in staat is om bestaande workflows naadloos en effectief te verbeteren.

Dit werk laat zien dat verliesloze aligneringsmethoden prestatieniveaus kunnen bereiken die vergelijkbaar zijn met op heuristiek gebaseerde tools, zonder te moeten inboeten op volledigheid. Door een brug te slaan tussen precisie en efficiëntie maakt Columba uitputtende en nauwkeurige sequentiealignering praktisch haalbaar voor diverse genomische aligneringsscenario's.

Concluderend transformeert dit werk verliesloze alignering van een traag en niche-proces naar een schaalbare en toegankelijke technologie, toepasbaar in een breed scala aan genomische analyses. Door gebruik te maken van dynamische partitionering, hybride verwerkingsstrategieën en geoptimaliseerde zoekschema's, biedt Columba onderzoekers en clinici een krachtige tool om subtiele genetische variaties op te sporen. Door uitputtende en nauwkeurige alignering toegankelijk en schaalbaar te maken, draagt dit onderzoek bij aan een dieper begrip van (menselijke) genetische diversiteit. Het vormt daarmee een nieuwe cruciale schakel in het proces naar betrouwbaardere diagnostiek, effectievere therapieën en grootschalige genomische analyses.

# Summary

Deoxyribonucleic acid (DNA) encodes the genetic information essential for life, consisting of sequences of four nucleotide bases—Adenine (A), Cytosine (C), Guanine (G), and Thymine (T), which together form the building blocks of all organisms. The human genome, spanning approximately 3 billion base pairs, is organized into 23 pairs of chromosomes and is 99.9% identical among humans, with the remaining 0.1% contributing to the immense diversity observed between individuals. This small variation influences a range of critical biological traits, including immune responses, neural function, metabolic processes, and susceptibility to a variety of diseases. Among these, the Human Leukocyte Antigen (HLA) region is one of the most variable areas in the genome, playing a pivotal role in immune system function by enabling the body to distinguish the "self" from the "non-self". However, the enormous size and complexity of the genome, coupled with the variability of regions like the HLA region, pose significant computational challenges for accurate genomic analysis.

Accurate sequence alignment is foundational to genomic studies, enabling the identification of genetic variants, genome assembly, and applications such as HLA typing, where precision is paramount. Heuristic-based alignment tools, including widely used programs like BWA-MEM and Bowtie, have traditionally provided high-speed performance by employing heuristics to quickly identify candidate alignments. While effective for many standard scenarios, these tools can fail to report other equally probable matches. This limitation is particularly critical in scenarios like HLA typing, where the high variability of alleles and small sequence differences between them demand comprehensive read alignment to ensure accurate genotyping outcomes. Missing candidate matches can result in incomplete collections of reads aligned to specific alleles, which can ultimately lead to ambiguities or inaccuracies in the final determination of an individual's genotype. Similarly, in pan-genomic studies, where multiple closely related genomes are analyzed as a reference set, overlooking equally likely alignments can obscure important information and lead to incorrect conclusions.

Lossless alignment methods address these shortcomings by exhaustively reporting all matches within a given error threshold, thereby ensuring complete and accurate alignment data for downstream analysis. However, these methods have historically been hindered by slow processing speeds, making them impractical for large-scale genomic applications that require fast processing. Bridging this gap between the exhaustiveness of lossless methods and the efficiency of heuristic-based tools requires the development of innovative computational strategies that are both fast and memory-efficient.

This work presents Columba (Latin for 'pigeon'), a highly optimized lossless align-

ment tool designed to address these challenges through the integration of advanced algorithms and novel methodologies. Columba leverages the bidirectional FM-index and refined search schemes to enhance the speed and scalability of lossless alignment while maintaining exhaustive and complete coverage. Bidirectional FM-indexing enables efficient pattern matching in both forward and reverse directions, an essential feature for implementing flexible search schemes effectively. Search schemes provide a structured framework for exploring all possible matches within defined error thresholds, ensuring comprehensive analysis of genomic sequences. Traditionally, search schemes have been designed to support error thresholds up to $k \leq 4$, with some less efficient schemes available for arbitrary values of $k$ (like those based on the pigeonhole principle). In this work, we also introduce Hato (Japanese for 'pigeon') a complementary tool developed to refine and optimally design search schemes, addressing these limitations. Hato utilizes an Integer Linear Programming (ILP) approach to create mathematically optimized search schemes. Using Hato, we have designed search schemes for up to 7 allowed errors. Furthermore, Hato can optimize existing inefficient search schemes for arbitrary values of $k$ by employing a greedy heuristic, thereby supporting thresholds as high as $k = 13$. The search schemes designed with Hato are integrated into Columba. By supporting error thresholds up to $k = 13$, Columba ensures improved performance and scalability in complex alignment scenarios.

Columba achieves its performance through a combination of key innovations that improve computational efficiency while preserving completeness. Dynamic partitioning of search patterns adapts the search procedure to the sequence and reference content, significantly reducing the search space and runtime while maintaining precision in the alignment process. A hybrid strategy combines in-index matching with bit-parallel in-text verification, dynamically choosing the processing method based on the distribution of candidate occurrences. This approach minimizes computational overhead for rare occurrences through in-text verification and employs batch processing within the index for frequent occurrences, achieving a more than twofold runtime reduction compared to pure in-index methods.

Columba's performance is further enhanced through techniques such as reducing redundancy in edit distance calculations and applying memory optimizations tailored to genomic alignment tasks, including interleaved bitvectors. These interleaved bitvectors are specifically designed to reduce cache misses by optimizing memory access patterns during alignment processes, further improving runtime efficiency.

Moreover, Columba's support for a run-length compressed (RLC) index enhances memory efficiency for highly repetitive reference genomes, addressing the challenges typically associated with pan-genomic applications. While bidirectional indexing often increases memory requirements, Columba's RLC implementation offsets this, enabling both versatility and scalability across diverse datasets without compromising efficiency.

Additionally, Columba supports paired-end alignment, leveraging information about both ends of a DNA fragment to filter alignments based on expected fragment size constraints. By utilizing the spatial relationship between paired reads, this feature reduces ambiguities and enhances mapping accuracy. With multithreading capabilities, Columba efficiently parallelizes tasks, enabling effective processing of large-scale

datasets in high-throughput sequencing applications.

Columba was rigorously benchmarked against state-of-the-art lossless tools such as RazerS3, Yara, and Bwolo, demonstrating substantial speed improvements while maintaining comparable precision across all tested scenarios. In challenging situations involving high error thresholds, Columba achieved runtime reductions of up to 38 times relative to its counterparts, underscoring its efficiency. Remarkably, its performance was also competitive with heuristic-based tools like BWA-MEM and Bowtie2, highlighting its ability to bridge the long-standing performance gap between lossy and lossless methods.

Columba can be successfully integrated into real-world bioinformatics workflows, exemplified by its incorporation into the OptiType pipeline for HLA genotyping. In this application, Columba significantly improved runtime while maintaining high accuracy, showcasing its ability to enhance existing workflows seamlessly and effectively. Its adaptability extends to both single-genome and pan-genome analyses, making it a powerful and versatile tool for a wide range of genomic studies.

This work demonstrates that lossless alignment methods can achieve performance levels comparable to heuristic-based tools, effectively narrowing the gap between accuracy and efficiency in genomic alignment tasks. The innovations in this work, including dynamic partitioning, hybrid in-index matching/in-text verification, and advanced search schemes, represent a major advance in lossless alignment methodology. By supporting applications ranging from HLA typing to pan-genomics, Columba highlights the transformative potential of optimized lossless methods in genomic analysis.

In conclusion, this work transforms lossless genomic alignment from a slow, niche technique into a practical and impactful tool. By achieving performance on par with commonly used heuristic aligners while preserving completeness, Columba empowers researchers and clinicians to more confidently identify subtle genetic differences. By making exhaustive and accurate alignment accessible at scale, this research ultimately supports a deeper understanding of (human) genetic diversity and paves the way for more reliable diagnostics, therapies, and large-scale genomic analyses.

# List of Abbreviations

| | |
|---|---|
| **A** | Adenine |
| **AIDS** | Acquired Immunodeficiency Syndrome |
| **APM** | Approximate Pattern Matching |
| **BAM** | Binary sequence Alignment/Map |
| **BLAST** | Basic Local Alignment Search Tool |
| **BLAT** | BLAST-Like Alignment Tool |
| **bp** | base-pair |
| **BWA** | Burrows-Wheeler Aligner |
| **BWT** | Burrows-Wheeler Transform |
| **C** | Cytosine |
| **CCS** | Circular Consensus Sequencing |
| **CIGAR** | Compact Idiosyncratic Gapped Alignment Report |
| **CLR** | Continuous Long Reads |
| **CPU** | Central Processing Unit |
| **ddNTP** | dideoxynucleotide |
| **DNA** | Deoxyribonucleic acid |
| **DP** | Dynamic Programming |
| **FGS** | First-Generation Sequencing |
| **G** | Guanine |
| **GPU** | Graphics Processing Unit |
| **GWAS** | genome-wide association study |
| **gRNA** | guide RNA |
| **HGP** | Human Genome Project |
| **HiFi** | High Fidelity |
| **HIV** | Human Immunodeficiency Virus |
| **HLA** | Human Leukocyte Antigen |
| **HRG** | Human Reference Genome |

| | |
|---|---|
| **HiSeq** | High Throughput Sequencing |
| **ILP** | Integer Linear Programming |
| **indel** | insertion or deletion |
| **IUPAC** | International Union of Pure and Applied Chemistry |
| **MEM** | Maximal Exact Match |
| **MHC** | Major Histocompatibility Complex |
| **MILP** | Mixed ILP |
| **mRNA** | messenger RNA |
| **MSA** | Multiple Sequence Alignment |
| **NGS** | Next-Generation Sequencing |
| **OH** | hydroxyl |
| **ONT** | Oxford Nanopore Technologies |
| **PacBio** | Pacific Biosciences |
| **PAF** | Pairwise mApping Format |
| **PCR** | Polymerase Chain Reaction |
| **RLC** | Run-length compressed |
| **RNA** | Ribonucleic acid |
| **SA** | Suffix Array |
| **SAM** | Sequence Alignment/Map |
| **SMRT** | Single-Molecule Real-Time |
| **SMS** | Single Molecule Sequencing |
| **SNP** | Single Nucleotide Polymorphism |
| **SNR** | signal-to-noise Ratio |
| **T** | Thymine |
| **TGS** | Third-Generation Sequencing |
| **U** | Uracil |
| **WGS** | Whole Genome Sequencing |

# List of Symbols

| | |
|---|---|
| $\mathbb{S}$ | A search scheme |
| $S$ | A search in a search scheme consisting of three arrays $\pi$, $L$ and $U$ |
| $\pi$ | Array representing the order of processing the parts in a search |
| $U$ | Array representing the upper bound to the cumulative number of errors in a search |
| $L$ | Array representing the lower bound to the cumulative number of errors in a search |
| $[i, j[$ | The half-open interval containing all numbers $k$, $i \leq k < j$ |
| $\mathcal{S}(S, T)$ | The range over the suffix array of $T$, where all suffixes are prefixed by $S$ |
| $\Sigma$ | The alphabet |
| $\mathcal{T}$ | The unidirectional search trie for text $T$ |
| $\mathcal{T}^r$ | The unidirectional search trie for the reverse of text $T$ ($T^r$) |
| $\mathbb{T}$ | The bidirectional search trie for text $T$, linking $\mathcal{T}$ and$\mathcal{T}^r$ |
| $\mathcal{P}_i$ | The $i$th part of pattern $P$ |
| $[n]$ | the set $\{0, 1, \ldots, n-1\}$ |

# List of Figures

# List of Tables

# 1 Introduction

The human immune system is a marvel of biological complexity, tirelessly working to protect us from infections, cancers, and other diseases. At its core is a finely tuned mechanism that distinguishes "self" from "non-self", enabling our bodies to fight off harmful invaders while sparing our own cells. This extraordinary precision is largely governed by the Human Leukocyte Antigen (HLA) system — a group of highly variable genes that define how our immune system recognizes and responds to threats.

Remarkably, the chance of two unrelated individuals with identical HLA molecules is extremely low. This uniqueness is both a blessing and a challenge: it underpins our personalized immune responses but also complicates tasks like organ transplantation, where even minor mismatches can trigger life-threatening rejection. Beyond transplantation, subtle differences in HLA variants influence everything from autoimmune disease susceptibility to the progression of infections such as HIV/AIDS [1]. The ability to accurately identify and distinguish these hundreds of HLA variants— often differing by just a single base pair — directly impacts patient outcomes and therapeutic decisions. This critical need has driven the development of increasingly sophisticated methods for HLA typing, the process of determining an individual's specific HLA variants. HLA typing is a cornerstone of personalized medicine and relies on accurate DNA sequence alignment.

Traditional methods for DNA sequence alignment often focus on speed, reporting only a single "best" match and relying on shortcuts that, while faster, can overlook other equally possible matches. This trade-off is especially limiting in HLA typing, where certain genetic sequences are so similar across HLA genes that they can map equally well to multiple variants, leading to ambiguous results [2]. This "lossy" approach, though efficient, risks missing these subtle yet clinically significant differences. Lossless methods, on the other hand, capture all possible matches to ensure complete

accuracy, but they tend to be slower, creating a significant gap in performance.

To address this limitation, this research focuses on an efficient lossless approximate pattern-matching approach that exhaustively reports all co-optimal (equally possible) occurrences. This thesis explores the algorithms and search schemes that make this lossless approach computationally feasible, pushing the boundaries of lossless sequence alignment to narrow or even eliminate the lossy-lossless performance gap and consequently improving HLA typing speed, without compromising accuracy. Here, HLA typing exemplifies the biological relevance of a lossless approximate pattern-matching approach.

---

This introductory chapter begins by presenting essential bioinformatics concepts, that form the basis for the methods used in this research, in Section 1.1. Following this, Section 1.2 covers computer science engineering principles relevant to the technical groundwork in this dissertation. Later, the chapter examines the FM-index, central to the work in this dissertation, in Section 1.3, and explores the eponymous search schemes in Section 1.4. Following this, Section 1.5 on sequence aligners describes the existing tools and approaches used for biological sequence alignment and comparison. In the final sections of this introduction, the research questions that guide this PhD are presented, followed by an outline of the dissertation and a summary of publications resulting from this research.

## 1.1 Bioinformatics Concepts

### 1.1.1 Biological Background

Deoxyribonucleic acid, commonly known as DNA, is a molecule present in nearly all cells of every living organism and serves as the essential blueprint for life. DNA carries the genetic instructions necessary for the growth, development, functioning, and reproduction of all known organisms, as well as many viruses. Essentially, DNA encodes the information needed to construct proteins, which, in turn, carry out countless functions that sustain life.

The discovery of DNA's significance and structure has a fascinating history. In 1869, Friedrich Miescher first observed DNA when he isolated a substance from the nuclei of white blood cells, calling it "nuclein" [3]. Nearly a century later, in 1953, James Watson and Francis Crick, building on key contributions from Rosalind Franklin and Maurice Wilkins, unveiled the iconic helical structure of DNA, describing it as a double helix [4]. This model revealed how genetic information is stored within the sequence of nucleotide bases along each strand of DNA.

DNA's double-helix structure resembles a twisted ladder and consists of two long strands winding around each other, as shown in Figure 1.1. Each strand is composed of a sequence of simpler molecules called nucleotides. Each nucleotide contains three essential components: a phosphate group, a deoxyribose sugar molecule, and one of four nitrogenous bases — Adenine (A), Thymine (T), Cytosine (C) and Guanine (G).

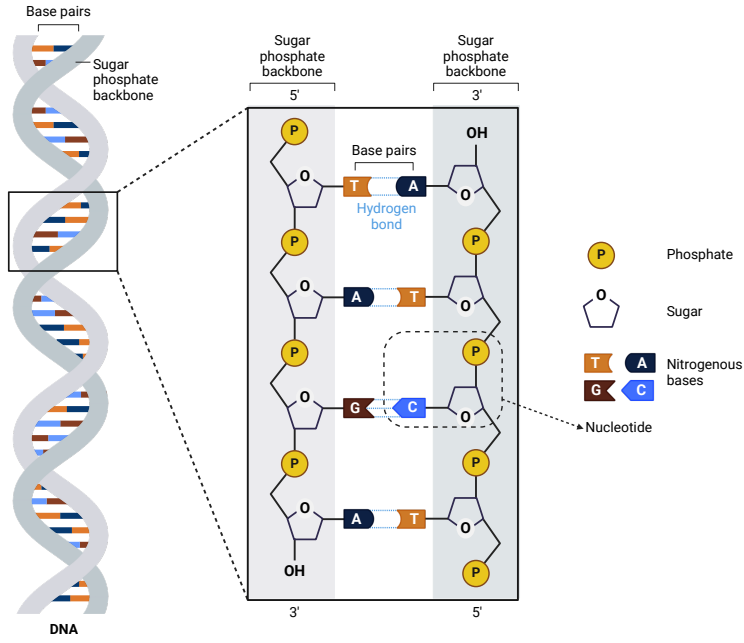


***Figure 1.1:*** *Illustration of the DNA double helix structure showcasing the sugar-phosphate backbone and the paired nitrogenous bases (adenine-thymine and cytosine-guanine) connected by hydrogen bonds. The strand on the left within the highlighted rectangle has its 3' end at the bottom, while the complementary strand has its 5' end at the bottom, illustrating the antiparallel orientation of the two strands. The figure is inspired by [5] and is created in BioRender.*

These nitrogenous bases pair specifically: adenine with thymine, and cytosine with guanine, forming the 'rungs' of the DNA ladder. The sequence of these base-pairs (bp) encodes the genetic information that determines how the proteins of an organism are structured and function.

Each strand of DNA has directionality, defined by the structure of the deoxyribose sugar in the sugar-phosphate backbone. The sugar molecule has two distinct attachment points known as the 3' (three-prime) end and 5' (five-prime) end. The 5' end has a phosphate group attached, while the 3' end has a hydroxyl (OH) group. DNA synthesis and extension, whether occurring naturally during DNA replication or in laboratory processes like sequencing, always proceed from the 5' end to the 3' end, with new nucleotides added to the 3' end.

In the double helix, the two DNA strands are antiparallel, meaning they run in opposite directions: one strand goes from 5' to 3', while its partner runs from 3' to 5'. This antiparallel structure allows each base on one strand to pair with its complement on the opposite strand. For instance, in the highlighted rectangle in Figure 1.1 the left strand reads 5'-TAGA-3' (top to bottom), the complementary strand, running in the reverse direction, reads 5'-TCTA-3' (bottom-to-top). This arrangement is known as

the reverse complement. Generally, we omit the 5' and 3' labels, and sequences are assumed to flow from 5' to 3', when denoting a DNA strand, unless otherwise specified.

While DNA serves as the genetic storage of information, Ribonucleic acid (RNA) plays a crucial role in translating this information into functional proteins. Unlike DNA, RNA is typically single-stranded and contains the sugar ribose rather than deoxyribose. Additionally, RNA includes Uracil (U) in place of Thymine. Furthermore, some viruses contain only RNA as their genetic material. The process of converting DNA instructions into proteins involves two main steps: transcription and translation.

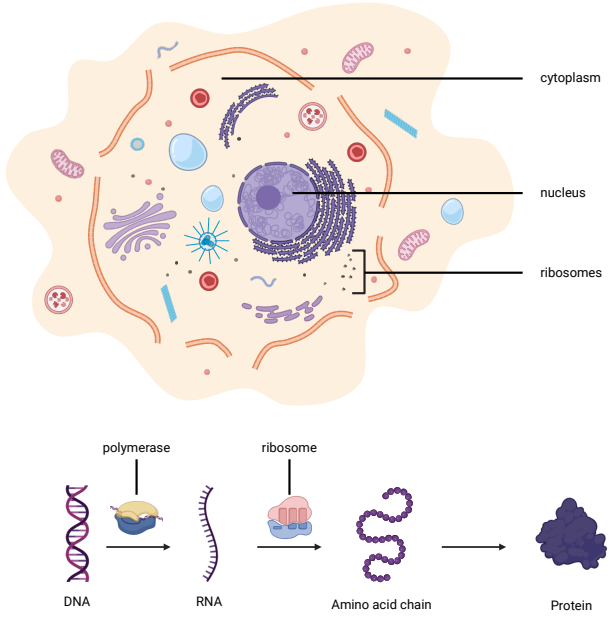


***Figure 1.2:*** *Top: An animal eukaryotic cell, highlighting the nucleus as the site of transcription, and the cytoplasm containing ribosomes for translation. Bottom: The central dogma of molecular biology, showing transcription in the nucleus, where DNA is transcribed into RNA by polymerase, and translation in the cytoplasm, where ribosomes synthesize amino acid chains, that fold into proteins. This figure is created in BioRender.*

In eukaryotic cells (cells with a nucleus, such as those in plants, animals, and fungi), transcription occurs within the nucleus. During transcription, a segment of DNA is copied into messenger RNA (mRNA), creating a temporary copy of the genetic code. This process and its spatial context are illustrated in Figure 1.2, where the top panel highlights the eukaryotic cell structure, including the nucleus where transcription occurs. Before the mRNA leaves the nucleus, it undergoes several post-transcriptional modifications, including the addition of a 5' cap and a poly-A tail, which help stabilize the mRNA and aid in its export from the nucleus. Introns, or non-coding regions,

are also spliced out to create a mature mRNA strand ready for translation. This modified mRNA then exits the nucleus and enters the cytoplasm. Once in the cytoplasm, as shown in the bottom panel of Figure 1.2, translation occurs at ribosomes (complex macromolecular structures in the cytoplasm). In prokaryotic cells (cells without a nucleus, such as bacteria), the process is similar, but without a defined nucleus, transcription and translation can occur simultaneously in the cytoplasm.

At the ribosome, the mRNA sequence is read in sets of three nucleotides, known as codons, each corresponding to a specific amino acid — the building blocks of proteins. The ribosome assembles amino acids in the correct sequence according to the mRNA code, forming a polypeptide chain. After translation, this polypeptide chain often undergoes post-translational modifications, such as phosphorylation, glycosylation, and folding, which are essential for the protein to achieve its functional, three-dimensional structure and become biologically active [6].

#### 1.1.1.1 The Genome

The genome of an organism is the complete set of genetic material within its cells. This includes not only the DNA sequences that code for proteins but also other regions that play regulatory or structural roles. In eukaryotic organisms, the genome is organized into chromosomes within the cell nucleus, while in prokaryotic organisms, the genome typically exists as a single, circular chromosome located in the cytoplasm.

The human genome, for example, consists of approximately three billion base-pairs spread across 23 pairs of chromosomes, with each chromosome containing thousands of genes. In addition to the nuclear genome, eukaryotic cells also have a mitochondrial genome, which is a small, circular DNA molecule found within mitochondria that is inherited maternally.

**Genes and Alleles** Genes are segments of DNA that encode instructions for producing proteins, which perform essential roles in an organism's structure, function, and overall traits. Each gene occupies a specific position, or locus, on a chromosome and contributes to various characteristics, such as eye color, blood type, or enzyme function. The exact expression of these traits can vary, depending on the specific versions of a gene that an individual inherits. These different versions of a gene are known as alleles. Alleles can have slight variations in their DNA sequences, which can lead to observable differences in the traits a gene influences.

One of the most common types of genetic variation that can result in different alleles is a Single Nucleotide Polymorphism (SNP). A SNP is a change in a single nucleotide, or DNA building block, at a specific position in the genome. For instance, a particular SNP might replace an Adenine (A) with a Cytosine (C) at a given location in the DNA sequence. SNPs are a major source of genetic diversity within populations and can affect how genes function, often contributing to subtle differences in traits or susceptibility to diseases.

In diploid organisms, such as humans and most animals, each cell contains two sets of chromosomes, with one copy of each gene from each parent. This gives each individual two alleles for most genes, which can be either identical (homozygous) or

different (heterozygous). If an individual is homozygous, they have two identical alleles at a gene locus, leading to consistent expression of that trait. For traits linked to a recessive allele, homozygosity is necessary for the trait to be expressed. In heterozygous individuals, who have two different alleles, the dominant allele often masks the recessive one, displaying the dominant trait. However, some traits result in blended or co-dominant expression, as with the AB blood type, where both alleles are expressed. These combinations, along with variations like SNPs, contribute to the genetic diversity seen in populations, shaping how traits are inherited and expressed across generations.

**Example: Human Leukocyte Antigen (HLA)** The HLA region, also referred to as the Major Histocompatibility Complex (MHC) in humans, offers a compelling example of genetic diversity and its role in immune function. Located on chromosome 6, the HLA region spans approximately four megabases and is one of the most gene-dense regions in the human genome. It contains over 150 protein-coding genes that are essential for immune system regulation, particularly in recognizing and responding to foreign antigens.

HLA genes are highly polymorphic, with numerous alleles, some of which differ by just a SNP, that contribute to a wide array of immune responses across individuals. This diversity is crucial for the adaptive immune system, as HLA molecules present antigenic peptides to T cells, enabling the immune system to detect and respond to pathogens.

This genetic variability has been linked to differential susceptibility to infectious diseases and immune-related disorders. For example, while specific mutations like the CCR5$\Delta$32 variant are known to directly impact Human Immunodeficiency Virus (HIV) susceptibility, genome-wide association studies (GWAS) have documented numerous HLA-related SNPs associated with diseases such as Acquired Immunodeficiency Syndrome (AIDS), hepatitis, tuberculosis, and malaria, highlighting the complex relationship between host genetic variation and disease susceptibility across populations [1].

Due to the high variability in HLA alleles, matching donor and recipient HLA types is a critical factor in transplantation medicine. A closer HLA match reduces the likelihood of transplant rejection, as the immune system is less likely to recognize the donor tissue as “non-self”. Beyond its role in immunity, HLA diversity has also been linked to factors such as mate selection, pregnancy maintenance, and even neurological and behavioral traits, underscoring the extensive influence of the HLA region on human health and biology [7].

### 1.1.2 Genome Sequencing

Genome sequencing is a fundamental technique used to determine the precise order of nucleotides in a DNA molecule. This process enables researchers to explore genetic variation, diagnose diseases, and advance personalized medicine. Genome sequencing typically produces “reads”, which are short fragments of DNA that correspond to specific portions of the genome. The complexity of genome sequencing arises from

the fact that these reads are often fragmented and must be reassembled or aligned to a reference genome for analysis.

Paired-end sequencing takes advantage of DNA's double-stranded nature by sequencing both ends of a fragment, producing a pair of reads: one from each end. These paired reads are separated by a known distance, or *insert size*, which provides additional context that helps in aligning reads more accurately, particularly in complex or repetitive regions.

Advances in sequencing technologies, particularly with Next-Generation Sequencing (NGS) and more recently, Third-Generation Sequencing (TGS), have dramatically decreased the cost and time required to sequence genomes. These technologies offer improved throughput and higher accuracy, though challenges remain. The increasing amount of genomic data generated calls for efficient storage, alignment, and analysis methods.

#### 1.1.2.1 Reference Genome

A reference genome is a representative DNA sequence for a species, acting as a standardized baseline against which individual genomic reads can be compared. By compiling sequences from multiple individuals, a reference genome facilitates the identification of genetic variations, such as SNPs and structural variations.

The creation of the first human reference genome was a landmark achievement completed by the Human Genome Project (HGP) in 2003. This international effort, which took over a decade and involved hundreds of researchers, produced a comprehensive map of the human genome at a cost of approximately three billion USD. However, because the initial human reference genome was derived from a limited sample of individuals, it reflects only a fraction of the genetic diversity found within human populations.

**Pan-Genomes** To address the limitations of single-reference models, researchers are increasingly developing pan-genomes. A pan-genome represents the genetic diversity across multiple individuals of a species, capturing a more comprehensive view of its genetic variability. Pan-genomes are not limited to humans; they have been extensively used to characterize genetic diversity within bacterial species, where variation can affect pathogenicity, antibiotic resistance, and metabolic functions. In bacterial genomics, pan-genomes enable a thorough understanding of the core genes shared across all strains as well as the accessory genes that contribute to strain-specific traits. This approach enhances species identification by providing a reference that includes the genetic breadth within a species, allowing for the differentiation between closely related strains or species based on unique genetic markers [8].

For human genomics, pan-genomes are essential for studying complex and highly variable regions, such as the HLA locus. Initiatives like the Human Pan-Genome Project aim to produce more inclusive reference models that capture rare variants and population-specific alleles, improving the accuracy of genomic analyses and supporting personalized medicine across diverse populations [9].

#### 1.1.2.2 Sequencing Technologies

DNA sequencing has undergone significant transformations since the 1970s, progressing through three major technological generations, each with unique methodologies, applications, and improvements in accuracy, speed, and cost. The Polymerase Chain Reaction (PCR) [10], developed in 1983, has been central to sequencing advancements across these generations, enabling the rapid amplification of DNA fragments and enhancing sequencing signal strength and reliability.

**Early Innovations: Sanger Sequencing and the Foundations of First-Generation Sequencing** The first widely adopted DNA sequencing method, Sanger sequencing, was developed by Frederick Sanger in the late 1970s [11]. Originally known as chain-termination sequencing, this approach was the first to successfully determine the sequence of DNA strands by terminating DNA synthesis at specific bases. The technique uses chemically modified nucleotides, known as dideoxynucleotides (ddNTPs), which lack the 3' hydroxyl group necessary for chain elongation. When incorporated into a DNA strand, these nucleotides halt the extension, creating fragments of various lengths that can be detected through electrophoresis.

Each terminating nucleotide (ddNTP) is tagged with a fluorescent dye, allowing detection and identification of the final base in each fragment. The fragments are sorted by length, and the fluorescent labels reveal the DNA sequence in a method that became the standard for high-accuracy, long-read sequencing, with read lengths up to 1 000 bp. Sanger sequencing, later retrospectively named First-Generation Sequencing (FGS), was fundamental to landmark projects like the HGP. However, its low throughput and high costs limited its scalability, as sequencing large genomes with Sanger required considerable time and resources.

**Scaling Up: The Rise of Next-Generation Sequencing and the Role of PCR** Next-Generation Sequencing (NGS), which emerged in the early 2000s, transformed the field by introducing massively parallel sequencing, a method allowing millions of DNA fragments to be sequenced simultaneously. This advancement reduced sequencing costs dramatically and enabled high-throughput sequencing at an unprecedented scale. Unlike Sanger sequencing, NGS produces short reads (typically between 35 and 500 base pairs) and relies heavily on PCR for sample preparation.

In NGS, sample DNA is fragmented, and specific adapters are added to the ends of each fragment. Using PCR, the fragments are amplified to produce numerous identical copies, essential for detecting each base with high accuracy. Illumina, one of the most widely used NGS platforms, employs a method called sequencing by synthesis. In this process, illustrated in Figure 1.3, DNA fragments are immobilized on a flow cell and amplified into clusters through bridge PCR, forming dense clusters of identical fragments. Fluorescently labeled nucleotides are sequentially incorporated, and emitted fluorescence allows each base to be read, cycle by cycle.

Other NGS platforms, such as Roche 454 and SOLiD, introduced alternative methods, including pyrosequencing and sequencing by ligation, each with unique advantages for specific applications. Nowadays, Illumina dominates the production of NGS

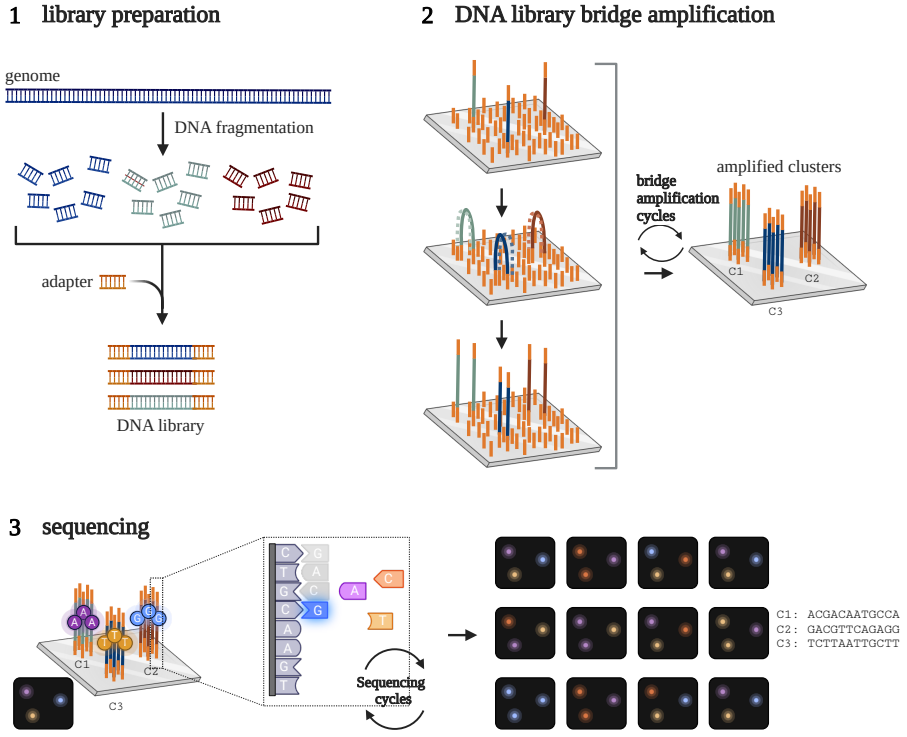


***Figure 1.3:*** *Illustration of Illumina Next-Generation Sequencing (NGS) workflow, highlighting the main steps of DNA library preparation, bridge amplification, and sequencing by synthesis. During library preparation (1), DNA is fragmented, and adapters are added to the ends of each fragment. The fragments are then attached to a flow cell and amplified through bridge amplification (2), forming clusters of identical DNA sequences. In the sequencing phase (3), fluorescently labeled nucleotides are incorporated one at a time into each DNA strand. When a labeled nucleotide binds, it emits light. After each incorporation, the flow cell is washed to remove any unincorporated nucleotides, and a digital image is captured to record the fluorescence signal. In the bottom right corner of the figure, these digital images are arranged in a left-to-right and top-to-bottom order, reflecting the cycle-by-cycle progression of sequencing across multiple clusters. Each cluster, such as C1, C2, and C3, can thus be sequenced by analyzing the sequence of fluorescence signals, enabling high-throughput determination of the DNA sequences. The figure is created in BioRender and is based on images from [12, 13].*

data and most often NGS data implies data produced by Illumina machines.

Although NGS revolutionized genomics by providing fast, affordable sequencing, its reliance on short reads creates challenges for accurately sequencing repetitive and structurally complex regions of the genome. PCR amplification, while crucial for signal strength, can also introduce biases in sequence data, particularly in regions with high or low GC content [14].

**Moving Beyond Amplification: Third-Generation Sequencing and Single-Molecule Approaches** The development of Third-Generation Sequencing (TGS) in the 2010s addressed several limitations of NGS by enabling single-molecule sequencing, which eliminates the need for PCR amplification. TGS technologies, such as Single-Molecule Real-Time (SMRT) by Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT) sequencing, can produce reads that range from 5 000 to 50 000 base-pairs or more. This increased read length is particularly valuable for accurately sequencing complex genomic regions, structural variants, and large-scale rearrangements, which are difficult to resolve with shorter reads.

In PacBio's SMRT sequencing, DNA fragments are prepared with circular adapters, allowing each molecule to be sequenced multiple times as it passes through a polymerase. This results in two types of reads: Continuous Long Reads (CLR), which capture the full length of the DNA fragment, and Circular Consensus Sequencing (CCS) reads, which produce highly accurate High Fidelity (HiFi) reads by sequencing the same DNA molecule repeatedly to achieve consensus. These HiFi reads offer higher accuracy, reducing the errors often associated with long-read technologies [15].

Oxford Nanopore's sequencing uses a different principle, threading DNA strands through tiny protein pores (nanopores) embedded in a membrane. Each nucleotide passing through the pore disrupts an electric current in a characteristic way, allowing real-time base identification. Nanopore sequencing offers the advantage of portability and scalability, as it can be performed on small devices like the MinION and can sequence DNA strands of virtually unlimited length in real-time. While nanopore sequencing has higher error rates than SMRT, it is rapidly improving in accuracy with advances in software and nanopore chemistry.

#### 1.1.2.3 The Read Alignment Problem

The read alignment problem is akin to solving a puzzle where each read represents a small fragment of a much larger picture — the reference genome. However, this "puzzle" is complicated by several factors. Some reads may contain sequencing errors, similar to puzzle pieces with slight printing mistakes, which can make it difficult to place them accurately. Additionally, genetic variations, such as SNPs and small insertions or deletions, mean that even correct reads may differ slightly from the reference sequence.

Given the vast number of reads generated (often millions or billions) and the large size of reference genomes, directly scanning the entire genome for each read would be computationally unfeasible. To address this, read alignment tools like Burrows-Wheeler Aligner (BWA) [16] and Bowtie [17] (see Section 1.5) use indexing methods

to narrow down potential alignment locations quickly. These tools employ data structures such as the Burrows-Wheeler Transform (BWT) and FM-index (see Section 1.3) to locate possible matches efficiently, allowing them to balance alignment accuracy with the need for speed.

Efficient read alignment is crucial because it maps raw sequencing reads to a reference genome, providing the foundation for downstream analyses such as detecting genetic variants, including SNPs, insertions or deletions (indel), and structural variants. Once reads are aligned, subsequent steps like variant calling and comparative genomics can be conducted, providing insights into genetic differences, disease susceptibility, or species diversity in metagenomic samples.

#### 1.1.2.4 Storage of Genomic Data

Effective storage and management of genomic data are essential for handling the large amounts of sequence information generated by sequencing technologies. Standardized file formats, including FASTA, FASTQ, and SAM, are widely used for storing sequence data and associated metadata, each serving specific roles in bioinformatics workflows.

**FASTA Format** The FASTA format, popularized by Pearson and Lipman's FASTA software [18], is one of the most basic file formats for storing nucleotide or protein sequences. Each entry in a FASTA file begins with a header line, starting with the '>' symbol followed by a unique identifier and optional descriptive information. The sequence itself follows, typically wrapped to a standard line length. FASTA files are commonly used for storing raw sequence data. Example 1.1 shows a FASTA record.

In a FASTA file, nucleotide sequences are represented using standard one-letter codes. When a specific base is unknown, the International Union of Pure and Applied Chemistry (IUPAC) nucleic acid notation provides characters to denote ambiguity in the sequence. Table 1.1 summarizes the IUPAC codes for both unambiguous and ambiguous bases to indicate uncertainty about particular nucleotides.

In paired-end sequencing, two reads originating from the same DNA fragment are referred to as "mates". Typically, these mate pairs share the same sequence identifier prefix, with one read labeled as _1 and the other as _2. This convention helps maintain the association between the two reads while distinguishing them within the data, ensuring they are recognized as originating from the same DNA fragment.

---

**Example 1.1.** ***Example of a FASTA Entry:***

```
>seq1 Human mitochondrial DNA
ATGCTTAGCTAGCTACGATCGATCGTAGCTAGCTAGCATCGATCGTAGCTA
GCTAGCTACGTAGCTAGCTGACCTAGCTACGATCGATCGTAGCTACATCGA
```

*The identifier of this FASTA entry is "seq1", while "Human mitochondrial DNA" is additional descriptive information.*

---

***Table 1.1:*** *IUPAC codes used in FASTA format for nucleotides.*

| Symbol | Description |
|---|---|
| A | Adenine |
| C | Cytosine |
| G | Guanine |
| T | Thymine |
| U | Uracil (for RNA sequences) |
| R | A or G (purine) |
| Y | C or T (pyrimidine) |
| S | G or C (strong interaction) |
| W | A or T (weak interaction) |
| K | G or T (keto) |
| M | A or C (amino) |
| B | C, G, or T (not A) |
| D | A, G, or T (not C) |
| H | A, C, or T (not G) |
| V | A, C, or G (not T) |
| N | Any nucleotide (A, T, C, G) |

**FASTQ Format** The FASTQ format [19] extends the FASTA format by including quality scores for each base in the sequence. Each entry in a FASTQ file contains four lines:

1. a header line beginning with '@', followed by a unique identifier;
2. the nucleotide sequence;
3. a '+' line, which may optionally repeat the sequence identifier and
4. a line with Phred quality scores.

The Phred Quality scores represent probability estimations of base-calling accuracy, where base-calling refers to the process of determining the nucleotide sequence from raw sequencing data, made by the sequencing platform or basecaller. Each Phred score $Q$ indicates the likelihood that a particular base call is incorrect, based on the signal-to-noise Ratio (SNR) observed during sequencing. The Phred score $Q$ is calculated from the probability $P$ of an error occurring in a base call using the formula $Q = -10 \cdot \log_{10} P$. This logarithmic transformation means that higher Phred scores correspond to lower probabilities of error, implying greater confidence in the accuracy of a base call. For instance, a Phred score $Q = 0$ is the lowest possible score and corresponds to an almost certain error, while a Phred score $Q = 30$ suggests that only one in every 1 000 base calls is expected to be incorrect.

In a FASTQ file, these Phred scores are encoded as ASCII characters, with an offset applied to map each score to a character. Typical offsets are 33 and 64. When using an offset of 33, for example, $Q = 0$ corresponds to the ASCII character '!',

while higher scores are represented by subsequent characters. This encoding allows the quality information for each base in a sequencing read to be compactly represented as a string of ASCII characters, making FASTQ files both space-efficient and highly informative for downstream bioinformatics analysis. Example 1.2 shows a FASTQ entry.

**Example 1.2.** ***Example of a FASTQ Entry:***

```
@seq1 Read 1
GATCGGAAGAGCACACGTCTGAACTCCAGTCAC
+
IIIIIIIIIHIIHIHIIHIIIHHHHHIHIIIII
```

**SAM Format** The SAM (Sequence Alignment/Map) format [20] is designed for storing aligned sequence data and is commonly used as an output format for read alignment tools. A SAM file consists of a header section and an alignment section. The header begins with lines starting with '@', which provide metadata, including information about the reference genome. Each alignment entry contains 11 tab-separated mandatory fields, as described in Table 1.2, along with optional fields for additional data.

***Table 1.2:*** *Mandatory fields in a SAM alignment entry.*

| Field | Description |
|---|---|
| QNAME | Query (read) name |
| FLAG | Bitwise flag indicating read properties (e.g., paired, mapped) |
| RNAME | Reference sequence name (chromosome or contig) |
| POS | 1-based leftmost position of the alignment |
| MAPQ | Mapping quality score (Phred-scaled) |
| CIGAR | CIGAR string indicating alignment (e.g., matches, insertions, deletions) |
| RNEXT | Mate reference name (if paired) |
| PNEXT | Mate alignment position |
| TLEN | Observed Template Length |
| SEQ | Query sequence (aligned DNA) |
| QUAL | Quality scores for each base in SEQ (ASCII-encoded +33) |

For example, the FLAG field in a SAM file is a bitwise code that represents key properties of the read, such as whether it is part of a paired-end read or mapped to the reverse strand. See Table 1.3 for a detailed breakdown of the flags. The CIGAR string (short for Compact Idiosyncratic Gapped Alignment Report) describes the alignment in terms of operations like matches (M), insertions (I), deletions (D), and other structural changes, providing essential information for understanding gaps or mismatches in the alignment. Additional optional fields in the SAM entry offer further details about the alignment, often used for downstream analyses.

***Table 1.3:*** *Description of Flags and Their Meanings in SAM Format.*

| Hex | Decimal | Description |
|---|---|---|
| 0x1 | 1 | Template having multiple segments in sequencing (i.e. two mates) |
| 0x2 | 2 | Each segment properly aligned according to the aligner |
| 0x4 | 4 | Segment unmapped |
| 0x8 | 8 | Next segment in the template unmapped |
| 0x10 | 16 | SEQ being reverse complemented |
| 0x20 | 32 | SEQ of the next segment in the template being reverse complemented |
| 0x40 | 64 | The first segment in the template |
| 0x80 | 128 | The last segment in the template |
| 0x100 | 256 | Secondary alignment |
| 0x200 | 512 | Not passing filters, such as platform/vendor quality controls |
| 0x400 | 1024 | PCR or optical duplicate |
| 0x800 | 2048 | Supplementary alignment |

**Example 1.3.** ***Example of a SAM Entry:***

```
seq1 147 chr1 100 60 21M chr1 140 62 GATCGGAAGAGCACACGTCTG IIIIHI
IHIHIIHIHHIHII
```

*In the above SAM entry the alignment of the read "seq1" to the reference "chr1" is shown. In this entry, the FLAG value is 147, which breaks down to* 1+2+16+128. *This indicates that the read is part of a paired-end sequence (1), both mates are properly aligned (2), the read is aligned to the reverse complement of the reference genome (16) and this read was the last read of the pair (128). The read is aligned to position 100 on the sequence called "chr1", while its mate is mapped to position 140 on the same sequence. The read was aligned with a mapping score of 60. The Compact Idiosyncratic Gapped Alignment Report (CIGAR) string "21M" specifies that the read was aligned without any insertions or deletions, spanning 21 matching bases. The size of the total fragment, incorporating both mates, as determined by the aligner is 62.*

## 1.2 Computer Science Engineering Concepts

This section provides an overview of key computer science concepts necessary to understand the computational approaches discussed in this thesis. The topics covered include data structures, complexity of algorithms, dynamic programming, cache memory and parallel computing. These foundational ideas are introduced with formal definitions, examples, and notational conventions used throughout the thesis.

### 1.2.1 Data Structures and Notation

At the core of all computational processes is the representation and manipulation of data. Computers store data as sequences of binary values (0s and 1s), where each binary digit, or **bit**, represents a state: either on (1) or off (0). Collections of bits are grouped into larger units to encode meaningful information that the hardware can process. Eight bits make up one **byte**, which is often used as the basic unit of storage in computer systems. A computer **word** is a fixed-length sequence of bytes and is typically 4 bytes (32 bits) or 8 bytes (64 bits) long, depending on the architecture of the system. Figure 1.4 shows this hierarchy.

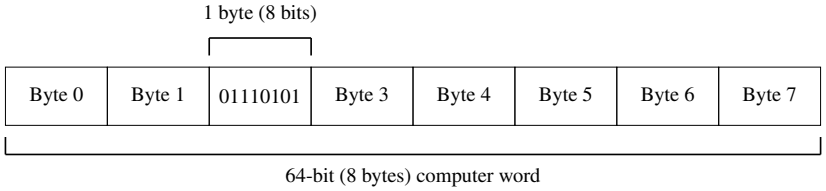


***Figure 1.4:*** *Illustration of data representation in a 64-bit computer system: A single byte (8 bits) highlighted within a 64-bit (8-byte) computer word, demonstrating the hierarchical grouping of bits into bytes and words.*

Data structures provide a way to organize and manage this raw binary data to enable efficient computation. They define how data is encoded in memory, how it can be accessed, and how relationships between different pieces of data are represented. By combining fundamental elements like bits and bytes into structured forms such as arrays, strings, and graphs, data structures bridge the gap between machine-level binary representation and the abstract concepts required for problem-solving in computer science.

#### 1.2.1.1 Linear Data Structures

In this section, we introduce foundational linear data structures: bitvectors, arrays, and strings. These structures play a critical role in efficiently organizing and accessing sequential data. An example is provided in Example 1.4.

A **bitvector** is a compact data structure used to represent a sequence of binary values (0s and 1s). It consists of a collection of bits in a set order. Bitvectors are widely used because they offer an efficient way to store binary data, especially when memory usage needs to be minimized.

**Definition 1.1.** *A **bitvector** $B$ of length $n = |B|$ is defined as:* $\boldsymbol{B} = [b_0, b_1, b_2, \ldots, b_{n-1}]$ *where each* $b_i \in \{0, 1\}$ *for* $0 \leq i < n$.

An **array** is a fundamental data structure that represents a collection of elements, all of the same type, stored in contiguous memory locations. While a bitvector can be viewed as an array of bits, arrays in general are more versatile and are often used to store numbers, characters, or other data types. Each element in an array has a fixed size, such as a bit, byte, word, or larger unit, depending on the type of data being stored.

**Definition 1.2.** *An **array** $A$ of length $n = |A|$ is defined as: $A = [a_0, a_1, a_2, \ldots, a_{n-1}]$ where each $a_i$ is an element of a specific type (e.g., integer, character), and $0 \leq i < n$ represents the index of the element in the array. The $i$-th element of the array, starting from index 0, is denoted as $A[i]$, hence the first element is $A[0]$.*[1] *The size of each element is uniform, allowing efficient indexing and direct access to any element using its index. If the elements are easily distinguishable, such as in the case of single-digit numbers, the array may also be represented as: $A = a_0a_1a_2 \ldots a_{n-1}$.*

A **string** is an array of characters over a defined alphabet. Strings are used to store textual data, such as DNA sequences encoded with characters from the alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ or plain English text represented using the standard English alphabet.

**Definition 1.3.** *A **string** $S$ of length $n = |S|$ is a sequence of characters over an alphabet $\Sigma$, defined as: $S = [c_0, c_1, c_2, \ldots, c_{n-1}]$, where $S[i] = c_i$ is the character at position $i$ in $S$, with $0 \leq i < n$. We can also denote $S$ as "$c_0c_1c_2 \ldots c_{n-1}$".*

---

**Example 1.4.** *Illustration of bitvectors, arrays and strings.*

1. *$B = [1, 0, 1, 1, 0]$ is a bitvector of length 5. $B[2] = 1$ and $B[4] = 0$.*
2. *$A = [3, 1, 4, 1, 5]$ (or $A = 31415$) is an array of single-digit integers of length 5. $A[2] = 4$ and $A[4] = 5$.*
3. *$S =$ "`HELLO`" is a string of length 5. $S[2] =$ '`L`' and $S[4] =$ '`O`'.*

---

In many scenarios, it is useful to refer to a portion of an array or string rather than the entire structure. This is achieved using the concept of a **range** or interval. A range is defined by two integers, $i$ and $j$, where $i \leq j$, and represents a portion of the structure defined by the interval $[i, j[$. A key characteristic of these ranges is that they are **half-open intervals**,[2] meaning the start index is inclusive, while the end index is exclusive. For example, the range $[0, n[$ over a string of length $n$ corresponds to the entire string. Similarly, a range $[i, i[$ denotes an empty range.

**Definition 1.4.** *A **range** is a pair of integers $i$ and $j$, with $i \leq j$, defining a half-open interval $[i, j[$. The size of the range is $j - i$. A range $[i, j[$ over a string $S$ corresponds to all characters $S[k]$, where $i \leq k < j$ , $0 \leq i$ and $j \leq |S|$. Ranges over arrays are defined analogously.*

A **suffix** of a string $S$ is a substring that starts at a given position $i$ and continues to the end of the string. For a string $S$ of length $n$, the suffix starting at position $i$ is denoted as $\text{suf}_i(S) = S[i, n[$ and is called the $i$th suffix of $S$. For example, for the string $S =$"`HELLO`", the suffix $\text{suf}_2(S)$ is "`LLO`".

[1] Note that here we start indexing at zero. This is common in computer science and programming languages such as C, Python, and Java.

[2] This convention of half-open intervals and zero-based indexing is widely adopted in computer science and programming, aligning with the preferences of Dijkstra [21] and the ISO C++ standard [22].

Similarly, a **prefix** of a string $S$ is a substring that starts at the beginning of the string and ends at a given position $j$. The prefix ending at position $j$ is denoted as $\text{pre}_j(S) = S[0, j[$. For the string $S =$"HELLO", the prefix $\text{pre}_3(S)$ is "HEL".

A **2D array**, also known as a **matrix**, is an extension of the one-dimensional array concept to two dimensions. It represents a collection of elements arranged in rows and columns, where each element is uniquely identified by two indices: the row index and the column index.

**Definition 1.5.** *Matrix A **matrix** (or 2D array) $M$ with r rows and c columns is defined as:*

$$M = \begin{bmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,c-1} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,c-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{r-1,0} & m_{r-1,1} & \cdots & m_{r-1,c-1} \end{bmatrix},$$

*where $m_{i,j}$ is the element in the ith row and jth column of the matrix, with $0 \leq i < r$ and $0 \leq j < c$. Accessing the element at row i and column j is denoted as $M[i, j]$.*

Each row in the matrix can be viewed as a one-dimensional array, and the matrix itself can be represented as an array of arrays in memory. For example, the first row is $M[0] = [m_{0,0}, m_{0,1}, \ldots, m_{0,c-1}]$.

---

**Example 1.5.** *Consider a $2 \times 3$ matrix:*

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

*Here, $M[0, 1]$=2 represents the element in the first row and second column.*

---

#### 1.2.1.2 Non-linear Data Structures

Arrays, strings, and matrices are all types of linear data structures, i.e., they have a natural linear ordering. In this work, we will also make use of two non-linear data structures: **rooted trees** and **tries**. Unlike linear structures, non-linear data structures do not have a strict sequential ordering and are used to represent hierarchical or relational data.

**Definition 1.6.** *A **rooted tree** is a hierarchical data structure consisting of a finite set of nodes and directed edges (or branches) that connect pairs of nodes. Formally, a tree $T$ is defined as $T = (V, E)$, where $V$ is a set of **nodes** and $E$ is a set of directed edges, such that there exists exactly one path from a designated **root node** $r \in V$ to any other node. The root node is unique and has no incoming edges, while each other node has exactly one incoming edge from its **parent**. Nodes may have zero or more **children**, which are connected by outgoing edges. A node with no children is called a **leaf**. The nodes along the path from the root to a given node (excluding that node) are its **ancestors**, while nodes that share the same parent are called **siblings**. The **depth of a node** is the number of edges from the root to that node, and the **height of the tree** is the maximum depth of any node.*

**Example 1.6.** *Consider the rooted tree $T$ depicted in Figure 1.5. In this tree, node $\alpha$ serves as the root and has two children: nodes $\beta$ and $\gamma$. node $\beta$ has a single child, node $\delta$, which is a leaf because it has no children. Nodes $\epsilon$ and $\zeta$ are the children of node $\gamma$, and both are leaves. The depth of the leaf nodes $\delta$, $\epsilon$, and $\zeta$ is 2, which also represents the height of the tree as it is the maximum depth.*

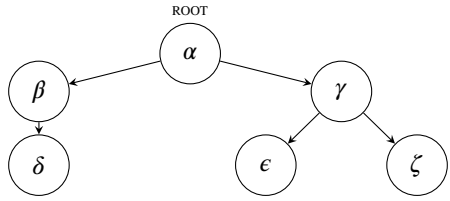


***Figure 1.5:*** *Example of a rooted tree with root $\alpha$, internal nodes $\beta$ and $\gamma$, and leave nodes $\delta$, $\epsilon$, and $\zeta$.*

**Tries.** A **trie**, also known as a **prefix tree**, is a specialized tree structure used to store and retrieve strings efficiently. Unlike general rooted trees, tries are designed to represent a set of strings by organizing them based on shared prefixes. Each path in a trie represents a sequence of characters that is a prefix of one or more of the stored strings. In contrast, a **suffix tree** is a variation of a trie in which each path in the trie corresponds to a suffix of one of the stored strings.

**Definition 1.7.** *A **trie** is a rooted tree in which each node represents a prefix of the strings stored in the trie. The root node represents the empty prefix. Edges between nodes are labeled with characters, indicating extensions of the prefix. Leaf nodes represent complete strings in the set. A string $S = c_0c_1c_2 \dots c_{n-1}$ can be stored in a trie by following a path from the root node through edges labeled $c_0, c_1, \dots, c_{n-1}$.*

**Example 1.7.** *Consider a trie constructed from the strings "`ARE`", "`ARM`", "`BAD`", and "`BAR`". The trie would have the following structure:*

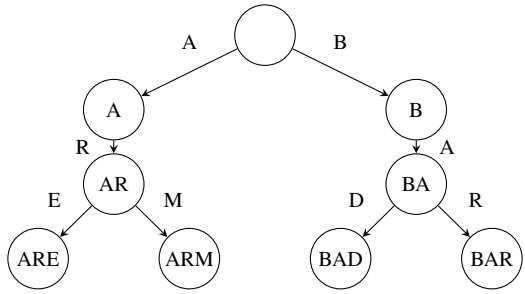


***Figure 1.6:*** *A trie storing the strings "`ARE`", "`ARM`", "`BAD`", and "`BAR`" and their prefixes "`A`", "`AR`", "`B`", and "`BA`".*

### 1.2.2 Complexity of Algorithms

Algorithmic complexity describes how the required steps or memory usage of an algorithm grow as input size increases, using **Big-O** notation to represent these relationships. Big-O focuses on the dominant factors influencing growth, ignoring constant factors and lower-order terms, as their impact diminishes with sufficiently large inputs.

Time complexity measures how the number of steps required by an algorithm scales with the size of the input. For instance, accessing a specific element in a fixed position in arrays always takes the same number of steps, regardless of the input size. This is known as constant time, or $O(1)$. By contrast, tasks that involve processing every element of an array scale proportionally with the size of the array $n$, resulting in linear time, or $O(n)$. More complex operations, such as comparing every pair of elements in a dataset, require steps proportional to the square of the input size, described as quadratic time, or $O(n^2)$. Space complexity, similarly, measures how memory usage grows with input size.

In practice, simplifying Big-O expressions is key to understanding scalability. For instance, an algorithm requiring $(m + 1) \times (2n)$ steps has a complexity of $O(mn)$, as constant additions or multiplications (e.g., $+1$ or a factor like 2) are disregarded in Big-O notation. This abstraction helps focus on the dominant growth factors, which are crucial for evaluating algorithm performance on large-scale datasets.

### 1.2.3 Dynamic Programming

Dynamic programming is a method in computer science engineering used to solve problems by breaking them down into smaller, overlapping subproblems. A dynamic programming algorithm solves each subproblem only once and stores its result, thereby avoiding redundant computations and significantly improving efficiency. Solving the original problem then involves making choices among the solutions of subproblems to construct the final result.

This technique is particularly effective for solving optimization problems, where multiple solutions may exist, and the objective is to find one that optimizes a specific criterion, such as minimizing cost or maximizing efficiency. Dynamic programming provides a systematic approach to achieving this by:

1. Identifying the structure of an optimal solution,
2. defining the value of the solution recursively, and,
3. computing the value of the optimal solution iteratively in a bottom-up manner.

In a bottom-up approach, the values of optimal solutions to smaller subproblems are computed first and then combined to solve larger problems. If required, the actual solution can also be reconstructed from the computed information [23].

#### 1.2.3.1 Dynamic Programming in Sequence Alignment

In the context of sequence alignment, dynamic programming is used to pairwise align DNA, RNA, or protein sequences while accounting for sequencing errors, genetic vari-

ations, and structural differences like insertions or deletions (indels). The **edit distance** or **Levenshtein distance** is a fundamental metric for sequence alignment that measures how similar two strings are by calculating the minimum number of edits required to transform one string into the other.

---

**Example 1.8.** *To transform the string "`kitten`" into "`sitting`", three edits are needed: replace '`k`' with '`s`', replace '`e`' with '`i`', and add '`g`' at the end. This transformation is shown below. An 'X' denotes a mismatch or indel.*

```
s i t t i n g
X       X   X
k i t t e n
```

---

**Definition 1.8.** ***Edit Distance*** *(Levenshtein distance) The edit distance of two strings $X$ and $Y$ of length m and n, respectively, is defined by the smallest number of substitutions, insertions and deletions needed to transform $X$ into $Y$.*

The edit distance can be computed using a dynamic programming approach and can be seen as a minimization problem. To compute the edit distance between two strings $X$ and $Y$, of lengths $m$ and $n$, we leverage the idea that the solution to the full problem can be constructed from solutions to smaller subproblems (calculating the edit distance between prefixes of $X$ and $Y$). Specifically, aligning $\text{pre}_i(X)$ with $\text{pre}_j(Y)$ requires us to consider how the last characters of these prefixes align, while reusing the results from previously computed subproblems to minimize redundant calculations.

If the last characters $X[i]$ and $Y[j]$ are the same, then the edit distance between $\text{pre}_i(X)$ and $\text{pre}_j(Y)$ is the same as the edit distance between $\text{pre}_{i-1}(X)$ and $\text{pre}_{j-1}(Y)$. If they differ, a substitution is needed, increasing the cost by 1. Alternatively, if one of the characters is inserted (or deleted), the new edit distance corresponds to adding 1 to the edit distance between $\text{pre}_i(X)$ and $\text{pre}_{j-1}(Y)$ (or between $\text{pre}_{i-1}(X)$ and $\text{pre}_j(Y)$).

We can use a matrix $D$ of size $(m+1) \times (n+1)$ to store the edit distances. $D[i, j]$ then stores the edit distance between $\text{pre}_i(X)$ and $\text{pre}_j(Y)$. Note that the edit distance between a string of length $l$ and the empty string is equal to $l$. To find the edit distance between $X$ and $Y$, we can calculate $D[m, n]$ using the following recursive equation:

$$D[m,n] = \begin{cases} \max(m,n) & & \text{If } \min(m,n) = 0 \\ \min \begin{cases} D[m-1,n]+1 & \text{(deletion)} \\ D[m-1,n-1]+1 & \text{(insertion)} \\ D[m-1,n-1]+I & \text{(match/subst.)} \end{cases} & & \text{Otherwise} \end{cases} \tag{1.1}$$

where $I$ denotes the indicator function. It equals 1 if $X[m-1] \neq Y[n-1]$ and 0 otherwise. This recursive relationship is illustrated in Figure 1.7. The complexity of this algorithm is $O(mn)$.

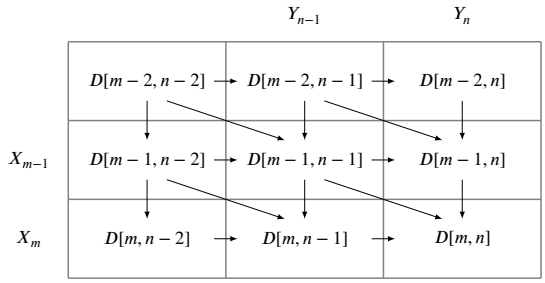


***Figure 1.7:*** *Illustration of the recursive relationship of the edit distance calculation. The value at a given matrix cell (e.g. $D[m,n]$ ) is computed using the values of three other cells: the cell above ($D[m-1,n]$ ), the cell to the left ($D[m,n-1]$) and the cell diagonally above and to the left ($D[m-1,n-1]$ ).*

---

**Example 1.9.** *Figure 1.8 shows the edit distance matrix between strings "`kitten`" and "`sitting`" from Example 1.8. The matrix represents the minimum edit distance calculations for transforming one string into the other. Arrows are drawn to indicate the direction of the computation for a given cell. Specifically, an arrow is drawn if the value in a cell is derived from the cell where the arrow originates. The arrows that form the optimal alignment path are emphasized.*

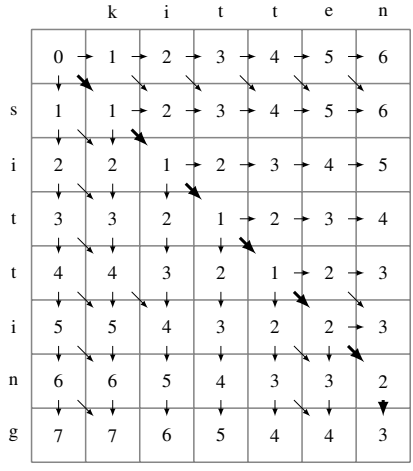


***Figure 1.8:*** *The edit distance matrix for strings "`kitten`" and "`sitting`".*

**Other Forms of Alignment** Edit distance itself is a form of global alignment, where the entire lengths of the two sequences are compared to determine the minimal number of edits needed to transform one sequence into the other. The Needleman-Wunsch algorithm [24] generalizes this concept by introducing customizable scores for matches, mismatches, and gaps (insertions or deletions), making it suitable for cases where alignment accuracy depends on biologically meaningful scoring schemes rather than just counting edits. In contrast, local alignment methods, such as the Smith-Waterman algorithm [25], focus on identifying regions of high similarity within two sequences, ignoring dissimilar portions. Affine gap penalties extend these approaches by distinguishing between the cost of opening a gap and the cost of extending it, enabling more realistic alignments for biological sequences. Hamming distance, on the other hand, measures the number of mismatches between two strings of equal length and does not account for insertions or deletions. It provides a simpler alternative to edit distance in scenarios where sequence lengths are fixed.

This dissertation focuses on edit distance and Hamming distance because they allow us to directly count the number of edits, which is a key requirement for search schemes, the central topic of this thesis.

**Definition 1.9.** *The **Hamming distance** between two strings $X$ and $Y$ of equal length $n$ is defined as the number of positions $i$ where $X[i] \neq Y[i]$. Formally:*

$$H(X, Y) = \sum_{i=1}^{n} I(X[i] \neq Y[i]),$$

*where $I$ is again the indicator function as defined on page 20.*

---

**Example 1.10.** *The Hamming distance between "`GATTACA`"and "`GACTATA`"is 3, as differences occur at positions 2, 4, and 7.*

---

### 1.2.4 Cache Memory

This dissertation focuses on developing fast and efficient computational approaches for sequence alignment, where speed is critical due to the massive scale of genomic data. Achieving high performance necessitates optimizing data access and processing at the hardware level. Notably, even when an algorithm's theoretical computational complexity remains unchanged, significant runtime improvements can be realized through effective utilization of cache memory.

To understand the concept of cache memory intuitively, imagine a student conducting research in a vast library. The library, with its extensive collection of books, represents the main memory, while the student's personal workspace within the library symbolizes the cache. When working on a specific topic, the student cannot keep all the library's books on their desk due to its limited size. Instead, they select the most relevant books and bring them to their workspace. This allows the student to access the needed information quickly, without repeatedly walking back to the shelves.

However, if the student realizes mid-task that a required book is not on their desk, they must retrieve it from the distant shelves, which takes considerably more time. Moreover, when retrieving a book, it is often efficient to bring other related books nearby, as there is a good chance they will be needed soon. Similarly, if a student frequently references the same book, even if it is for different pages, keeping it close by ensures their workflow remains smooth and uninterrupted. This approach mirrors the principles underlying cache memory, which is designed to minimize delays by storing frequently used data close to the processor.

Cache memory operates as a small, high-speed storage layer located near the Central Processing Unit (CPU), temporarily holding frequently accessed data and instructions. Although cache memory is typically organized into multiple layers (L1, L2, L3) with varying speeds and costs, a detailed exploration of this hierarchy is outside the scope of this dissertation. When a program requests data, the system first checks the cache. If the data is present — a scenario known as a **cache hit** — it is retrieved swiftly. Conversely, if the data is absent — a **cache miss** — the system must fetch it from the slower main memory, resulting in significantly higher latency. Depending on the hardware architecture and memory hierarchy, a cache miss can be 10 to 100 times slower than a cache hit.

Modern cache architectures are organized into cache lines, which represent the smallest units of data transferred between the cache and main memory. While most modern cache lines are typically 64 bytes in size, some architectures use cache lines of 32 or 128 bytes, depending on their design and performance requirements. When the CPU accesses a specific memory location, the entire cache line containing that address is loaded into the cache. This mechanism leverages the principle of *spatial locality*, as programs often access contiguous memory addresses. For instance, consider iterating through an array in memory. When the CPU accesses the first element, the entire cache line containing that element, along with nearby elements, is loaded into the cache. As a result, when the program proceeds to access the next element in the array, it is already present in the cache, eliminating the need to fetch it again from the slower main memory. This mechanism effectively reduces the frequency of cache misses

In addition to spatial locality, *temporal locality* — the tendency of programs to reuse recently accessed data — plays a significant role in cache efficiency. If data is reused shortly after being loaded into the cache, the CPU can access it directly without incurring the cost of fetching it from main memory again. Consequently, an efficient program is designed to optimize cache usage, structuring its memory access patterns to maximize cache hits and minimize the occurrence of costly cache misses, thereby improving overall performance. [26, 27].

### 1.2.5 Parallelization

Parallelization is the process of dividing a computational task into smaller, independent subtasks that can be completed simultaneously. Imagine a large field that needs to be harvested. If one person works alone, they must cover the entire field row by row. However, if multiple workers are each assigned a specific row, the work can be completed much faster since everyone contributes independently. In computing, this

division of labor allows complex tasks to be executed more efficiently by breaking them into manageable units and processing them concurrently.

Computational parallelization takes several forms, such as thread-level, bit-level, instruction-level, and distributed parallelism. This section focuses on thread-level parallelization and bit-level parallelization, as these are touched upon within this dissertation.

#### 1.2.5.1 Thread-Level Parallelization

Thread-level parallelization operates at a higher level, akin to dividing tasks among different workers in the field. Here, each "worker" is represented by a thread, an independent sequence of instructions that runs concurrently on a processor core. On modern multi-core processors, multiple threads can execute simultaneously, sharing the computational workload.

For sequence alignment, thread-level parallelization is particularly effective because the alignment of each query sequence, such as DNA or RNA reads, to a reference genome (e.g., a model organism's genome or a pan-genome) is independent of other alignments. Each thread can handle a single sequence alignment, allowing multiple alignments to proceed at the same time. For example, if there are four cores available, four alignments can be performed simultaneously, cutting down the time required to process large datasets significantly.

#### 1.2.5.2 Bit-Level Parallelization

Bit-level parallelization operates at the most granular level, exploiting the fact that a computer's CPU processes data in chunks called words, typically 32 or 64 bits wide. Just as a worker harvesting a field uses a scythe to cut multiple stalks of grain in one motion, bit-level parallelization allows a CPU to manipulate multiple bits of data simultaneously using a single instruction.

For instance, logical operators such as AND, OR, and XOR work on all the bits within a word at once. If we wanted to compare two 64-bit numbers, the CPU could process all 64 bits in parallel using a single operation. This capability is especially powerful for low-level tasks, such as bit masking, compact data storage, or performing bulk comparisons across sequences.

While threads handle independent subtasks, bit-level operations maximize efficiency within each thread by fully utilizing the CPU's ability to process data in wide chunks.

## 1.3 FM-Index

The FM-index is a pivotal data structure in bioinformatics, celebrated for its efficiency in compressing and searching large-scale genomic sequences. Proposed by Ferragina and Manzini in 2000 [28], it is built upon the Burrows-Wheeler Transform (BWT) [29], achieving both space efficiency and fast exact pattern matching. This makes it a foundational tool in sequence alignment and genomic data analysis pipelines.

In this thesis, we leverage the bidirectional FM-index [30], which facilitates flexible pattern searches in both forward and reverse directions. Bidirectionality is crucial for implementing search schemes (see Section 4.2).

This section explores the theoretical foundations and mechanics of the (bidirectional) FM-index, establishing its role in enabling efficient and accurate lossless approximate pattern matching.

### 1.3.1 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler Transform (BWT) is a reversible transformation that rearranges a text $T$ into a string BWT($T$) containing the same characters. The construction of BWT($T$) involves generating all cyclic rotations of $T$, sorting them lexicographically, and extracting the last character of each sorted rotation. To ensure unique sorting, a sentinel character \$, smaller than any character in $T$ and appearing exactly once, is appended to $T$. Algorithm 1 outlines this process, employing helper functions for cyclic rotation and lexicographic sorting. Example 1.11 illustrates this transformation. The matrix representation of the cyclic rotations reveals structural properties of the BWT. Specifically, the first column contains the characters of $T$ in lexicographic order, while the last column forms BWT($T$).

A defining property of the BWT is its ability to cluster identical characters. This occurs because repeated substrings in the original text lead to the same character often preceding suffixes that start with the same prefix, resulting in groups of identical characters in the transformed string BWT($T$).

**Algorithm 1:** Burrows-Wheeler Transform (BWT) Construction

**Input :** String $S$ of $n$ characters
**Output:** BWT($S$): the Burrows-Wheeler Transform of $S$

```
cyclicRotations ← [] // List to store cyclic rotations of S
for i ← 0 to n − 1 do
 | cyclicRotations.add(S.rotate(i))
end
// Sort rotations lexicographically
sortedRotations ← sortLexicographically(cyclicRotations)
BWT ← "" // Initialize BWT(S) as an empty string
for i ← 0 to n − 1 do
 | BWT.add(sortedRotations[i][n − 1])
end
```

#### 1.3.1.1 Relationship Between the BWT and the Suffix Array (SA)

The Burrows-Wheeler Transform can also be derived using the Suffix Array (SA). The suffix array SA($T$) lists the starting indices of all suffixes of $T$, ordered lexicographically. Instead of storing the suffixes themselves, SA($T$) stores their indices, making it space-efficient.

**Definition 1.10.** ***Suffix Array (SA)*** *The suffix array of a string $S$, denoted* SA($S$)*, is an array of length $|S|$ where each entry* SA($S$)[$i$] *contains the starting index of the $i$-th smallest suffix of $S$ in lexicographical order.*

---

**Example 1.11.** *Consider the string $S =$ "`banana$`". Below the cyclic rotations, their lexicographic ordering, and the resulting* BWT($S$) ="`annb$aa`" *are shown. The last column, which is the BWT, is underlined and highlighted.*

| | | |
|---|---|---|
| *banana$* | | *$banan**a*** |
| *anana$b* | | *a$banan**n*** |
| *nana$ba* | sort | *ana$ban**n*** |
| *ana$ban* | ⇒ | *anana$**b*** |
| *na$bana* | lexicographically | *banana**$*** |
| *a$banan* | | *na$ban**a*** |
| *$banana* | | *nana$b**a*** |

---

Given SA($T$), the $i$-th character of BWT($T$) is the character in $T$ (cyclically) preceding the suffix starting at SA($T$)[$i$]. This relationship can be formalized using Equation 1.2. Example 1.12 shows the BWT suffix array and the sorted suffixes for string "`banana$`".

$$\text{BWT}(T)[i] = \begin{cases} T[\text{SA}(T)[i]-1], & \text{if SA}(T)[i] > 0, \\ \$, & \text{otherwise.} \end{cases} \tag{1.2}$$

---

**Example 1.12.** *Table 1.4 demonstrates* SA($S$)*,* BWT($S$)*, and their connection for $S =$"`banana$`". For instance,* BWT($S$)[3] = $S$[SA($S$)[3] − 1] = '`b`'.

***Table 1.4:*** *Relation between* SA($S$)*,* BWT($S$)*, and the suffixes of $S =$"`banana$`".*

| Index | SA($S$) | BWT($S$) = $L$ | Suffix |
|---|---|---|---|
| 0 | 6 | a | $ |
| 1 | 5 | n | a$ |
| 2 | 3 | n | ana$ |
| 3 | 1 | b | anana$ |
| 4 | 0 | $ | banana$ |
| 5 | 4 | a | na$ |
| 6 | 2 | a | nana$ |

---

## 1.3.2 From BWT and Suffix Array to FM-Index

The LF Mapping is a key property of the BWT that enables efficient navigation through the transformed representation. This property is crucial for efficiently reconstructing the original text from the BWT, as well as for enabling search operations without explicitly reversing the transform.

**Definition 1.11.** ***LF Property*** *Consider the matrix $M$ that holds the lexicographically sorted cyclic rotations of the string $S$. The $i$th occurrence of character $c$ in the last column of $M$ and the $i$th occurrence of $c$ in the first column of $M$ correspond to the same character in $S$.*

Consider again the matrix $M$ of all cyclic rotations of a string $T$, sorted lexicographically. The *first column* ($F$) of $M$ contains the characters of $T$ arranged lexicographically according to their right contexts. The *last column* ($L$) of $M$ corresponds to the BWT of $T$, denoted BWT($T$).

The LF Mapping states that the $i$-th occurrence of a character $c$ in the last column ($L$) corresponds to the $i$-th occurrence of $c$ in the first column ($F$). The property is illustrated in Example 1.13 This mapping allows us to move between the columns of the matrix $M$, effectively tracing the position of each character back to its original context in $T$.

**Example 1.13.** *Figure 1.9 illustrates the LF Property for the string $S =$"`banana$`". Observe that the order of the '`a`' characters ($a_2, a_1, a_0$) remains consistent between the first column ($F$) and the last column ($L$). Similarly, the order of the '`n`' characters ($n_1, n_0$) also remains consistent across both columns.*

| $F$ | | | | | | $L$ |
|---|---|---|---|---|---|---|
| $\$_0$ | $b_0$ | $a_0$ | $n_0$ | $a_1$ | $n_1$ | $a_2$ |
| $a_2$ | $\$_0$ | $b_0$ | $a_0$ | $n_0$ | $a_1$ | $n_1$ |
| $a_1$ | $n_1$ | $a_2$ | $\$_0$ | $b_0$ | $a_0$ | $n_0$ |
| $a_0$ | $n_0$ | $a_1$ | $n_1$ | $a_2$ | $\$_0$ | $b_0$ |
| $b_0$ | $a_0$ | $n_0$ | $a_1$ | $n_1$ | $a_2$ | $\$_0$ |
| $n_1$ | $a_2$ | $\$_0$ | $b_0$ | $a_0$ | $n_0$ | $a_1$ |
| $n_0$ | $a_1$ | $n_1$ | $a_2$ | $\$_0$ | $b_0$ | $a_0$ |

***Figure 1.9:*** *The lexicographically sorted rotations of the string $S =$"`banana$`" represented in matrix form. The characters are labeled with their respective ranks to distinguish between multiple occurrences of the same character. Columns F and L represent the first and last columns of the matrix.*

To map a character $c$ at index $i$ in the $L$ column to its corresponding position in the $F$ column, we first count the number of occurrences of $c$ in $L[0, i[$ (i.e. the number of $c$ occurrences *before* $L[i]$). Let this count be $n$. $L[i]$ then contains the $n+1$th occurrence of $c$ in $L$. In the $F$ column, which is lexicographically sorted, all occurrences of $c$ are grouped together. The LF-property states that the $i$th occurrence of $c$ in $L$ corresponds to the $i$th occurrence in $F$. Therefore, we locate the first occurrence of $c$ in $F$ and move forward $n$ positions to find the $n + 1$th occurrence that maps to the character at $L[i]$.

By repeatedly applying this mapping, we can traverse the text in reverse order and reconstruct portions of the original text starting from any position in the BWT. The intuition behind this is straightforward: when we are in the $F$ column at a particular

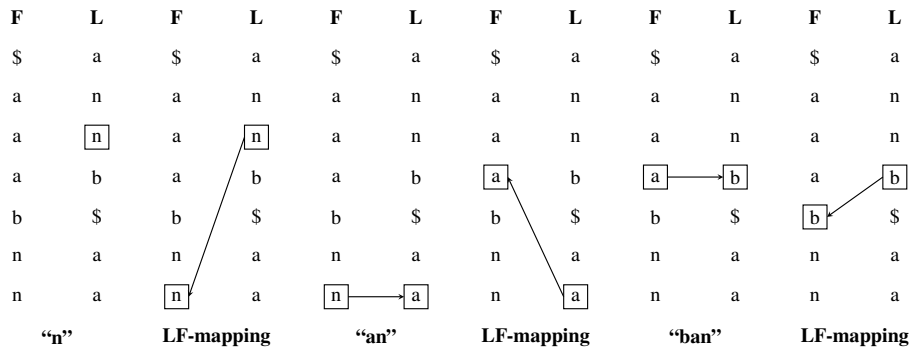


***Figure 1.10:*** *Illustration of LF-mapping and backward search for the string reconstruction process. Starting from the L-column, the LF-mapping property is used to traverse the BWT and locate characters in the lexicographically sorted F-column. This enables the reconstruction of substrings such as "*`n`*", "*`an`*", and "*`ban`*" in a step-by-step manner. Each step highlights the mapping between corresponding positions in L and F, with arrows and boxes indicating transitions.*

position, we can see what character came before it by looking at the $L$ column at the same position. By repeatedly applying the LF mapping, which links the character in $L$ to its corresponding position in $F$, we effectively "walk backward" through the text. This allows us to reconstruct the original text or a portion of it in reverse order. Figure 1.10 illustrates this principle by backwards reconstructing substring "`ban`", starting from an '`n`' character in the BWT ($L$ column).

However, to perform this backward navigation efficiently, we need to avoid recomputing the count of occurrences each time we map a character from $L$ to $F$. Instead, we precompute auxiliary structures, known as the **count table** ($C$) and the **occurrence table** (Occ). $C$ provides information about the cumulative counts of characters lexicographically smaller than a given character. This allows us to quickly locate the starting index of each character's block in the $F$ column. The occurrence table tracks how many times a specific character has occurred in the BWT ($L$) up to a particular position. This information helps in efficiently applying the LF-mapping by determining the exact rank of a character in $F$.

**Definition 1.12.** *The* ***count table*** $C$ *is an array of length* $|\Sigma|$*, where* $\Sigma$ *is the alphabet of the string* $T$*. Each entry* $C[c]$ *stores the total number of characters in* $T$ *that are lexicographically smaller than the character* $c$*. Formally, for all* $c \in \Sigma$*,*

$$C[c] = |\{i \mid T[i] < c,\ 0 \leq i < |T|\}|.$$

**Definition 1.13.** *The* ***occurrence table*** Occ *is a two-dimensional array of size* $|T| \times |\Sigma|$ *where* $\mathrm{Occ}[i, c]$ *represents the number of times character* $c$ *appears in* $L[0, i[$*. Formally,*

$$\mathrm{Occ}[i, c] = |\{j \mid L[j] = c,\ 0 \leq j < i\}|.$$

**Example 1.14.** *Table 1.5 presents the C and* Occ *tables for string "*`banana$`*", for which the BWT is "*`annb$aa`*". Note that the header row with characters is shown for clarity but is not actually stored in practice. For example, C['n'] = 5, because there are 5 characters in the string that are lexicographically smaller than 'n' (1 × '$', 3 × 'a' and 1 × 'b'). Similarly, Occ*[4, 'a'] *= 1 since there is exactly one occurrence of 'a' BWT*[0, 4[ *="*`annb`*".*

***Table 1.5:*** *The C and* Occ *tables for string "*`banana$`*", with BWT="*`annb$aa`*". Left: C Table; Right:* Occ *Table.*

| c | '$' | 'a' | 'b' | 'n' |
|---|---|---|---|---|
| C[c] | 0 | 1 | 4 | 5 |

| '$' | 'a' | 'b' | 'n' |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 2 |
| 1 | 1 | 1 | 2 |
| 1 | 2 | 1 | 2 |
| 1 | 3 | 1 | 2 |

With the count table and the occurrence table precomputed, we can now define the LF-mapping operation, which allows us to move from an index $i$ in the $L$ column (or BWT) to the corresponding index $j$ in the $F$ in constant time.

**Lemma 1.1.** *Given an index i in the BWT, the LF-mapping operation* LF($i$) *is computed in constant time as:*

$$\text{LF}(i) = C[\text{BWT}[i]] + \text{Occ}[i, \text{BWT}[i]].$$

*This operation maps the position in the L-column of the character at index i to its corresponding position in the F-column.*

**Lemma 1.2.** *Given a character c and an index i in the BWT, the LF-mapping operation* LF($c, i$) *is computed in constant-time as:*

$$\text{LF}(i, c) = C[c] + \text{Occ}[i, c].$$

*This operation maps the position in the L-column of the last occurrence of c strictly before index i to its corresponding position in the F-column.*

#### 1.3.2.1 Exact Pattern Matching

Finding all exact occurrences of a pattern $P$ in a text $T$ can be achieved in $O(|P| + o)$ time, where $o$ is the number of occurrences. This involves applying the LF-property $|P|$ times, followed by $o$ look-ups in the suffix array. Similar to reconstructing parts of

the original text (see Figure 1.10), exact matching proceeds from right to left. To match a pattern $P$, begin by identifying the range $r = [b, e[$ of $P[|P| - 1]$, the last character of $P$, in the $F$-column. This range corresponds to the rotations/suffixes of the indexed string $T$ that start with this character. We can also denote this as $\mathcal{S}(P[|P| - 1], T)$. The equivalent range in the $L$-column represents all characters that precede $P[|P| - 1]$ in $T$.

The LF-mapping operation is then employed to determine the next range, $r' = [b', e'[ = \mathcal{S}(\text{suf}_{|P|-2}(P), T)$, which corresponds to all suffixes of $T$ that are prefixed by $\text{suf}_{|P|-2}(P)$. To compute the new range $r'$ in the $F$-column, we perform the LF-mapping operation for the bounds of $r$ with character $c = P[|P| - 2]$. This gives $b' = \text{LF}(b, c)$ and $e' = \text{LF}(e, c)$. Since $e$ is exclusive in the half-open interval $[b, e[$, $\text{LF}(c, e)$ maps to the first occurrence of $c$ in the $F$-column that, in the $L$-column, corresponds to a position occurring after the original range $[b, e[$. Repeating this process iteratively for each character of $P$, moving from right to left, refines the range in the $F$-column that corresponds to suffixes in $T$ matching the progressively longer suffixes of $P$.

If at any step the computed range becomes empty (i.e., $b = e$), it indicates that no suffixes in $T$ match $P$, and the process terminates early. Conversely, if the range is non-empty after processing all $|P|$ characters, the final range $[b'', e''[ = \mathcal{S}(P, T)$ represents the suffix array indices of all occurrences of $P$ in $T$. The number of occurrences $o$ is given by $o = e'' - b''$, and the corresponding positions in $T$ can be retrieved using $\text{SA}[b''], \text{SA}[b''+1], \ldots, \text{SA}[e''-1]$. A pseudo-code implementation of exact matching using the FM-index is provided in Algorithm 2.

**Algorithm 2:** Exact Matching in the FM-index

```
Input: Pattern P
Output: An array of positions in S where instances of pattern P are found
def matchexact (P):
    // Initialize range of last character in BWT
    c ← P[|P| − 1]
    r ← [C[c], C[Σ.nextCharacter(c)][
    // Iterate over the characters in P in reverse order
    for i ← |P| − 2 to 0 do
        // Calculate the new range using the LF property
        r ← [LF(r.begin, P[i]), LF(r.end, P[i])[
        if r.begin == r.end then
            // Early exit
            return []
    m ← []
    for i ← r.begin to r.end − 1 do
        m.add(SA[i])
    return m
```

**Example 1.15.** *In Figure 1.11, exact matching is demonstrated for the patterns "*ana*" (above) and "*aba*" (below) in the indexed string $T =$ "*banana$*". The search begins with the rightmost character of each pattern, using the $C$-table to locate the initial range $r$ in the $F$-column corresponding to suffixes of $T$ prefixed by the character.*

*For the pattern "*ana*", the initial character '*a*' identifies the range $r = [1, 4[$, corresponding to all suffixes of "*banana$*" that begin with '*a*'. Using the LF-mapping operation with $c =$'*n*', the range is refined.*

*The half-open interval convention ensures that* $\mathrm{LF}(c, b)$ *maps the first occurrence of '*n*' in the $L$-column within $[b, e[$ to the $F$-column, while* $\mathrm{LF}(c, e)$ *maps the first occurrence of '*n*' after $[b, e[$ in $L$ to the $F$-column. This refinement produces the new range $r' = [5, 7[$, representing $\mathcal{S}($"*na*", $T)$. Repeating this step for the final '*a*' in the pattern results in $r'' = [2, 4[$, which identifies $\mathcal{S}($"*ana*", $T)$. Consulting the suffix array reveals that "*ana*" occurs at positions 3 and 1 in the original text.*

*For the pattern "*aba*", the process begins by identifying the range for the rightmost character '*a*' as $[1, 4[$. Applying the LF-mapping operation with $c =$'*b*' on this range gives $r' = [4, 5[$. However, the next LF-mapping step with $c =$'*a*' results in an empty range, indicating that no suffixes of "*banana$*"are prefixed by "*aba*". Thus, "*aba*" does not occur in the original text.*

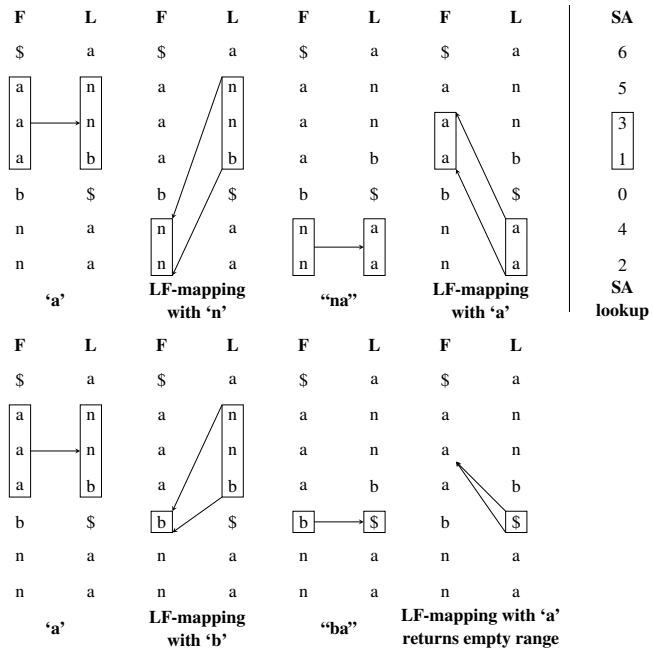


***Figure 1.11:*** *Finding exact occurrences of patterns "*ana*" (top) and "*aba*" (bottom) in the indexed string "*banana$*" via repeated LF-mapping operations. Exact occurrences of "*ana*" are found at start positions 3 and 1 in the original text. No exact occurrences of "*aba*" are found.*

### 1.3.3 Naive Approximate Pattern Matching in an FM-Index

Approximate pattern matching utilizing an FM-index—or any substring index—allows for the efficient detection of all instances where a pattern $P$ appears in a text $T$ with up to $k$ errors. Rather than directly aligning $P$ to each substring of $T$, which would require extensive computations, the FM-index offers a structured approach to navigate potential matches. This is achieved by leveraging its associated suffix array and the Burrows-Wheeler Transform (BWT).

This method of approximate pattern matching is well represented as a backtracking traversal through the search trie, a conceptual framework for organizing all substrings of $T$. Each node in this trie corresponds to a substring $S$ of $T$, where the node's depth indicates the length of $S$. The substring $S$ is determined by the reversed path from the root to the node. The search trie is thus a suffix tree of the reversed text $T^r$. The reversal is necessary as the FM-index allows only backward extensions. Each node is associated with a range $r = \mathcal{S}(S, T) = [b, e[$ over the suffix array, representing all suffixes of $T$ that begin with $S$. For each node at depth $d$, its children at depth $d + 1$ represent substrings $S'$ formed by prepending a character $c$ to $S$. The range for the child node with associated substring $cS$ is computed through the LF-mapping operation as $r' = \mathcal{S}(cS, T) = [LF(b, c), LF(e, c)[$. If $r'$ is empty, the node does not exist since no substring of $T$ equals $cS$. Note, that for this purpose we do not consider extensions with the sentinel character.

During search trie traversal, each visited node receives a Hamming distance score or a row of the edit distance matrix, depending on whether Hamming distance or edit distance is used (see Section 1.2.3.1). Under the Hamming distance, the score at a node depth $d$ reflects the Hamming distance between $S$ and the substring $P[|P| - d, |P|[$ of the pattern $P$. This score is calculated as the score of the parent node plus an increment $I$, where $I$ is 1 if $P[|P| - d] \neq S[0]$ and 0 otherwise. For edit distance, each node at depth $d$ is assigned a filled-in row in the edit distance matrix. This matrix aligns the reversed pattern $P$ horizontally and the reversed substring $S$ vertically. The new row at depth $d$ is derived using dynamic programming, based on the row of the parent node, the character $c = S[0]$ and $P$.

The traversal begins at the root node, representing the empty string, with an initial range $r = [0, |T|]$. At the start, the Hamming distance is zero, and for edit distance, the first row of the matrix is sequentially initialized from 0 to $|P|$, corresponding to the number of insertions needed to transform the empty string into each prefix of the reverse of $P$. As the search progresses, each child node is visited in a depth-first manner. Each child node represents a potential backwards extension of $S$, prepending one character at a step. If the score at a node exceeds $k$, or in the case of edit distance, if the minimum value in the current row exceeds $k$, the search along that branch is halted, and the algorithm backtracks to explore any remaining unexplored siblings of earlier ancestors.

When the traversal reaches a node at the depth $|P|$ with a Hamming distance $h \leq k$, the range $r$ at that node corresponds to a subset of all suffixes of $T$ that match $P$ within $h$ errors. These matches are retrieved using the suffix array to pinpoint their start positions in $T$ and the search again backtracks. For edit distance, a match is

confirmed if a node at depth $d$ is reached where the matrix value $D[d, |P|] \leq k$, indicating that the substring represented by this node can be transformed into $P$ with $k$ or fewer edits. Matches are similarly located in $T$ using the suffix array, ensuring a systematic exploration and identification of all potential matches. If we reach a node at depth $d > |P| + k$ the search can also safely backtrack, as no more matches with $k$ or fewer edits can be found along this branch. Algorithms 3 and 4 show pseudo-code for these naive backtracking approaches for the Hamming and edit distances. Example 1.16 on page 35 shows an approximate pattern matching procedure for the Hamming distance in the search trie.

**Algorithm 3:** Approximate Pattern Matching in the FM-index with Hamming Distance

**Input:** Text $T$, Pattern $P$, Maximum errors $k$, FM-index of $T$ (with alphabet $\Sigma$, suffix array SA and LF-mapping operation) $\mathcal{I}$
**Output:** Vector of positions and hamming distance scores of substrings in $T$ matching $P$ within $k$ errors

```
Node:Each node n has: r (range over suffix array with
     begin b and end e), d (depth)
Function ApproximateMatch(T, P, k, I):
    root ← Node([0, |T|], 0)                    // Initialize root node
    v ← empty vector        // Vector to store reported positions
    I.DFS_Hamming (root, 0, v, P)               // Begin DFS traversal
    return v                              // Return collected positions
Function DFS_Hamming(n, h, v, P):
    Doc:n: current node, h: Hamming distance, v: vector
        to store positions, P: the pattern
    if n.d = |P| then
        foreach i ∈ n.r do
            v.add ([SA[i], h])  // Add position and score to vector
        return
    foreach c ∈ Σ do
        r' ← [LF(n.r.b, c), LF(n.r.e, c)[
        if r'.b ≠ r'.e then
            i ← 1 if P[|P| − (n.d + 1)] ≠ c else 0
            h' ← h + i                          // Increment if mismatch
            if h' ≤ k then
                child ← Node(r', n.d + 1)          // Create child node
                DFS_Hamming(child, h', v, P)         // Recursive call
```

**Algorithm 4:** Approximate Pattern Matching in the FM-index with Edit Distance

**Input:** Text $T$, Pattern $P$, Maximum errors $k$, FM-index of $T$ (with alphabet $\Sigma$, suffix array SA and LF-mapping operation) $\mathcal{I}$

**Output:** Vector of tuples (position, length, edit distance) in $T$ matching $P$ within $k$ edits

```
Function ApproximateMatch_EditDistance(T, P, k, FM):
    root ← Node([0, |T|], 0)                    // Initialize root node
    v ← empty vector                            // Vector to store results
    D ← matrix of size (|P| + k + 1) × (|P| + 1) with all elements set to 0
    for j ← 0 to |P| do
        D[0][j] ← j                             // Set top row with increasing values
    I.DFS_Edit (root, D, v, P, k)               // Begin DFS traversal
    return v                                    // Return collected tuples
Function DFS_Edit(n, D, v, P, k):
    Doc:n: current node, D: edit distance matrix, v:
        result vector, P: pattern, k: max edits
    if D[n.d][|P|] ≤ k then
        foreach i ∈ n.r do
            v.add ((SA[i], d, D[n.d][|P|] ))    // Add match to vector
    if n.d < |P| + k then
        foreach c ∈ Σ do
            r' ← [LF(n.r.b, c), LF(n.r.e, c)[
            if r'.b ≠ r'.e then
                d' ← n.d + 1
                D[d'][0] ← d'                   // Initialize first column
                for j ← 1 to |P| do
                    // DP update: insertion, deletion or
                       substitution
                    i ← 1 if P[|P| − (n.d + 1)] ≠ c else 0
                    D[d'][j] ←
                      min (D[n.d][j] + 1, D[d'][j − 1] + 1, D[n.d][j − 1] + i)
                if min(D[d']) ≤ k then
                    child ← Node(r', d')        // Create child node
                    DFS_Edit(child, D, v, P, k) // Recursive call
```

**Example 1.16.** *Let the indexed string $T$ be "mississippi$" and the pattern $P$ be "ippi". The BWT and SA of $T$ are respectively "ipssm$pissii" and* $[11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$. *For approximate pattern matching with at most $k = 2$ errors under the Hamming distance, the traversal of the search trie is shown in Figure 1.12.*

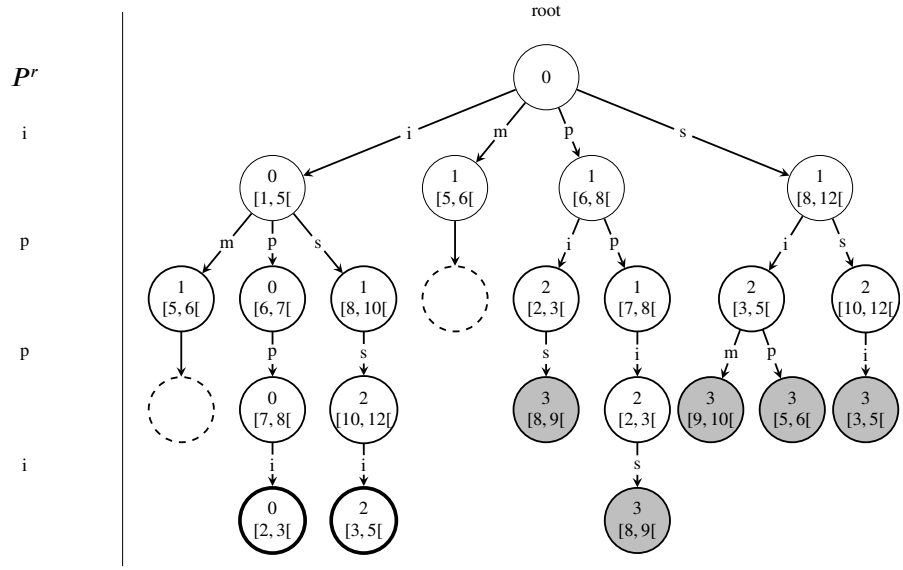


***Figure 1.12:*** *Traversal of the search trie for the indexed string "mississippi" and pattern $P$ ="ippi", for approximate pattern matching with at most $k = 2$ errors under Hamming distance. Each drawn node contains the current Hamming distance score and the associated SA range. A dashed node corresponds to an extension with the sentinel character, not visited during the search trie traversal, a greyed-out node means it exceeded $k = 2$ and the search backtracked, and a node with a thick hull indicates the reporting of an occurrence.*

This naive backtracking approach explores all possible alignments of $P$ within the constraints set by $k$. While it ensures all matches are found, the computational cost can become prohibitive as $P$, $T$, and $k$ increase. The **search space** is equal to the number of nodes that are explored in the search trie. Specifically for edit distance, but to a lesser extent also for the Hamming distance, the branching possibilities multiply exponentially with $k$ due to each permissible error, leading to many unproductive branches and wasted computations. Hence, while feasible for small patterns or low values of $k$, more efficient search strategies are required for scalable approximate pattern matching in larger texts and patterns, as discussed in upcoming sections. This naive approach, however, lays a fundamental groundwork for understanding approximate matching mechanisms in FM-indexes.

#### 1.3.3.1 Banded Alignment

In Algorithm 4, we can optimize space and the number of computations by replacing the matrix $D$ with a banded matrix of width $2k + 1$, where $k$ is the maximum number of allowed errors. This restricts computations to a narrow band around the diagonal, as any valid alignment must stay within $k$ insertions, deletions, or substitutions from the diagonal, which represents an exact match.

We can safely initialize the elements to the left of the band with $k + 1$. During the algorithm, we update only the elements $D[i, j]$ that lie within the band, defined as $|i-j| \leq k$. For a node at depth $d$, before checking if $D[d, |P|] < k$, we first verify that $|d - |P|| \leq k$. This approach ensures correctness by restricting computations to the feasible alignment region, avoiding unnecessary calculations for out-of-band entries.

### 1.3.4 Memory Requirements

The FM-index for a string $S$ of length $n$ comprises several components. Numbers are assumed to be stored using $\lceil \log n \rceil$ bits. For most genomes, 32 bits suffice, while exceptionally large genomes may require 64 bits. Consequently, operations like copying and arithmetic are considered to run in constant time, and the contribution of $\lceil \log n \rceil$ is excluded from the complexity analysis. The FM-index exists out of four components, **The Alphabet** $\Sigma$, **The Counts Array** $C$, **The Occurrences Matrix** Occ and **The Suffix Array** SA($S$). $\Sigma$ is mapping of characters to their ranks in the alphabet. The memory complexity is $O(|\Sigma|)$. $C$ is an array of integers of length $|\Sigma|$. Its memory complexity is also $O(|\Sigma|)$. Occ is a matrix with $n$ rows and $|\Sigma|$ columns. The memory complexity is $O(n|\Sigma|)$. Lastly, SA($S$) is an array of $n$ integers. The memory complexity is $O(n)$. In total, the memory complexity of the FM-index is $O(n|\Sigma|)$. If $|\Sigma|$ is significantly smaller than $n$, this simplifies to $O(n)$.

---

**Example 1.17.** *The human genome can be stored efficiently using the FM-index. The genome consists of approximately 3 billion base pairs. The memory requirements for the FM-index are calculated as follows:*

- ***Alphabet*** *(*$\Sigma$*):* $5 \times 8\,\text{bit} = 5\,\text{B}$
- ***Counts Array*** *(*$C$*):* $5 \times 32\,\text{bit} = 20\,\text{B}$
- ***Occurrences Matrix*** *(*Occ*):* $3{,}000{,}000{,}000 \times 5 \times 32\,\text{bit} = 60\,\text{GB}$
- ***Suffix Array*** *(*SA*):* $3{,}000{,}000{,}000 \times 32\,\text{bit} = 12\,\text{GB}$

  ***Total:*** $\sim 72\,\text{GB}$

*In this context, we assume that characters are represented using the ASCII alphabet, requiring 8 bits per character, and that the genome contains only* `ACGT` *characters. The memory requirements for the alphabet and counts array are insignificant compared to those of the suffix array and the occurrences matrix.*

---

#### 1.3.4.1 Improvements

As highlighted in Example 1.17, the largest memory consumption in the FM-index comes from the suffix array (SA) and the occurrences table (Occ). Significant memory savings can be achieved by optimizing these two structures.

**Suffix Array** One approach to reduce the memory usage of the suffix array is to store a sparse version, retaining only every $f$-th element. This reduces the memory requirement by a factor of $f$. If the requested index $i$ satisfies $i \bmod f = 0$, the value is directly accessible in $O(1)$ time. Otherwise, the LF property is used to compute $\text{SA}[i]$. Specifically, the index $i'$ is found such that $\text{SA}[i'] = \text{SA}[i] - 1$. If $i' \bmod f = 0$, the stored value $\text{SA}[i']$ is used to compute $\text{SA}[i]$. If not, the LF property is applied repeatedly (say $q$ times) until an index $i''$ is found where $i'' \bmod f = 0$. At this point, $\text{SA}[i]$ is computed as $\text{SA}[i''] + q$. While this increases the worst-case time complexity to $O(n)$, the memory usage is significantly reduced.

Another method involves storing only every $f$-th suffix explicitly and using a bit vector to indicate which indices are stored. If $\text{SA}[i]$ is stored, its position in the sparse array can be efficiently determined using a `rank` operation, which takes $O(1)$ time. Otherwise, the LF property is applied repeatedly to find a stored index and compute $\text{SA}[i]$. In the worst case, $\text{SA}[i]$ may require $f - 1$ LF operations, such as when $i = \alpha f - 1$. In the best case, where $i = \alpha f$, no LF operations are needed. Assuming uniform access probabilities, the average number of LF applications per suffix is $(f - 1)/2$, increasing the time complexity to $O(f)$ on average while reducing the memory requirement by a factor of $f$. The choice of the sampling factor $f$ thus represents a trade-off between space and time, where larger values of $f$ reduce memory usage but increase query time.

**Occurrences Table** The memory footprint of the Occ table can be significantly reduced using the rank9 data structure proposed by Vigna [31]. The purpose of the Occ table is to answer queries about the number of occurrences of a character $c$ before a given index $i$ in the BWT. For each character $c$, a bit vector $\text{bv}_c$ is constructed as follows:

$$\text{bv}_c[i] = \begin{cases} 1 & \text{if BWT}[S][i] = c, \\ 0 & \text{otherwise.} \end{cases} \tag{1.3}$$

The number of occurrences of $c$ before index $i$ is the sum over the range $[0, i)$ of the bit vector $\text{bv}_c$, referred to as the rank of $\text{bv}_c$ at $i$:

$$\texttt{rank}(\text{bv}_c, i) = \sum_{j=0}^{i-1} \text{bv}_c[j].$$

The rank9 data structure enables $O(1)$ time computation of this rank, regardless of $i$, using a three-stage approach with a 25% memory overhead. The bit vector $\text{bv}_c$ is

treated as an array of 64-bit words. The bit at index $i$ in $\text{bv}_c$ is then located at position ($i$ mod 64) in word $\lfloor i/64 \rfloor$.

The data structure divides $\text{bv}_c$ into basic blocks of 512 bits (8 words). For each block, two additional 64-bit words are stored: the *first-level count*, which is the total number of 1-bits before the start of the block, and the *second-level counts*, which represent the cumulative number of 1-bits before the current word in the block. Of course, for the first word in the block, the second level count is always zero, thus only 7 second-level counts need to be stored. For the $k$-th word ($1 \leq k < 8$), the second-level count is computed as:

$$\texttt{rank}(\text{bv}_c, p + 64k) - \texttt{rank}(\text{bv}_c, p),$$

where $p$ is the starting index of the block. The maximal value of a second-level count is found at word 7. This value is maximal if the first 7 words all contain 1-bits. In this case, the second-level count will be $7 \times 64 = 448$, which can be stored using 9 bits. Hence, all 7 second-level counts fit into a single 64-bit word ($7 \times 9 = 63 < 64$). The second-level count for the second word occupies the 9 least significant bits, the most significant bit of the word is then left open.

The first- and second-level counts are stored in an interleaved array called $\text{counts}_c$, which allows `rank` evaluations with at most two cache misses: one to access the counts array and one to access the bit vector itself. For any index $r$, the rank is computed as the sum of the first-level count (total 1-bits before the block containing $r$), the second-level count (total 1-bits within the words in the block before the word that contains $r$), and a popcount operation that counts the 1-bits between the start of the word containing $r$ and $r$ itself.

Accessing the first-level count is a lookup operation in the $C$ array. The second-level count is more complicated, as it should only be added for positions that are not in the first word of the block. Vigna showed that this can be done branchless by using bit operations [31].

Popcount operations are hardware-accelerated and operate in constant time. The bitmask for the popcount is created by shifting the word starting at $\lfloor r/64 \rfloor$ by ($r$ mod 64) positions. Because $r$ can be a multiple of 64, a two-step shift is used to ensure correctness:

$$\text{mask} = (\text{word} \ll 1) \ll (63 - (r \bmod 64)).$$

The rank9 data structure improves the memory efficiency of the Occ table, transforming it into an array of $|\Sigma|$ rank9 structures. Each structure consists of a bit vector of length $n$ (the length of the string) and a $\text{counts}_c$ array containing $2\lceil n/(8 \times 64) \rceil$ 64-bit entries.

Other strategies to reduce the memory footprint of the Occ table include applying a sampling technique similar to that used for the suffix array or employing wavelet trees [32].

**Example 1.18.** *Consider Figure 1.13, where the rank at index r is requested. Index r is located in word 5 of basic block 2 of* $\text{bv}_c$*. The first-level count, shown in light gray, represents the sum of all 1-bits in blocks 0 and 1 (range* $[0, p)$*). The second-level count, shown in dark gray, is the sum of 1-bits within words 0 to 4 of block 2 (range* $[p, q)$*). The requested rank is the sum of the first- and second-level counts, plus the number of 1-bits between q and r (computed via popcount).*

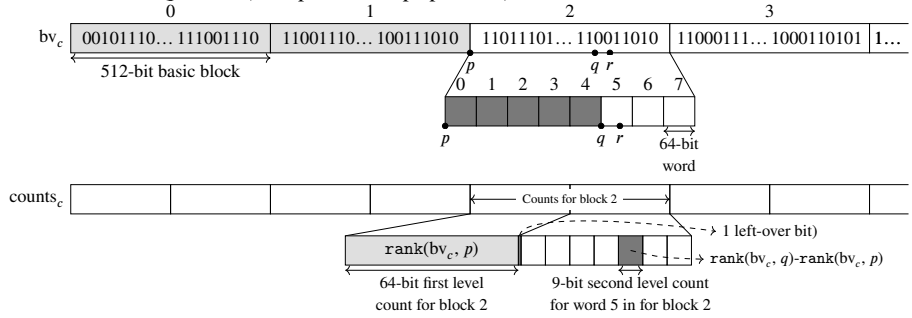


***Figure 1.13:*** *Visualization of the rank9 data structure for the example.* $\text{bv}_c$ *is split into basic blocks of 8 64-bit words. Counts are stored in an interleaved manner as a 64-bit word array.*

### 1.3.5 Bidirectional FM-Index

In 2009, Lam et al. introduced the bidirectional FM-index, an extension of the unidirectional FM-index that supports both forward and backward pattern extensions [30]. This is achieved by also storing the Occ table for BWT[$T^r$], the Burrows-Wheeler Transform of the reversed text $T$ (denoted as $T^r$). The additional table is referred to as $\text{Occ}^r$. A bidirectional FM-index maintains two synchronized ranges: $[i, j[$ over BWT[$T$] (or equivalently over SA($T$)) and $[i', j'[$ over BWT[$T^r$] (or equivalently over SA($T^r$)). This synchronization enables extending a pattern $P$ both backward to $cP$ and forward to $Pc$.

Let $\mathcal{S}(P, T)$ represent the range over SA($T$) containing all suffixes of $T$ that are prefixed by the pattern $P$. Similarly, let $\mathcal{S}(P^r, T^r)$ represent the range over SA($T^r$) containing all reversed prefixes of $T$ (equivalently, suffixes of $T^r$) that are prefixed by $P^r$. With a bidirectional FM-index, we simultaneously track both $\mathcal{S}(P, T)$ and $\mathcal{S}(P^r, T^r)$. The core challenge lies in synchronizing these two ranges during operations. The bidirectional FM-index supports two primary extension operations: the backwards extension, computing $\mathcal{S}(cP, T)$ and $\mathcal{S}(P^r c, T^r)$, and a forwards extension, computing $\mathcal{S}(cP^r, T^r)$ and $\mathcal{S}(Pc, T)$. Bidirectional forward and backward extensions are analogous and the bidirectional forward extension is detailed below.

To compute $\mathcal{S}(cP^r, T^r)$, we use a process analogous to the backward extension in the unidirectional FM-index of the reversed text. This involves the $\text{Occ}^r$ table and the LF-mapping operation. However, calculating the corresponding range $\mathcal{S}(Pc, T)$ is less straightforward. Given the range $\mathcal{S}(P, T) = [s, e[$, the range $\mathcal{S}(Pc, T)$ becomes a subrange $[s + x, s + x + y[$. Here, $x$ is the number of suffixes of $T$ prefixed by $Pd$

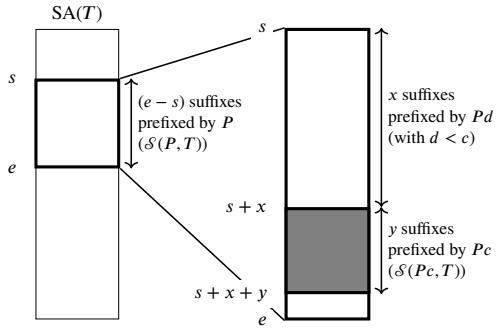


***Figure 1.14:*** *Finding $\mathcal{S}(Pc, T)$, given $\mathcal{S}(P, T)$.*

for $d < c$, while $y$ is $|\mathcal{S}(Pc, T)|$ (the number of suffixes prefixed by $Pc$). Figure 1.14 shows this principle.

To compute $x$ and $y$, we use the property that the number of suffixes of $T$ prefixed by $Pa$ equals the number of suffixes of $T^r$ prefixed by $aP^r$: $(|\mathcal{S}(Pa, T)| = |\mathcal{S}(aP^r, T^r)|)$. With the ranges $\mathcal{S}(P^r, T^r) = [s', e'[$ and $\mathcal{S}(cP^r, T^r)$ available, these values are derived as follows:

$$x = \sum_{d<c, d\in\Sigma} \mathrm{Occ}^{\mathrm{r}}[e', d] - \mathrm{Occ}^{\mathrm{r}}[s', d], \quad y = |\mathcal{S}(cP^r, T^r)|. \tag{1.4}$$

Calculating $x$ requires a cumulative summation over the alphabet. The time complexity to calculate $x$ is $O(|\Sigma|)$. The runtime increases for characters higher up in the alphabet. If the number of characters smaller than $c$ equals $k$ then $2k$ calls to the $\mathrm{Occ}^r$ (forward search) or Occ (backward search) table are needed.

By replacing cumulative summations with direct lookups using $\mathrm{Prefix-Occ}$ and $\mathrm{Prefix-Occ}^r$ tables, denoted as PO, respectively $\mathrm{PO}^r$ , this computation avoids the overhead of iterating over the alphabet and reduces the number of memory accesses from $2k$ to 2 [33]. The PO structure stores cumulative counts of all characters less than or equal to $c$, defined as:

$$\mathrm{PO}[i, c] = \sum_{d\le c, d\in\Sigma} \sum_{j=0}^{i-1} \begin{cases} 1 & \text{if } \mathrm{BWT}(T)[j] = d, \\ 0 & \text{otherwise.} \end{cases}$$

The corresponding $\mathrm{PO}^r$ table for $\mathrm{BWT}^r$ is defined analogously. Using these structures, the number of suffixes of $T$ prefixed by $Pd$ for $d < c$ can be computed as:

$$x = \begin{cases} 0 & \text{if } c = \$, \\ \mathrm{PO}^r[e', c-1] - \mathrm{PO}^r[s', c-1] & \text{otherwise,} \end{cases}$$

where $c - 1$ indicates the character just before $c$ in the alphabet.

This lookup-based approach ensures $O(1)$ time complexity for both forward and backward extensions. The rank9 data structure can also be used used to implement PO and $\mathrm{PO}^r$, ensuring efficient storage and constant-time access to cumulative counts.

Accessing the Occ (or Occ$^r$) table can be by-passed by using the PO (or PO$^r$) table as follows:

$$\text{Occ}[i, c] = \text{PO}[i, c] - \begin{cases} 0 & \text{if } c = \$, \\ \text{PO}[i, c - i] & \text{otherwise.} \end{cases} \tag{1.5}$$

#### 1.3.5.1 Search Trie Traversal in a Bidirectional FM-Index

The bidirectional search trie $\mathbb{T}$ for an indexed text $T$ consists of two separate search tries: one corresponding to backward extensions, $\mathcal{T}$, and another corresponding to forward extensions, $\mathcal{T}^r$. These two tries are linked together. $\mathcal{T}$ is the same unidirectional trie as explained in Section 1.3.3. Each node corresponds to a substring $S$ of $T$, determined by the reversed path from the root to the node, and is associated with a range $\mathcal{S}(S, T)$. Conversely, $\mathcal{T}^r$ represents the search trie for the unidirectional FM-index of $T^r$ (the reverse of $T$). Each node in $\mathcal{T}^r$ corresponds to a substring $S'$ of $T^r$, determined by the reversed path from the root to the node in $\mathcal{T}^r$, and is associated with a range $\mathcal{S}(S', T^r)$. A node $n$ in $\mathbb{T}$ corresponds to the combination of a node $n_b$, with associated substring $S_b$, in $\mathcal{T}$, and a node $n_f$, with associated substring $S_f$, in $\mathcal{T}^r$, where $S_b = {S_f}^r$. We call $S_b$ the associated substring of bidirectional node $n$.

A bidirectional node in $\mathbb{T}$ has two sets of children. The first set corresponds to backward extensions, consisting of children with associated substrings $cS$ for any $c \in \Sigma$. The second set corresponds to forward extensions, consisting of children with associated substrings $Sc$ for any $c \in \Sigma$. For a backward extension, the child of $n_b$ is first calculated in $\mathcal{T}$, followed by finding the corresponding node of this child in $\mathcal{T}^r$ using Equation 1.4 and $\mathcal{S}(S^r, T^r)$ . Conversely, for a forward extension, the child of $n_f$ is first calculated in $\mathcal{T}^r$, and then the corresponding child node of this child is identified in $\mathcal{T}$ using the same equation and $\mathcal{S}(S, T)$.

---

**Example 1.19.** *The backward and forward search tries, $\mathcal{T}$ and $\mathcal{T}^r$, for the indexed string $T =$ "banana\$" are depicted in Figure 1.15. Each node in these tries is labeled with its associated range over the suffix array of $T$, or $T^r$ for the reversed string. The substring associated with a node in either of the unidirectional search tries is determined by concatenating the edge labels along the path from the node back to the root of that trie. Links between non-terminal nodes in the two unidirectional tries are represented by dashed lines. The associated substrings of linked nodes are reverses of each other. Two such linked nodes form a single bidirectional node, combining their respective information. Together, the two unidirectional tries form the bidirectional search trie, $\mathbb{T}$.*

*The branch corresponding to the substring $S =$ "bana" ($S^r =$ "anab" in $\mathcal{T}^r$) is highlighted in both unidirectional search tries. The associated ranges, $\mathcal{S}(S, T)$ in $\mathcal{T}$ and $\mathcal{S}(S^r, T^r)$ in $\mathcal{T}^r$, are $[4, 5[$ and $[2, 3[$ respectively. Referring to Table 1.6, we see that SA$(T)[4]$ corresponds to "bana", while SA$(T^r)[2]$ corresponds to "anab", verifying the relationship between the linked nodes.*

***Table 1.6:*** *F column,* SA($T$), BWT($T$), SA($T^r$) *and* BWT($T^r$) *for* $T =$ "`banana$`"

| | | Tables for $T$ | | | Tables for $T^r$ | | |
|---|---|---|---|---|---|---|---|
| $i$ | $F$ | SA($T$) | BWT($T$) | $\text{suf}_{\text{SA}(T)[i]}(T)$ | SA($T^r$) | BWT($T^r$) | $\text{suf}_{\text{SA}(T^r)[i]}(T^r)$ |
| 0 | $ | 6 | a | $ | 0 | b | $ananab |
| 1 | a | 5 | n | a$ | 5 | n | ab |
| 2 | a | 3 | n | ana$ | 3 | n | anab |
| 3 | a | 1 | b | anana$ | 1 | $ | ananab |
| 4 | b | 0 | $ | banana$ | 6 | a | b |
| 5 | n | 4 | a | na$ | 4 | a | nab |
| 6 | n | 2 | a | nana$ | 2 | a | nanab |

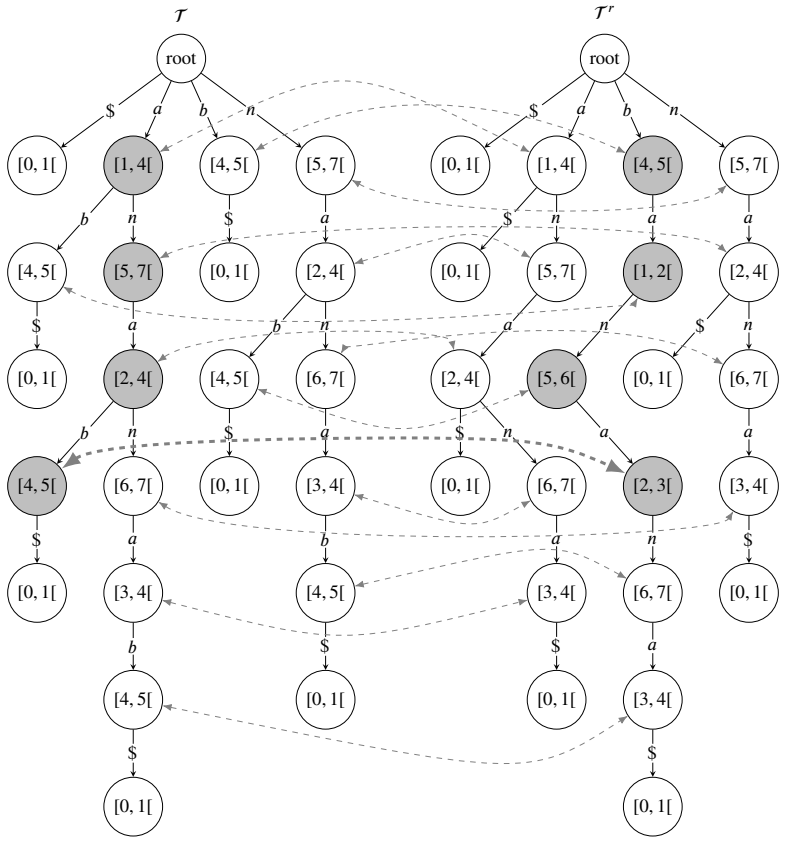


***Figure 1.15:*** *Backward and forward search trie for string* $T =$ "`banana$`"*. The corresponding nodes are linked with double sided dashed arrows. Merging these nodes creates the bidirectional nodes. The branch which corresponds to substring "*`bana`*" of* $T$ *is highlighted in both the forward and the backward search trie. The link between these branches is highlighted with a thicker arrow.*

Algorithm 5 presents the forward and backward extensions in $O(1)$ time for a bidirectional FM-index. The functions LF and $\text{LF}^r$ perform LF-mapping operations, relying on Occ and $\text{Occ}^r$ queries. These queries are calculated on the fly using Equation 1.5.

**Example 1.20.** *Consider indexed string $T$ ="banana\$" and matched pattern $P$ ="bana" (see Figure 1.15). Assume we want to extend the pattern by adding character 'n'. We start by calculating $\mathcal{S}$("nanab", "\$ananab") $= [s'_f, e'_f[ = [\text{LF}^r(s_f, \text{'n'}), \text{LF}^r(s_f, \text{'n'})[ = [\text{LF}^r(2, \text{'n'}), \text{LF}^r(3, \text{'n'})[ = [6, 7[$, using the LF-operation and BWT($T^r$) (see Table 1.6).*

*Then, we can calculate the corresponding range $\mathcal{S}$("banan", "banana\$") $= [s'_b, e'_b[$. First, we find x, the number of occurrences of $d <$ 'n' in BWT($T^r$)$[s_f, e_s[ =$ BWT($T^r$)$[2, 3[=$"n", which is 0. Then we find y, the size of $\mathcal{S}$("nanab", "\$ananab"), which is $|\mathcal{S}(\text{"nanab"}, \text{"\$ananab"})| = |[6, 7[| = 7 - 6 = 1$. From this we find $\mathcal{S}(\text{"banan"}, \text{"banana\$"}) = [4 + 0, 4 + 0 + 1[ = [4, 5[$.*

**Algorithm 5:** Forward/backward extension in a bidirectional FM-index of text $T$

**Input:** Bidirectional node $n = (n_b, n_f, d)$ corresponding to substring $S$, character $c$.
$n_b$: Unidirectional node in $\mathcal{T}$, represented by range $r_b = [s_b, e_b[ = \mathcal{S}(S, T)$
$n_f$: Unidirectional node in $\mathcal{T}^r$, represented by range $r_f = [s_f, e_f[ = \mathcal{S}(S^r, T^r)$
$d$: Depth of the node $n$
$c$: Character $c$ in $\Sigma$ excluding the sentinel.
**Output:** Child node of $n$ corresponding to forward extension $Sc$ or backward extension $cS$

**Function** `extensionForward(n, c):`
- `// Step 1: Compute the child in` $\mathcal{T}^r$
- $s'_f \leftarrow \text{LF}^r(s_f, c), e'_f \leftarrow \text{LF}^r(e_f, c)$
- **if** $s'_f == e'_f$ **then**
  - **return** `None` `// Extension does not exist`
- **end**
- `// Step 2: Compute range for` $\mathcal{S}(Sc, T)$ `in` $\mathcal{T}$
- $s'_b \leftarrow s_b + \text{PO}'[e_f][\Sigma.\text{prev}(c)] - \text{PO}'[b_f][\Sigma.\text{prev}(c)]$ `//` $s_b + x$
- $e'_b \leftarrow s'_b + (e'_f - s'_f)$ `//` $s_b + x + y$
- **return** $([s'_b, e'_b[, [s'_f, e'_f[, d{+}1)$

**return**

**Function** `extensionBackward(n, c):`
- `// Step 1: Compute the child in` $\mathcal{T}$
- $s'_b \leftarrow \text{LF}(s_b, c), e'_b \leftarrow \text{LF}(e_b, c)$
- **if** $s'_b == e'_b$ **then**
  - **return** `None` `// Extension does not exist`
- **end**
- `// Step 2: Compute range for` $\mathcal{S}(cS^r, T^r)$ `in` $\mathcal{T}^r$
- $s'_f \leftarrow s_f + (\text{PO}[e_b][\Sigma.\text{prev}(c)] - \text{PO}[b_b][\Sigma.\text{prev}(c)])$ `//` $s_f + x$
- $e'_f \leftarrow s'_f + (e'_b - s'_b)$ `//` $s_f + x + y$
- **return** $([s'_b, e'_b[, [s'_f, e'_f[, d{+}1)$

**return**

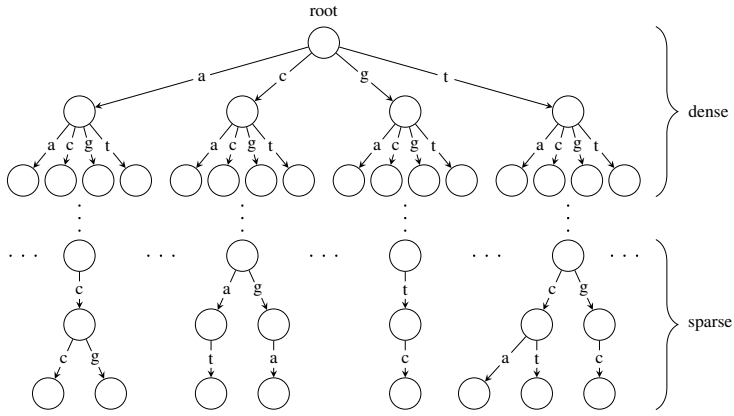


***Figure 1.16:*** *Visualization of the bidirectional search trie used for approximate pattern matching in a repetitive genome. Near the root of the trie, the dense region shows high branching, with each node typically having four children. In contrast, the sparse region represents deeper levels of the trie, where branching becomes less frequent as the sequence specificity increases, leading to more isolated paths.*

## 1.4 Search Schemes

Consider the human genome as a string over the 4-letter alphabet $\Sigma = \{A, C, G, T\}$. Assuming the bases are randomly and uniformly distributed across the genome, the human genome, which contains approximately 3 000 000 000 bases, has an expected occurrence count for any given 15-mer that is greater than 1 ($4^{15} < 3\,000\,000\,000$).

As a result, the (bidirectional) search trie is highly dense near the root (see Figure 1.16), where each node is expected to have 4 children on average. This density leads to a significant computational burden during approximate pattern matching, as many branches explored in the search trie will ultimately prove unsuccessful.

Lam et al. were the first to highlight this issue, noting that the high density near the root causes inefficiency in the lossless approximate pattern matching process [30]. To address this, they proposed a strategy based on the pigeonhole principle: first, a portion of the pattern $P$ is matched exactly, allowing the search to move deeper into the trie along a single branch. The approximate matching for the remaining parts of $P$ is then conducted in a less dense subtrie, significantly improving efficiency.

Kucherov et al. later generalized this approach by introducing the concept of **search schemes**, a structured framework that defines how a pattern $P$ is matched using a bidirectional full-text index such that unsuccessful branches are discarded as quickly as possible, reducing the search space and hence, runtime [34]. Pockrandt further refined and extended the original definition by Kucherov et al., making it less restrictive [35].

Search schemes require a bidirectional index, such as the bidirectional FM-index, the affix tree [36] and the affix array [37]. In this dissertation we use a bidirectional FM-index.

**Definition 1.14.** *A **search** is a triplet of arrays $S = (\pi, L, U)$, given parameters $p$ and $k$. The $\pi$-array is a permutation of $\{0, \ldots, p-1\}$ and indicates the order in which the parts of the pattern are processed. To ensure that a search can be performed with a bidirectional index, this permutation must satisfy the **connectivity property**: for every $i > 0$, $\pi[i]$ is either $(\min_{j<i} \pi[j] - 1)$ or $(\max_{j<i} \pi[j] + 1)$. The $U$- and $L$-arrays are arrays of length $p$ over $\{0, \ldots, k\}$; they respectively provide upper and lower bounds on the* cumulative *number of errors in the parts in $\pi$-order. Because they are cumulative, both $L$ and $U$ must be non-decreasing, and they must satisfy $L[i] \leq U[i]$ for all $i$.*

---

**Example 1.21.** *Consider the search $S = (102, 001, 012)$ defined for $k = 2$ and $p = 3$. The search starts by handling part $\pi[0] = 1$. As $U[0] = 0$, an exact match of this part is searched. If such a match is found, the search continues with part $\pi[1] = 0$, where $U[1] = 1$ errors are allowed. Finally, the search continues with part $\pi[2] = 2$. Here, up to two cumulative errors are allowed ($U[2] = 2$), but at the end at least one error ($L[2] = 1$) should have been encountered.*

---

**Definition 1.15.** *An **error distribution** $e$, defined for at most $k$ errors over $p$ parts, is an integer array over $\{0, \ldots, k\}$ of length $p$, for which $\sum_{i=0}^{p-1} e[i] \leq k$.*

---

**Example 1.22.** *Consider error distribution* 101 *defined for 2 errors and 3 parts. This distribution has one error in the leftmost part and one error in the rightmost part.*

---

**Lemma 1.3.** *There are $Q = \binom{p+k}{k}$ distinct error distributions for $p$ parts with at most $k$ errors [38].*

*Proof.* We use exactly $k$ errors and $p$ part separators for $p + 1$ parts; the last "part" collects errors that do not appear in the pattern (thereby allowing fewer than $k$ errors). In a sequence of length $p+k$ of errors and separators, we choose a subset of $p$ positions for the separators; there are $Q = \binom{p+k}{k}$ ways to do this [38]. □

**Definition 1.16.** *A search $S = (\pi, L, U)$ **covers** an error distribution if and only if for all $0 \leq i < p$: $L[i] \leq \sum_{j=0}^{i} e[\pi[j]] \leq U[i]$.*

---

**Example 1.23.** *The error distribution from example 1.22 is covered by the search from example 1.21. We have $L[0] \leq e[\pi[0]] \leq U[0]$ (spelling out $0 \leq e[1] = 0 \leq 0$); then $L[1] \leq e[\pi[0]] + e[\pi[1]] \leq U[1]$ (spelling out $0 \leq 1 \leq 1$); and finally $L[2] \leq \sum_j e[j] \leq U[2]$ (spelling out $1 \leq 1 \leq 2$).*

---

**Definition 1.17.** *A **search scheme** $\mathbb{S}$ is a set of searches defined for fixed $k$ and $p$. A search scheme is **valid** if each error distribution with $p$ parts and at most $k$ errors is covered by at least one search of $\mathbb{S}$.*

**Example 1.24.** *Consider the search scheme for 2 errors and three parts proposed by Kucherov et al.* $\mathbb{S} = (012, 000, 022), (210, 000, 012), (102, 001, 012)$*. In Figure 1.17 the searches of this scheme are depicted. The parts of the pattern are labeled as* $\mathcal{P}_i$*, where i represents the index of each part, read sequentially from left to right. In the* $S_0$ *search, exact matching is first performed for the leftmost part* $\mathcal{P}_0$*. Next, this exact match is extended to the right, thus processing parts* $\mathcal{P}_1$ *and* $\mathcal{P}_2$*, using a backtracking procedure that allows up to two errors. In the* $S_1$ *search, exact matching is first performed for the rightmost part* $\mathcal{P}_2$*, and extended to the left by first allowing up to a single error in* $\mathcal{P}_1$*, and then two errors in* $\mathcal{P}_0$*. Indeed, occurrences of P with two errors in the middle part were already covered by search* $S_0$*. Finally, search* $S_2$ *first involves an exact matching of* $\mathcal{P}_1$*, which is then extended to the left allowing a single error, and finally to the right with at least one, and at most two errors.*

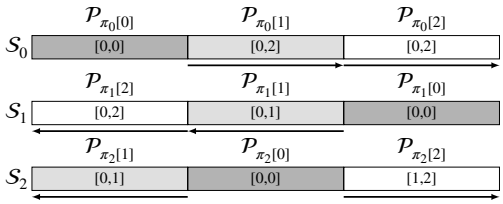


***Figure 1.17:*** *Search scheme for* $k = 2$ *errors and* 3 *parts proposed by Kucherov et al. The parts are processed from darkest to lightest shade of gray. In each part, the lower and upper bound to the cumulative number of errors (as an inclusive interval) up to and including that part, are indicated. The arrows indicate the search direction (left-to-right or right-to-left).*

The naive backtracking approach, allowing up to $k$ errors, as described in Algorithms 3 and 3 in Section 1.3.3, can be represented as a search scheme with a single search $S_{back}$, where $p = 1$, and $S_{back} = (0, 0, k)$.

For more complex search schemes (such as the one in Example 1.24), a bidirectional index is often necessary, as such schemes frequently have at least one search that starts by matching a part located in the middle of the pattern $P$.

## 1.4.1 Overview of Established Search Schemes

All search schemes discussed in this section, as well as those developed during this dissertation, are summarized in Appendix A.

### 1.4.1.1 Pigeonhole Search Strategy

The pigeonhole search strategy is rooted in the pigeonhole principle, which guarantees that if a pattern $P$ is divided into $k + 1$ parts, at least one part must match the text $T$ exactly for any approximate match with up to $k$ errors. This principle forms the basis of the $k$-mismatch pigeonhole search scheme.

| $\mathcal{P}_0$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ |
|---|---|---|
| 2 | 0* | 0* |
| 0* | 2 | 0* |
| 0* | 0* | 2 |
| 1 | 1 | 0* |
| 0* | 1 | 1 |
| 1 | 0* | 1 |
| 1 | 0* | 0* |
| 0* | 1 | 0* |
| 0* | 0* | 1 |
| 0* | 0* | 0* |

***Figure 1.18:*** *All distributions of 2 errors over 3 parts. It is indicated if a distribution is covered by the search from* $\mathbb{S}_2 = (012, 000, 022)$*,* $(120, 000, 022)$*,* $(210, 000, 022)$ *(the pigeonhole search scheme for 2 errors), that starts with exact matching part* $\mathcal{P}_i$*.*

In this approach, $P$ is partitioned into $k+1$ parts $(\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k)$, and each parts is initially matched exactly, followed by approximate matching for the remaining parts. In a specific search $\mathcal{S}_i$, the process starts with an exact match of part $\mathcal{P}_i$, followed by approximate forward matching of subsequent parts and backward matching of preceding parts. This design reduces the frequency of switching between forward and backward searches, though alternative configurations are feasible.

The pigeonhole search schemes for $0 < k \leq 7$ are presented in Table A.1. Such a search scheme can be generalized for any arbitrary $k$ as follows:

$$\mathcal{S}_i = ((i, i+1, \ldots, k-1, k, i-1, i-2, \ldots 0),\ 00 \ldots 0,\ 0kk \ldots k), \quad 0 \leq i \leq k$$

The pigeonhole Search scheme can result in redundant matches, as the same occurrence may be covered by multiple searches. For example, Figure 1.18 illustrates all possible error distributions for $k = 2$, showing overlaps between searches.

#### 1.4.1.2 Search schemes based on 01*0-seeds

In 2016, Vroland et al. defined 01*0 seeds [39]. They proved that, if $P$ is divided into $k+2$ parts, then for any approximate match of the pattern $P$ to the text $T$ with $k$ or fewer errors, there exists at least one pair of parts, $\mathcal{P}_m$ and $\mathcal{P}_n$, that both match exactly, and all parts $\mathcal{P}_j$ between them, where $k < j < m$, match with exactly one error each. Parts $\mathcal{P}_m$,…,$\mathcal{P}_n$ then form a 01*0-seed.

One possible approach to formalizing the 01*0 seeds is to define a separate search for each pair of error-free parts. However, this method results in a significant number of redundant searches. An alternative approach, proposed by Pockrandt [35], involves merging all seeds that share the same first error-free part into a single search.

In this method, the $L$ strings are merged by taking the minimum value at each position, and the $U$ strings are merged by taking the maximum. This consolidation

reduces the total number of searches from $\binom{k+2}{k}$ to $k+1$. The 01*0 search schemes are presented in Table A.4 for $1 < k \leq 7$. They can be defined for arbitrary $k$ as follows:

$$S_i = ((i, i+1, \dots, k, k+1, i-1, i-2, \dots 0),\ 00\dots00,\ 01k\dots k), \quad 0 \leq i < k+1$$
$$S_{k+1} = ((k+1, k, \dots, 1, 0),\ 00\dots00,\ 00k\dots k).$$

#### 1.4.1.3 Search schemes presented by Kucherov et al.

In 2015, Kucherov et al. introduced the concept of search schemes and analyzed their performance using the idea of a critical string [34]. This critical string, defined as the lexicographically largest $U$-string across all searches, often determines the computational bottleneck of a scheme.

---

**Example 1.25.** *Consider the 01*0 search scheme with k allowed errors. The critical string is* $\{0, 1, k, \dots, k\}$*, which is the* $U$*-array for searches* $S_i$ *with* $0 \leq i < k+1$*, represented as a string.*

---

Kucherov et al. developed a model to estimate the workload of a search based on the number of substrings enumerated (explored nodes in the search trie) by all searches, assuming a random text $T$ and a random pattern $P$. The researchers also noted that uneven partitioning (where not all parts of $P$ have the same length), can lead to lower workloads and created 'optimal' partitionings for their search schemes.

They presented search schemes for $k+1$ and $k+2$ parts, designed through a greedy algorithm that iteratively adds searches to the scheme. At each step, the algorithm identifies the uncovered error distribution $A$ of weight $k$ such that the lexicographically minimal $U$-string that covers $A$ is maximal. From the set of searches that can cover $A$ with this minimal $U$-string, it selects the search that covers the maximum number of remaining uncovered error distributions of weight $k$. The $L$-string of the chosen search is then set to be lexicographically maximal among all possible $L$-strings that do not reduce the number of uncovered strings of weight $k$ covered by the search. These search schemes are presented in Table A.5.

#### 1.4.1.4 Search schemes presented by Kianfar et al.

Kianfar et al. noted that the Kucherov schemes still exhibited redundancy, where some matches could be reported by multiple searches [40]. To address this, they developed a mixed-integer linear programming (MILP) model to minimize the number of visited nodes in the search trie while ensuring that each occurrence is covered by exactly one search, using Kucherov et al. 's model for the number of nodes visited under the Hamming distance.

This approach was applied to design search schemes for alignment with up to $k = 4$ errors and a limited number of searches. Their search schemes with $k+1$ parts are presented in Table A.7.

#### 1.4.1.5 $\text{Man}_{\text{Best}}$ search scheme

Pockrandt observed that the Optimal Kianfar strategy for $K = 3$ and $K = 4$ included searches that allowed errors in the first part of the pattern. This is inefficient, as the first part is aligned in the densest region of the search trie. To address this, Pockrandt manually refined the Kianfar search schemes for $K = 4$, ensuring that the first part of every search was error-free.

The resulting scheme, termed "$\text{Man}_{\text{Best}}$", consists of 5 searches and avoids errors in the first part, thereby improving efficiency. This scheme is also detailed in Table A.7.

### 1.4.2 Approximate Pattern Matching With Search Schemes

Approximate pattern matching of a pattern $P$ to a text $T$ using a search scheme involves executing each search defined in the scheme, either sequentially or in parallel. Each search operates, independently, on the conceptual bidirectional search trie of the bidirectional FM-index of $T$, allowing character extensions in both forward and reverse directions.

For a given search $S_s = (\pi_s, L_s, U_s)$, $P$ is divided into $|\pi_s|$ parts $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{\pi_s-1}$, which are processed in the order specified by $\pi_s$. The direction of processing (forward or backward extensions) for each part $\mathcal{P}_{\pi_s[i]}$ is determined by comparing $\pi_s[i]$ with the index of the previously processed part, $\pi_s[i-1]$. The direction of the first step $p_{0,s}$ is arbitrary.

The first step $p_{0,s}$ starts at the root node of the search trie, where the empty string is already considered a match. If $U_s[0] = 0$, the algorithm performs exact matching by extending $|\mathcal{P}_{\pi_s[0]}|$ characters to check if a node corresponding to $\mathcal{P}_{\pi_s[0]}$ exists in the trie. If no such node is found, the search terminates. For $U_s[0] > 0$, an approximate alignment procedure finds all nodes $n$ with substrings $S_n$ having a distance $\delta_n$ to $\mathcal{P}_{\pi_s[0]}$ within bounds $L_s[0] \leq \delta_n \leq U_s[0]$. These nodes $n$ and their scores $\delta_n$ are all passed to the next step $p_{1,s}$.

Subsequent steps $p_{i,s}$ use the nodes $n$ and scores $\delta_n$ from the previous step. For each node $n$, alignment starts at node $n$ with $\delta_n$ errors carried over from the previous step. For the edit distance, alignment starts with an initialized alignment matrix where all values in the top row and first column have been incremented with $\delta_n$. The search trie below node $n$ is then explored to find nodes $m$, for which $L_s[i] \leq \delta_m \leq U_s[i]$, with $\delta_m$ equal to the distance score between $S_m$ and $P[\min(\pi_s[0,i]), \max(\pi_s[0,i])]$. All such nodes $m$ are then passed together with their corresponding $\delta_m$ to the next step (if there is a next step). At the end of the final step, these nodes are converted from suffix array indices to text positions, completing the search.

Algorithms 6 and 7 (on pages 50 and 51) demonstrate this process for Hamming and edit distances, respectively. These algorithms use the bidirectional extensions defined in Algorithm 5. The key distinction lies in the *approximateBidirectionalStep* functions, which handle mismatches (Hamming distance) or mismatches and indels (edit distance). For efficiency, the alignment matrix $D$ in Algorithm 7 can also be replaced by a banded matrix (see Section 1.3.3.1).

**Algorithm 6:** Applying a Search Scheme With the Hamming Distance

**Input:** Pattern $P$ partitioned into $p$ parts $\mathcal{P}_0, \dots \mathcal{P}_{p-1}$, Index of Text $T$ $\mathcal{I}$, Search Scheme $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|-1}\}$, empty list $v$
**Result:** List $v$ is filled with approximate occurrences of $P$ in $T$ (tuples of position in $T$ and hamming distance)

root $\leftarrow$ ([0, $|T|$[, [0, $|T|$[, 0), $v \leftarrow$ [] **foreach** ***search*** $S_s \in \mathcal{S}$ **do**
  `executeStepHamming` ([root,0], 0, $S_s$, $P$, $v$)
**return** $v$

**Function** `getDirection`(*step index i, search* $S_s = (\pi_s, L_s, U_s)$)**:**
  **return** *FORWARD* ***if*** $i == 0$ ***or*** $\pi_s[i] < \pi_s[i-1]$ ***else*** *BACKWARD*

**Function** `getExtensionFunction`(*direction* $\mathcal{D}$)**:**
  **return** `extensionForward` ***if*** $\mathcal{D}$ == *FORWARD* ***else*** `extensionBackward`

**Function** `exactBidirectionalStep`(*node n, part* $\mathcal{P}_i$ *of pattern P, direction* $\mathcal{D}$*, extension function* $\mathcal{E}$*, distance score* $\delta_n$)**:**
  **foreach** $j \in [0, 1 \dots, |\mathcal{P}_i| - 1]$ *in direction* $\mathcal{D}$ **do**
    $n \leftarrow \mathcal{E}(n, \mathcal{P}_i[j])$
    **if** *n is* ***None*** **then return** []
  **return** $[n, \delta_n]$

**Function** `approximateBidirectionalStepHamming`(*node n, distance score* $\delta_n$*, pattern P partitioned into p parts* $\mathcal{P}_0, \dots \mathcal{P}_{p-1}$*, direction* $\mathcal{D}$*, extension function* $\mathcal{E}$*, step index i, search* $S_s = (\pi_s, L_s, U_s)$)**:**
  $\mathcal{P} \leftarrow \mathcal{P}_{\pi_s[i]}$ , $\sigma \leftarrow$ empty stack, $v \leftarrow$ []
  **foreach** $c \in \Sigma$ **do**
    $n' \leftarrow \mathcal{E}(n, c)$
    **if** $n' \neq$ ***None*** **then** $\sigma$.add $(c, (n'.n_b, n'.n_f, 1), \delta_n)$  `// Set depth to 1 sub-trie`
  **while** $|\sigma| > 0$ **do**
    $c, n', \delta_n \leftarrow \sigma$.pop()  `// Pop character, node and parent score`
    $c_{\mathcal{P}} \leftarrow \mathcal{P}[n'.d - 1]$ **if** $\mathcal{D} ==$ `FORWARD` **else** $\mathcal{P}[|\mathcal{P}| - n'.d]$
    $\delta_{n'} \leftarrow \delta_n + 1$ **if** $\mathcal{P}[n'.d - 1] \neq c$ **else** $\delta_p$
    **if** $\delta_{n'} > U_s[i]$ **then continue**
    **if** $n'.d == |\mathcal{P}| - 1$ **then**
      $n'.d \leftarrow n.d + n'.d$  `// Set depth to depth in main trie`
      **if** $\delta_{n'} \leq L_s[i]$ **then** $v$.add$((n', \delta_{n'}))$
      **continue**
    **foreach** $c' \in \Sigma$ **do**
      $n'' \leftarrow \mathcal{E}(n', c')$
      **if** $n'' \neq$ ***None*** **then** $\sigma$.add $(c', n'', \delta_{n'})$
  **return** $v$

**Function** `executeStepHamming`(*list v (of nodes n and distance scores* $\delta_n$*), step index i, search* $S_s = (\pi_s, L_s, U_s)$*, pattern P, report list r*)**:**
  **if** $i == |\pi_s|$ **then**
    **foreach** $(n, \delta_n) \in v$ **do**
      **foreach** $j \in n.n_b.r$ **do**
        $r$.add ([SA[$i$], $\delta_n$])  `// Add position and score to vector`
    **return**
  $\mathcal{D} \leftarrow$ `getDirection`$(i, S_s)$, $\mathcal{E} \leftarrow$ `getExtensionFunction`$(\mathcal{D})$, $v_i \leftarrow$ []
  **foreach** $(n, \delta_n) \in v$ **do**
    $v' \leftarrow$ `exactBidirectionalStep`$(n, P[\pi_s[i]], \mathcal{D}, \mathcal{E}, \delta_n)$ **if** $U_s[i] == \delta_n$ **else**
     `approximateBidirectionalStepHamming`$(n, \delta_n, P, , \mathcal{D}, \mathcal{E}, i, S_s)$
    `executeStepHamming`$(v', i + 1, S_s, P, r)$

**Algorithm 7:** Applying a Search Scheme With the Edit Distance

**Input:** Pattern $P$ partitioned into $p$ parts $\mathcal{P}_0, \dots \mathcal{P}_{p-1}$, Index of Text $T$ $\mathcal{I}$, Search Scheme $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|-1}\}$, empty list $v$
**Result:** List $v$ is filled with approximate occurrences of $P$ in $T$ (tuples of position in $T$ and hamming distance)

root $\leftarrow$ ([0, $|T|$[, [0, $|T|$[, 0), $v \leftarrow$ [] **foreach** ***search*** $S_s \in \mathcal{S}$ **do**
  `executeStepEdit` ([root,0], 0, $S_s$, $P$, $v$)
**return** $v$

**Function** `approximateBidirectionalStepEdit`(*node n, distance score* $\delta_n$*, pattern* $P$*, direction* $\mathcal{D}$*, extension function* $\mathcal{E}$*, step index i, search* $S_s = (\pi_s, L_s, U_s)$)**:**
  $\mathcal{P} \leftarrow P[\pi_s[i]]$ `// Pattern part for this step`
  $\sigma \leftarrow$ empty stack, $v \leftarrow$ []
  **foreach** $c \in \Sigma$ **do**
    $n' \leftarrow \mathcal{E}(n, c)$, $n'.d \leftarrow 1$ `// Set depth to 1 in sub-trie`
    **if** $n' \neq$ ***None*** **then** $\sigma$.`add` $(c, n')$
  $D \leftarrow$ matrix of size $(|\mathcal{P}| + U_s[i] + 1) \times (|\mathcal{P}| + 1)$ with all elements set to 0
  **for** $j \leftarrow 0$ **to** $|\mathcal{P}|$ **do**
    $D[0][j] \leftarrow j + \delta_n$ `// Set top row with increasing values`
  **while** $|\sigma| > 0$ **do**
    $c, n' \leftarrow \sigma$.`pop`() `// Pop node and parent score`
    $D[n'.d][0] \leftarrow d'$
    **for** $j \leftarrow 1$ **to** $|\mathcal{P}|$ **do**
      `// DP update: insertion, deletion or substitution`
      $c_{\mathcal{P}} \leftarrow \mathcal{P}[j-1]$ **if** $\mathcal{D} ==$ `FORWARD` **else** $\mathcal{P}[|P| - j]$
      $m \leftarrow 1$ **if** $c_{\mathcal{P}} \neq c$ **else** 0
      $D[n'.d][j] \leftarrow \min\left(D[n'.d-1][j]+1,\ D[n'.d'][j-1]+1,\ D[n'.d-1][j-1]+m\right)$
    **if** $\max(D[n'.d] > U_s[i]$ **then continue**
    **if** $L_s[i] \leq D[n'.d][|\mathcal{P}|] \leq U_s[i]$ **then**
      $n'.d \leftarrow n.d + n'.d$ `// Set depth to depth in main trie`
      $v$.`add`$((n', D[n'.d][|\mathcal{P}|]))$
    **if** $n'.d == |\mathcal{P}| + U_s[i]$ **then continue**
    **foreach** $c' \in \Sigma$ **do**
      $n'' \leftarrow \mathcal{E}(n', c')$
      **if** $n'' \neq$ ***None*** **then** $\sigma$.`add` $(c', n'', \delta_{n'})$
  **return** $v$

**Function** `executeStepEdit`(*list v (of nodes n and distance scores* $\delta_n$*), step index i, search* $S_s = (\pi_s, L_s, U_s)$*, pattern P, report list r*)**:**
  **if** $i == |\pi_s|$ **then**
    **foreach** $(n, \delta_n) \in v$ **do**
      `// Iterate over all indexes in SA(`$T$`) associated with node` $n$
      **foreach** $j \in n.n_b.r$ **do**
        $r$.`add` $([\text{SA}[i], \delta_n, n.d])$ `// Add position, score and depth`
    **return**
  $\mathcal{D} \leftarrow$ `getDirection`$(i, S_s)$, $\mathcal{E} \leftarrow$ `getExtensionFunction`$(\mathcal{D})$, $v_i \leftarrow$ []
  **foreach** $(n, \delta_n) \in v$ **do**
    $v' \leftarrow$ `exactBidirectionalStep`$(n,\ P[\pi_s[i]],\ \mathcal{D},\ \mathcal{E},\ \delta_n)$ **if** $U_s[i] == \delta_n$ **else**
     `approximateBidirectionalStepEdit`$(n,\ \delta_n,\ P,\ ,\ \mathcal{D},\ \mathcal{E},\ i,\ S_s)$
    `executeStepEdit`$(v',\ i + 1,\ S_s,\ P,\ r)$

## 1.5 Sequence Aligners Overview

This section discusses the sequence aligners compared against in this dissertation: *BWA* (`aln` and `mem` modes), *Bowtie* (versions 1 and 2), *RazerS3*, *Yara*, *GEM*, *Bwolo*, and *Ropebwt3*. These aligners can be classified as either **lossy** or **lossless** based on their alignment guarantees.

Lossless aligners ensure that all matches below a specified error threshold are reported. In contrast, lossy aligners, while generally faster, do not guarantee exhaustive reporting of matches but aim to find a subset of high-quality alignments.

A number of these aligners utilize *Maximal Exact Matches (MEMs)* as seeds to initiate alignment. A MEM is a substring common to both the query and reference sequences without mismatches, which cannot be extended in either direction without introducing one. MEMs play a crucial role in sequence alignment as they provide reliable starting points for the process.

**BWA** BWA is a widely used tool for mapping DNA sequences to reference genomes, offering two modes: `aln` (optimized for short reads up to 100 bp) and `mem` (designed for longer reads). BWA-MEM uses super-maximal exact matches (SMEMs), a subset of MEMs, as seeds. The aligner employs a unidirectional FM-index for genome indexing and supports affine gap penalties for enhanced accuracy. However, BWA uses heuristics to balance speed and sensitivity, which precludes exhaustive alignment below a threshold, categorizing it as a lossy aligner [16].

**Bowtie** Bowtie is an ultrafast, memory-efficient aligner. Bowtie 1 is optimized for speed and memory usage, while Bowtie 2 introduces support for gapped, local, and more flexible paired-end alignments, along with affine gap penalties. These improvements make it suitable for diverse applications, including longer and more complex reads. Both versions use a unidirectional FM-index. Like BWA, Bowtie employs heuristics and does not guarantee exhaustive alignment below a threshold [17, 41].

**RazerS3** RazerS3 is a flexible read mapper that uses $q$-gram counting to allow for adjustable sensitivity. It does not rely on precomputed indices, enabling versatile mapping. RazerS3 supports shared-memory parallelism and operates in a lossless mode, guaranteeing that all matches below a specified error threshold are reported. By default, it reports all best alignments, making it a lossless aligner [42, 43].

**Yara** Yara is a fast and memory-efficient aligner for both exact and approximate DNA sequence alignment. It uses a unidirectional FM-index and provides exhaustive enumeration of alignments under the edit distance threshold. Yara supports a lossless mode and, by default, reports all best alignments. Hence, it is classified as a lossless aligner [44, 45].

**GEM** GEM is a high-performance tool suite for processing high-throughput sequencing data, supporting both single- and paired-end reads. It uses a unidirectional FM-

index for indexing and employs heuristics to balance speed and accuracy [46]. While GEM claims to offer a lossless mode for exhaustive alignment below a specified error threshold, in practice, complete sensitivity is not always achieved [44].

**Bwolo** Bwolo, developed by Christophe Vroland et al., uses 01*0-seeds (see Section 1.4.1.2) for approximate string matching over DNA sequences. It is optimized for short patterns (<50 bp) in large datasets with high error rates (7–15%). Bwolo guarantees the identification of all approximate matches up to a predefined error limit and uses a unidirectional search index, categorizing it as a lossless aligner.

**Ropebwt3** Ropebwt3 is designed for efficient Burrows-Wheeler Transform (BWT) construction and alignment at the terabase scale. It supports both exact and inexact matching and is optimized for repetitive datasets such as pan-genomes. Ropebwt3 identifies MEMs, performs local alignments with affine gap penalties, and outputs results in Pairwise mApping Format (PAF) format. It reports only the primary alignment and the number of other matches, which classifies it as a lossy aligner [47].

RazerS3, Yara, GEM, BWA, and Bowtie can output alignments in SAM/BAM formats. However, Bwolo and Ropebwt3 do not support paired-end alignment.

## 1.6 Research Questions

Lossless approximate pattern matching is crucial for achieving high accuracy in sequence alignment, which is foundational to many bioinformatics applications. Despite its advantages in precision, lossless alignment methods have lagged behind heuristic-based aligners like BWA-MEM and Bowtie in terms of computational efficiency. This thesis investigates whether the performance gap can be closed through the development of more efficient lossless alignment techniques. The two main research questions throughout this work are: **(1) Can performance levels comparable to heuristic-based aligners be achieved by optimizing lossless alignment methods, particularly through the development of efficient search schemes and their implementation in a bidirectional FM-index?** and **(2) What defines an efficient search scheme, and can we design new or improved search schemes to further enhance the performance of lossless alignment?**

## 1.7 Outline of this Work

This thesis is accompanied by the release of Columba (Latin for pigeon), a lossless approximate pattern matching tool, with the implementation history and feature development documented throughout the chapters. Together, these contributions mark significant progress in bridging the performance gap between lossy and lossless alignment, while advancing the state of the art in lossless approximate pattern matching.

In Chapter 2, we introduce the first version of Columba, focusing on tailoring search schemes to the specific combination of search patterns and text. This chap-

ter presents a novel algorithm for dynamically partitioning search patterns, a universal approach applicable to any search scheme, which significantly reduces the search space (the number of nodes visited in the search trie) and enhances performance. Additionally, we address the inherent redundancy in the edit distance metric and propose methods to minimize redundant calculations. Cache optimization is also explored, with a focus on memory interleaving the PO tables. Benchmarking results establish Columba as a robust and efficient lossless aligner, demonstrating superior performance compared to state-of-the-art tools.

Chapter 3 builds on the original version of Columba by introducing Columba 1.1, an enhanced version that integrates hybrid in-index matching with in-text verification. This chapter explores how candidate occurrences can be validated directly in the text using a bit-parallel, pairwise alignment technique. By abandoning index alignment when the number of candidate occurrences falls below a certain threshold, this approach reduces cache misses and improves runtime efficiency. Furthermore, manual adaptations to search schemes are explored to optimize performance further. These innovations result in substantial runtime reductions, making Columba 1.1 competitive with both lossy and lossless alignment tools.

In Chapter 4, we turn to the automated design of search schemes, presenting two novel methods: a greedy heuristic and an Integer Linear Programming (ILP) model. These approaches enable the design of efficient search schemes for up to $k = 13$ and $k = 7$ errors, extending beyond the previously available designs for up to 4 errors. The chapter also introduces dynamic scheme selection, a complementary technique to dynamic partitioning, which further enhances runtime performance. These advancements are implemented in Columba 1.2, which outperforms state-of-the-art lossless aligners.

In Chapter 5, we present Columba as a fully realized tool, incorporating all previous innovations along with additional features such as multi-threading, paired-end alignment, and support for run-length compression. Comprehensive benchmarking demonstrates Columba's robustness and versatility across a variety of datasets, including the human genome and pan-genomes. This chapter also includes a use case highlighting Columba's application in HLA typing, the motivating example in this introductory chapter, as part of the OptiType pipeline, where it significantly reduces computational time without compromising accuracy. This illustrates the practical impact of the methods developed in this thesis.

Finally, Chapter 6 summarizes the key contributions of this thesis and discusses potential future directions. These include leveraging approximate occurrences as seeds for aligning longer reads, designing even more efficient search schemes, and exploring novel biological applications such as species identification. The chapter concludes with a reflection on the work presented, emphasizing its impact on narrowing the performance gap between lossy and lossless alignment methods and its contributions to advancing the field of lossless approximate pattern matching.

## 1.8 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of (inter)national conferences. The following list provides an overview of the publications during this PhD research.

### 1.8.1 Publications in International Journals

- **Renders L.**, Marchal K., and Fostier J. (2021) *Dynamic partitioning of search patterns for approximate pattern matching using search schemes*. *iScience*, vol. 24, no. 7, article no. 102687. doi: 10.1016/j.isci.2021.102687.

- **Renders L.**, Depuydt L., Rahmann S., and Fostier J. (2024) *Lossless approximate pattern matching: automated design of efficient search schemes*. *Journal of Computational Biology*, vol. 31, no. 10, pp. 975–989. doi: 10.1089/cmb.2024.0664.

- **Renders L.**, Depuydt L., Gagie T., and Fostier J. (2024) *Columba: Fast Approximate Pattern Matching with Optimized Search Schemes*. Submitted to *Bioinformatics* (March 2025).

- Depuydt L., **Renders L.**, Abeel T., and Fostier J. (2023) *Pan-genome de Bruijn graph using the bidirectional FM-index*. *BMC Bioinformatics*, vol. 24, no. 1, article no. 400, pp. 33. doi: 10.1186/s12859-023-05531-6.

### 1.8.2 Contributions to International Conferences

- **Renders L.**, Marchal K., and Fostier J. (2021) *Dynamic partitioning of search patterns for approximate pattern matching using search schemes*. Talk presented at 11th RECOMB Satellite Workshop on Massively Parallel Sequencing (RECOMB-Seq 2021), virtual conference due to the pandemic. Available at: https://www.youtube.com/watch?v=roCr-9GoNfE.

- **Renders L.**, Depuydt L., and Fostier J. (2022) *Approximate pattern matching using search schemes and in-text verification*. Presented at the International Workshop on Bioinformatics and Biomedical Engineering (IWBBIO 2022), Maspalomas, Spain. In *Bioinformatics and Biomedical Engineering, PT II*, vol. 13347, pp. 419–435. Springer. doi: 10.1007/978-3-031-07802-6_36.

- **Renders L.**, Depuydt L., and Fostier J. (2022) *Approximate pattern matching using search schemes and in-text verification*. Poster presented at the 7th European Student Council Symposium (ESCS 2022), Sitges, Barcelona.

- **Renders L.**, Depuydt L., and Fostier J. (2024) *Approximate pattern matching using search schemes and in-text verification*. Talk presented at the 9th Workshop on Data Structures in Bioinformatics (DSB 2023), Delft, Holland, March 21-22.

- **Renders L.**, Depuydt L., and Rahmann S., and Fostier J. (2024) *Automated design of efficient search schemes for lossless approximate pattern matching*. Talk presented at the 10th Workshop on Data Structures in Bioinformatics (DSB 2024), Montpellier, France, March 14-15.

- **Renders L.**, Depuydt L., and Rahmann S., and Fostier J. (2024) *Automated design of efficient search schemes for lossless approximate pattern matching*. Presented at RECOMB 2024, Cambridge, MA, USA. In *Research in Computational Molecular Biology, RECOMB 2024*, vol. 14758, pp. 164–184. Springer Nature Switzerland. doi: 10.1007/978-1-0716-3989-4_11.

- Depuydt L., **Renders L.**, and Fostier J. (2023) *Pan-genome de Bruijn graph using the bidirectional FM-index*. Talk presented at the 9th Workshop on Data Structures in Bioinformatics (DSB 2023), Delft, Holland, March 21-22.

- Depuydt L., **Renders L.**, Van de Vyver S., Veys L., Gagie T., and Fostier J. (2024) *B-move: faster bidirectional character extensions in a run-length compressed index*. Presented at the 24th International Workshop on Algorithms in Bioinformatics (WABI 2024), London, UK. In *Proceedings of the 24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, vol. 312, pp. 1–18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.WABI.2024.10.

### 1.8.3 Contributions to National Conferences

- **Renders L.**, Depuydt L., and Fostier J. (2022) *Approximate pattern matching using search schemes and in-text verification*. Pitch presented at the Faculty of Engineering and Architecture Research Symposium (FEARS 2022), Ghent, Belgium. Slide available at: http://doi.org/10.5281/zenodo.7401042.

### 1.8.4 Authors' Contributions

In publications where Luca Renders is a co-author, Lore Depuydt is the first author and primary contributor. These works build on the core functionalities of Columba, originally developed by L. Renders. While L. Depuydt implemented many of the new features required for these studies, L. Renders supported their integration into Columba, ensuring compatibility with existing functionalities. L. Renders also assisted in the technical development. In publications where L. Depuydt is a co-author, L. Renders is the first author and primary contributor. These works benefited from L. Depuydt's contributions, which included extending Columba's functionality for specific applications, assisting with result interpretation, and supporting the communication of findings. In either case, the first author is the main contributor to the work, while the second author (together with other authors) played a supporting role in designing and implementing algorithms and scripts, interpreting data, and preparing the manuscript.

# References

[1] A. Sanchez-Mazas. *A review of HLA allele and SNP associations with highly prevalent infectious diseases in human populations*. Swiss Medical Weekly, 150(1516):w20214, April 2020. doi:10.4414/smw.2020.20214.

[2] A. Szolek, B. Schubert, C. Mohr, M. Sturm, M. Feldhahn, and O. Kohlbacher. *OptiType: precision HLA typing from next-generation sequencing data*. Bioinformatics, 30(23):3310–3316, August 2014. doi:10.1093/bioinformatics/btu548.

[3] R. Dahm. *Friedrich Miescher and the discovery of DNA*. Developmental Biology, 278(2):274–288, 2005. doi:10.1016/j.ydbio.2004.11.028.

[4] J. D. Watson and F. H. Crick. *The structure of DNA*. In Cold Spring Harbor symposia on quantitative biology, volume 18, pages 123–131. Cold Spring Harbor Laboratory Press, 1953. doi:10.1101/sqb.1953.018.01.020.

[5] J. L. Fridovich-Keil. *Human Genome*. Encyclopedia Britannica, 25 Sep. 2024, 2024. Accessed: 2024-11-13. Available from: https://www.britannica.com/science/human-genome.

[6] J. M. Lee, H. M. Hammarén, M. M. Savitski, and S. H. Baek. *Control of protein stability by post-translational modifications*. Nature Communications, 14(1):201, 2023. doi:10.1038/s41467-023-35795-8.

[7] T. Shiina, K. Hosomichi, H. Inoko, and J. K. Kulski. *The HLA genomic loci map: expression, interaction, diversity and disease*. Journal of human genetics, 54(1):15–39, 2009. doi:10.1038/jhg.2008.5.

[8] H. Tettelin, D. Riley, C. Cattuto, and D. Medini. *Comparative genomics: the bacterial pan-genome*. Current Opinion in Microbiology, 11(5):472–477, 2008. Antimicrobials/Genomics. doi:10.1016/j.mib.2008.09.006.

[9] T. Wang, L. Antonacci-Fulton, K. Howe, H. A. Lawson, J. K. Lucas, A. M. Phillippy, A. B. Popejoy, M. Asri, C. Carson, M. J. Chaisson, et al. *The Human Pangenome Project: a global resource to map genomic diversity*. Nature, 604(7906):437–446, 2022. doi:10.1038/s41586-022-04601-8.

[10] K. Mullis, F. Faloona, S. Scharf, R. Saiki, G. Horn, and H. Erlich. *Specific enzymatic amplification of DNA in vitro: the polymerase chain reaction*. Cold Spring Harbor Symposia on Quantitative Biology, 51 Pt 1:263–273, 1986. doi:10.1101/sqb.1986.051.01.032.

[11] F. Sanger, A. Coulson, G. Hong, D. Hill, and G. Petersen. *Nucleotide sequence of bacteriophage λ DNA*. Journal of Molecular Biology, 162(4):729–773, 1982. doi:10.1016/0022-2836(82)90546-0.

[12] M. L. Metzker. *Sequencing technologies—the next generation*. Nature reviews genetics, 11(1):31–46, 2010. doi:10.1038/nrg2626.

[13] M. Wang. *Next-Generation Sequencing (NGS)*. In S. Pan and J. Tang, editors, Clinical Molecular Diagnostics, pages 305–327. Springer Singapore, Singapore, 2021. doi:10.1007/978-981-16-1037-0_23.

[14] 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, 12:1–14, 2011. doi:10.1186/gb-2011-12-2-r18.

[15] A. M. Wenger, P. Peluso, W. J. Rowell, P.-C. Chang, R. J. Hall, G. T. Concepcion, J. Ebler, A. Fungtammasan, A. Kolesnikov, N. D. Olson, et al. *Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome*. Nature biotechnology, 37(10):1155–1162, 2019. doi:10.1038/s41587-019-0217-9.

[16] H. Li and R. Durbin. *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

[17] 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, 10:1–10, 2009. doi:10.1186/gb-2009-10-3-r25.

[18] W. R. Pearson. *Using the FASTA Program to Search Protein and DNA Sequence Databases*, pages 307–331. Humana Press, Totowa, NJ, 1994. doi:10.1385/0-89603-246-9:307.

[19] P. J. A. 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, December 2009. doi:10.1093/nar/gkp1137.

[20] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup. *The Sequence Alignment/Map format and SAMtools*. Bioinformatics, 25(16):2078–2079, June 2009. doi:10.1093/bioinformatics/btp352.

[21] E. W. Dijkstra. *Why numbering should start at zero*. circulated privately, August 1982. Available from: http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

[22] ISO/IEC. *ISO/IEC 14882:2020: Programming Languages – C++*, 2020. Accessed: 2024-12-02. Available from: https://www.iso.org/standard/83626.html.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[24] 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. doi:10.1016/0022-2836(70)90057-4.

[25] T. F. Smith, M. S. Waterman, et al. *Identification of common molecular subsequences*. Journal of molecular biology, 147(1):195–197, 1981.

[26] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[27] D. A. Patterson and J. L. Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[28] P. Ferragina and G. Manzini. *Opportunistic data structures with applications*. In Proceedings 41st Annual Symposium on Foundations of Computer Science, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

[29] M. Burrows and D. Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. In DIGITAL SRC RESEARCH REPORT. Citeseer, 1994.

[30] 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 2009 IEEE International Conference on Bioinformatics and Biomedicine, pages 31–36, 2009. doi:10.1109/BIBM.2009.42.

[31] S. Vigna. *Broadword implementation of rank/select queries*. In International Workshop on Experimental and Efficient Algorithms, pages 154–168. Springer, 2008.

[32] G. Navarro. *Wavelet trees for all*. Journal of Discrete Algorithms, 25:2–20, 2014. 23rd Annual Symposium on Combinatorial Pattern Matching. doi:10.1016/j.jda.2013.07.004.

[33] C. Pockrandt, M. Ehrhardt, and K. Reinert. *EPR-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional FM-indices*, 2016. doi:10.1007/978-3-319-56970-3_12.

[34] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[35] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

[36] M. G. Maaß. *Linear Bidirectional On-Line Construction of Affix Trees*. In R. Giancarlo and D. Sankoff, editors, Combinatorial Pattern Matching, pages 320–334, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-45123-4_27.

[37] D. Strothmann. *The affix array data structure and its applications to RNA secondary structure analysis*. Theoretical Computer Science, 389(1):278 – 294, 2007. doi:10.1016/j.tcs.2007.09.029.

[38] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1*. Wiley, New York, 3rd edition, 1968.

[39] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[40] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert. *Optimum Search Schemes for approximate string matching using bidirectional FM-index*. arXiv preprint arXiv:1711.02035, 2017. doi:10.48550/arXiv.1711.02035.

[41] B. Langmead and S. L. Salzberg. *Fast gapped-read alignment with Bowtie 2*. Nature methods, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.

[42] 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, July 2009. doi:10.1101/gr.088823.108.

[43] D. Weese, M. Holtgrewe, and K. Reinert. *RazerS 3: Faster, fully sensitive read mapping*. Bioinformatics, 28(20):2592–2599, 08 2012. doi:10.1093/bioinformatics/bts505.

[44] T. H. Dadi, E. Siragusa, V. C. Piro, A. Andrusch, E. Seiler, B. Y. Renard, and K. Reinert. *DREAM-Yara: an exact read mapper for very large databases with short update time*. Bioinformatics, 34(17):i766–i772, September 2018. doi:10.1093/bioinformatics/bty567.

[45] E. Siragusa. *Approximate string matching for high-throughput sequencing*. PhD thesis, Freie Universität Berlin, 2015. doi:10.17169/refubium-15562.

[46] S. Marco-Sola, M. Sammeth, G. R, and P. Ribeca. *The GEM mapper: fast, accurate and versatile alignment by filtration*. Nature Methods, 9(December):1185–1188, 2012. doi:10.1028/nmeth.2221.

[47] H. Li. *BWT construction and search at the terabase scale*. Bioinformatics, page btae717, November 2024. doi:10.1093/bioinformatics/btae717.

# 2

# Dynamic Partitioning of Search Patterns for Approximate Pattern Matching using Search Schemes

*In this chapter, we introduce Columba, an open-source, lossless aligner built on search schemes. We present a new algorithm for dynamically partitioning search patterns, designed to work universally with any type of search scheme. Additionally, we compare the search space sizes across various search schemes and partitioning methods. Implementation details are discussed. Columba's performance is also benchmarked against another lossless alignment tool. This material was initially presented at the 2021 Research in Computational Molecular Biology Satellite Conference on Biological Sequence Analysis (RECOMB-Seq) and later selected for publication in iScience.*

**L. Renders, K. Marchal, and J. Fostier**

**Published in *iScience*, Volume 24, issue 7, 102687, https://doi.org/10.1016/j.isci.2021.102687**

**Abstract** Search schemes constitute a flexible and generic framework to describe how all approximate occurrences of a search pattern in a text can be found efficiently. We propose an algorithm for the dynamic partitioning of search patterns which can be universally applied to any kind of search scheme and demonstrate that this technique significantly reduces the search space. We present Columba, a software tool written

in C++, in which a multitude of search schemes are implemented. We discuss implementation aspects such as memory interleaving of Burrows-Wheeler Transform (BWT) representations and the reduction of redundancy that is inherently associated with the edit distance metric. Ultimately, we demonstrate that Columba has superior performance to the state-of-the-art. Using a single Central Processing Unit (CPU) core, Columba is able to retrieve all occurrences of 100 000 Illumina reads and their reverse complements within a maximum edit distance of four in the human genome in less than 3 minutes.

## 2.1 Introduction

Search schemes, paired with a bidirectional FM-index, provide a powerful framework for lossless approximate sequence alignment by significantly reducing the computational burden associated with pattern matching. For an in-depth explanation of these foundational concepts, readers are encouraged to refer to the introductory chapter (Sections 1.3 and 1.4).

Building on these principles, this chapter focuses on key challenges in optimizing the performance of search schemes and introduces a series of techniques aimed at improving both efficiency and scalability. These improvements are listed below:

1. **Dynamic Partitioning of Search Patterns**: We propose a novel algorithm for dynamically partitioning a search pattern $P$ based on its sequence content and its relation to the indexed text. By analyzing the pattern's composition and enlarging parts with a higher number of exact matches in the reference genome, this approach minimizes unnecessary exploration of low-likelihood areas. The dynamic partitioning method reduces the search space and runtime by up to 28% and 25%, respectively, compared to a uniform partitioning. Additionally, this technique is versatile and can be applied universally to any search scheme, providing consistent performance improvements.

2. **Memory Interleaving for the Burrows-Wheeler Transform (BWT)**: To address performance bottlenecks caused by memory access patterns, we introduce a memory interleaving strategy for the bit-vector representations of the BWT. By carefully organizing memory layouts to reduce cache misses during search trie traversal, this strategy accelerates search operations. Our evaluation shows that this interleaving strategy reduces runtimes by 35%.

3. **Redundancy Mitigation in Edit Distance Calculations**: We provide an analysis and discussion on how to maximally avoid redundancy that is inherently associated with edit distance computations. Avoiding this redundancy reduces the search space by up to 63% and the runtime by 52%.

4. **Integration and Tool Development in Columba**: These advancements collectively culminated in the development of the first version of **Columba**, an open-source tool for lossless sequence alignment written in standard C++11. Columba integrates dynamic partitioning, memory-efficient data structures, and

redundancy reduction to achieve substantial performance improvements over existing tools. Collectively, the three improvements above, reduce the runtime with a factor of almost 35%. Benchmarks show that Columba is nearly 4 times faster than **Bwolo** [1] for identifying all occurrences of 100 000 101 bp Illumina reads in the human reference genome with an edit distance of $k = 4$ errors. In addition to its performance, Columba supports a wide range of search schemes, including those based on the pigeonhole principle, $k + 1$ and $k + 2$ schemes from Kucherov et al., as well as the 01*0 seeds from Vroland et al. The tool is available at https://github.com/biointec/columba under the AGPL-3.0 license.

## 2.2 Dynamic Partitioning of Search Patterns

As noted by Kucherov et al. [2] the partitioning of $P$ into *equally sized* parts (called *uniform* partitioning) is not necessarily optimal. This is because different searches might enumerate a different number of strings during the search procedure. Using the search scheme for $k = 2$ errors from Example 1.24, it is clear that search $\mathcal{S}_0$ will *on average* be associated with a larger search space than $\mathcal{S}_1$ and $\mathcal{S}_2$, because $\mathcal{S}_0$ allows for two errors in the second part that is processed whereas $\mathcal{S}_1$ and $\mathcal{S}_2$ allow only for a single error. By increasing the size of $\mathcal{P}_0$, the number of exact matches of $\mathcal{P}_0$ will generally decrease. Consequently, search $\mathcal{S}_0$ will enumerate fewer candidate occurrences. Conversely, by decreasing the size of $\mathcal{P}_1$ and $\mathcal{P}_2$, the search space to explore by $\mathcal{S}_1$ and $\mathcal{S}_2$ will increase. Due to the asymmetry between searches, the decrease in search space can be larger than the increase, thus improving the overall performance of the search scheme. In their paper, Kucherov et al. propose a dynamic programming algorithm to find the optimal part sizes, using a model that assumes a random search text $T$ and a random search pattern $P$. In this work, we focus on the human genome as a reference genome and on search patterns that have a small edit distance to some subsequence of this reference genome. The assumption of randomness is not valid as the human genome has, unlike random sequences, a very complex repeat structure. Hence, we established optimal part sizes empirically using a first set of reads (see Section 2.5.1). We call this partitioning the *optimal static* partitioning. These optimal part sizes depend on characteristics of the search patterns (length, error rate and distribution) as well as the characteristics of the reference genome itself (repeat structure).

Even though these part sizes may be optimal *on average*, they are not necessarily optimal for each individual search pattern $P$. For example, depending on the sequence content of $P$ itself, some parts of $P$ might have a very low number of exact matches (or even no matches) whereas other parts might have a very high number of exact matches. In turn, this will again translate into an uneven workload among searches. By reducing the size of the parts with few exact matches while increasing the size of parts with many exact matches, it is again possible to achieve a global reduction in workload. Figure 2.1 illustrates this principle, and Appendix C provides a detailed analysis of a single read alignment to further demonstrate the impact of dynamic partitioning on search space reduction.

We propose an algorithm for the *dynamic partitioning* of $P$. The ideas are illus-

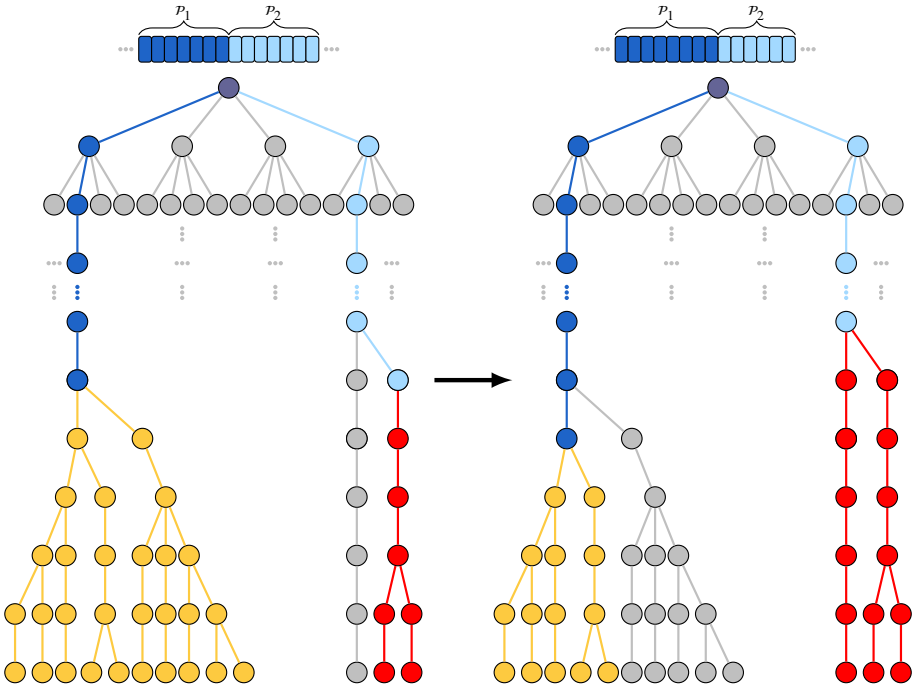


***Figure 2.1:*** *Diagram showing the effect of a different partitioning on the search space. The blue and light blue areas denote nodes explored exactly, while the yellow and red areas signify approximate explorations. In the initial configuration (left), the subtrie explored for approximate pattern matching after an exact match of $\mathcal{P}_1$ (yellow) is significantly larger than the subtrie explored after an exact match of $\mathcal{P}_2$ (red), despite both parts being the same size. The reconfigured partitioning on the right addresses this imbalance by increasing the length of $\mathcal{P}_1$ by one character, reducing the length of $\mathcal{P}_2$. This adjustment results in more balanced subtrie sizes for approximate pattern matching. Conversely, if the yellow subtrie had been smaller than the red subtrie, the dynamic partitioning would shift to enlarge $\mathcal{P}_2$, ensuring a more even distribution of the search space.*

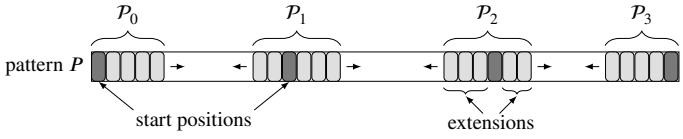


***Figure 2.2:*** *Dynamic partitioning of a search pattern $P$ with four parts: each part is initialized by matching a single character (dark gray squares). The part with the largest number of exact occurrences is extended by a single character (light gray squares), either to the left or to the right.*

trated in Figure 2.2. Each part is initialized by matching a single character. The position of that character is taken as the middle position of the corresponding uniformly sized parts, except for the first and last part where we take respectively the first and last character of $P$. During each next step, the part with the highest number of exact occurrences (and which can still be extended) is selected, and it is extended by a single character in the direction (left or right) of the adjacent part that has the fewest number of exact occurrences. This procedure is repeated until all characters of $P$ are assigned to some part. Intuitively, this greedy algorithm attempts to partition $P$ such that each part is associated with an equal number of exact matches. The actual partitioning that is obtained depends on the sequence content of $P$ itself and will therefore differ between search patterns, hence the name *dynamic partitioning*. The overhead imposed by dynamic partitioning is very small: the Suffix Array (SA)/BWT ranges of the exact matches of the parts of $P$ that emerge as a by-product of the procedure can be stored and are re-used during the execution of the search schemes. Indeed, efficient search schemes consist of searches that first involve the exact matching of some part of $P$. Note that the dynamic partitioning algorithm in general extends parts both to the left and right and therefore takes full advantage of the functionality offered by the bidirectional FM index.

In its basic form, the dynamic partitioning algorithm yields parts with a roughly equal number of exact matches, similar to what is expected *on average* from uniform partitioning. However, in order to obtain the best results, one should aim to balance the relative number of exact matches among parts such that they correspond to what is expected *on average* from the optimal static partitioning. The dynamic partitioning algorithm is easily adapted to this task as follows: 1) we initialize the first character of each part as the center position of the optimal static partitioning (again, except for the first/last part); 2) we assign per-part weights and balance the weighted number of exact matches among parts. Intuitively, the first modification allows certain parts to 'grow' more than others while the weights take into account the expected relative workload among searches. Again, we obtained these weights empirically using a first set of reads (see Section 2.5.1. Dynamic partitioning can reduce the search space by up to 18% compared to optimal static partitioning (see Section 2.5.2).

## 2.3 Memory Interleaving of Bit Vectors

To extend a partial match by a single character $c$, the FM index relies on $\text{Occ}[p, c]$ queries on the BWT to return the number of occurrences of character $c$ in the prefix $\text{BWT}[0, p[$ (see Section 1.3.2). Given our focus on DNA sequences, which have alphabet size $|\Sigma| = 4$, we realize this using four bit vectors $B_c$ with constant-time rank support. Then, $\text{Occ}[p, c] = B_c.\texttt{rank}_1(p) = \sum_{0 \leq i < p} B_c[i]$, i.e., the number of 1-bits in the first $p$ positions of $B_c$. For 64-bit CPU architectures, the rank9 algorithm [3] has attractive properties. This algorithm is detailed in Section 1.3.4.1. We recap here briefly. A bit vector of size $n$ bits is stored in $\lceil n/64 \rceil$ 64-bit words. For every $p$ that is a multiple of 512 bits, the pre-computed *first-level count* $\texttt{rank}_1(p)$ is stored as a 64-bit value. Each first-level count is associated with seven additional pre-computed *second-level*

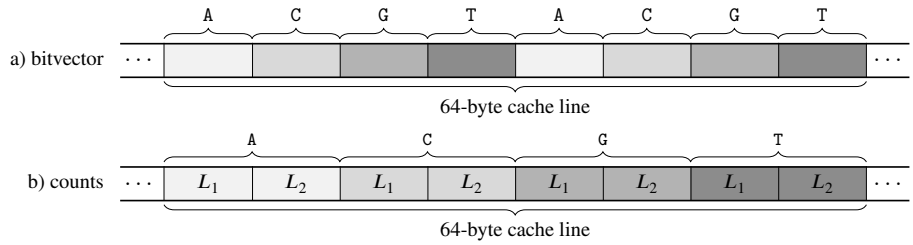


***Figure 2.3:*** *Interleaved storage of bit vectors $B_c$ and associated first- and second-level counts, for $\Sigma = \{A, C, G, T\}$. Four calls to* Occ$[p, c]$ *for fixed $p$ and all $c \in \Sigma$ require the data of only two cache lines.*

*counts* that contain values $\texttt{rank}_1(p + 64k)$ - $\texttt{rank}_1(p)$, for $1 \leq k \leq 7$. These second-level counts are stored as seven 9-bit values within a 64-bit word. Rank operations $\texttt{rank}_1(p)$ for arbitrary $p$ can then be answered in constant time by adding three contributions: 1) the appropriate first-level and 2) second-level counts; 3) a $\texttt{popcount}(w)$ instruction to count the number of 1-bits in $w$, the word of the bit vector that contains position $p$ and for which the bits at positions $p$ and higher were masked to zero. The memory overhead of the first- and second-level counts amounts to only 25% of the bit vector data.

Vigna [3] proposed to store the first- and second-level counts in an *interleaved* manner: the two words that hold corresponding first- and second-level counts are located next to each other in memory. When loading the cache line that contains first-level count information, the second-level count information is retrieved as well, thus reducing the number of cache misses. Gog and Petri [4] proposed to additionally interleave the pre-computed count information with the bit vector data itself. In that case, all information to answer a rank query is stored on either a single cache line, or two adjacent cache lines.

In the context of search schemes (or more generally: backtracking algorithms), the search trie is explored by extending a partial match with each character $c \in \Sigma$. Hence, within a short time duration, different calls to Occ$[p, c]$ are made for *fixed* $p$ and for all $c \in \Sigma$. In order to maximally fill a cache line (64 bytes or eight words on typical x86 architectures) with relevant information, we propose to interleave the data related to *different* bit vectors $B_c$ as shown in Figure 2.3. In the case of DNA sequences, all four calls to Occ$[p, c]$ with $c = \{A, C, G, T\}$ can then be answered using 12 words of data. By using 64-byte aligned vectors to store the interleaved bit vector data and pre-computed counts, we guarantee that only two cache lines are required for four Occ calls. For the same task, when using four non-interleaved bit vectors, at least four cache lines would be required when these individual bit vectors are stored using the scheme by Gog and Petri while eight cache lines would be required using Vigna's storage scheme. For large genomes, due to the fact that $p$ takes unpredictable values, most of these cache lines have to be retrieved from main memory, a task that requires ~100 ns (equivalent to 200-300 CPU cycles) on modern CPU architectures. Thus, reducing the number of cache lines that have to be retrieved from memory improves performance.

## 2.4 Reducing Redundancy for the Edit Distance Metric

The use of the edit distance metric inherently results in a certain degree of redundancy. An occurrence of a pattern $P$ may be reported multiple times with slightly different start and/or end positions in $T$. For example, if $P$ occurs as an exact match in $T$ and $k$ errors are allowed, then $\sum_{i=1\ldots k} 4i = 2k(k+1)$ redundant occurrences $O$ will be found for which the alignments between $P$ and $O$ have leading and/or trailing gaps in $P$ and/or $O$. Hence, the degree of redundancy increases rapidly with the number of errors allowed.

As searches of search schemes align a pattern $P$ part by part, the same redundancy occurs when dealing with partial matches of $P$. Consider a search $S = (\pi, L, U)$. If all partial occurrences up to part $\mathcal{P}_{\pi[i]}$ with an edit distance of at least $L[i]$ and at most $U[i]$ are reported, then an alignment procedure for part $\mathcal{P}_{\pi[i+1]}$ will be started for each such occurrence, even though some of them may be redundant. If the search procedure does not change direction in between parts $\mathcal{P}_{\pi[i]}$ and $\mathcal{P}_{\pi[i+1]}$, one can achieve this by increasing the width of the banded alignment matrix (AM) from $2U[i]+1$ to $2U[i+1]+1$ as exemplified in Figure 2.4 (left).

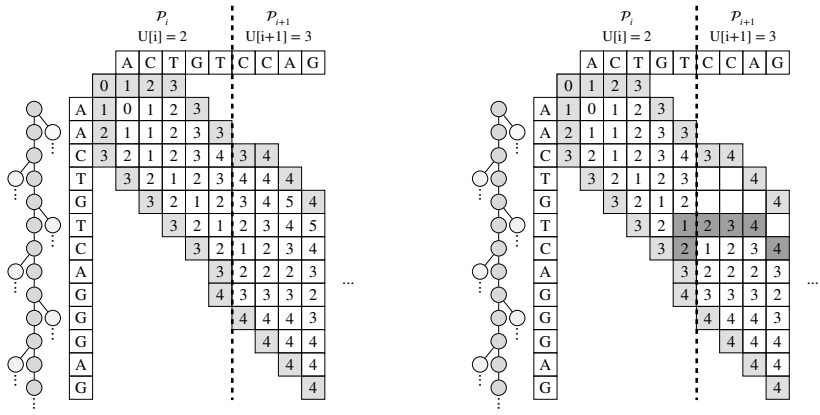


***Figure 2.4:*** *Left: a banded alignment matrix (AM) for which the allowed edit distance increases between parts. The grey-shaded cells are set during initialization whereas the white-shaded cells are completed during the execution of the search procedure. Search pattern $P$ is depicted horizontally whereas a branch of the index is depicted vertically. Right: the AM for part $\mathcal{P}_{\pi[i+1]}$ is initialized around the uppermost cluster center of the final column of part $\mathcal{P}_{\pi[i]}$.*

This approach can be improved upon by taking into account the actual edit distance values observed in the final column of the AM of part $\mathcal{P}_{\pi[i]}$. It may then be possible to reduce the width of the band of the next part $\mathcal{P}_{\pi[i+1]}$. For this, we introduce the notion of a *cluster*. We remind the reader that adjacent cells on a row or column of the AM differ by a value of at most one (see Lemma 3 of [5]). A cluster $\{s, c, e\}$ $(s \leq c < e)$ at column $j$ of the AM contains all elements $\mathrm{AM}[i, j]$ with $s \leq i < e$ for which it holds that $\mathrm{AM}[i, j] = \mathrm{AM}[c, j] + |c - i|$. Cell $\mathrm{AM}[c, j]$ is called the center of the cluster and can be thought of as a local minimum. Figure 2.5 (left) illustrates the clusters of a

column of the AM.

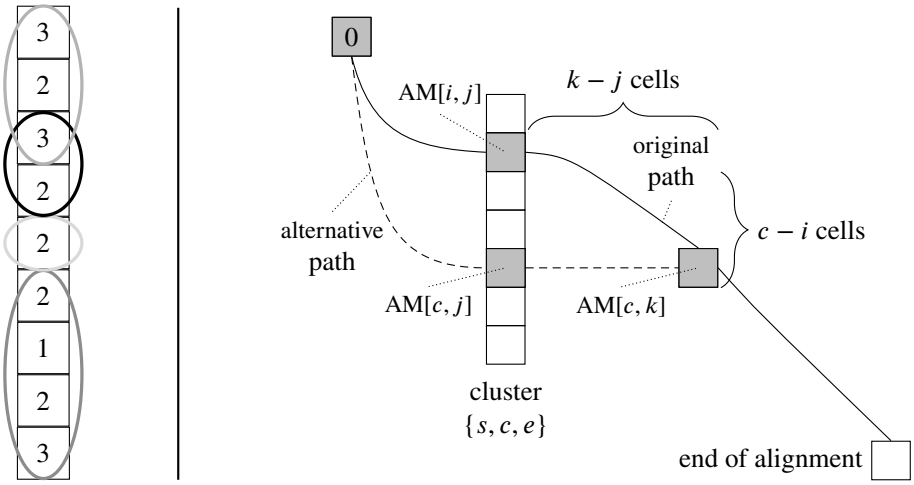


***Figure 2.5:*** *(left) The clusters of a column of the AM are encircled. Note that a cell can be part of two adjacent clusters and that a cluster can consist out of only a single cell. (right) Illustration of proof of Lemma 2.1, case 1.*

**Lemma 2.1.** *Consider the edit distance alignment of strings $X$ and $Y$ and consider the clusters of some column of the AM. For each optimal alignment path that passes through one or more cells of cluster $\{s, c, e\}$ and that does not pass through the center, an alternative optimal alignment path exists that passes through the center of that cluster.*

*Proof.* Case 1: the optimal alignment path passes through cell AM$[i, j]$, with $s \leq i < c$ (see Figure 2.5 right). By definition of a cluster, it holds that AM$[i, j]$ = AM$[c, j]$ + $(c - i)$. Assume that AM$[c, k]$ is the rightmost cell on row $c$ through which the optimal alignment path passes. Because the edit distance can only increase along an alignment path, AM$[c, k] \geq$ AM$[i, j]$, hence:

$$\text{AM}[c, k] \geq \text{AM}[c, j] + (c - i) \tag{2.1}$$

and

$$\text{AM}[c, k] \leq \text{AM}[c, j] + (k - j) \tag{2.2}$$

It follows from Equations 2.1 and 2.1 that $(k - j) \geq (c - i)$. The optimal alignment path between matrix cells AM$[i, j]$ and AM$[c, k]$ entails exactly $(k - j) - (c - i)$ gaps. It then follows that AM$[c, k] \geq$ AM$[i, j] + (k - j) - (c - i)$ and hence, AM$[c, k] \geq$ AM$[c, j] +$ $(k - j)$. Together with inequality (2), it follows that AM$[c, k]$ = AM$[c, j]$ + $(k - j)$. Therefore, an alternative optimal alignment path runs from the origin to AM$[c, j]$; from AM$[c, j]$ to AM$[c, k]$ and then proceeds in an identical manner as the original path.

Case 2: the optimal alignment path passes through cell AM$[i, j]$, with $c < i < e$. Because of the definition of a cluster, it holds that AM$[i, j]$ = AM$[c, j]$ + $(i - c)$. The

value AM$[i, j]$ can thus always originate from AM$[c, j]$ through a vertical path, as the difference in rows between these two cells is exactly $(i - c)$. Again, an alternative optimal alignment path exists from the origin to cell AM$[c, j]$; from AM$[c, j]$ to AM$[c, i]$; and then proceeds in an identical manner as the original path. □

We leverage Lemma 2.1 to save computations: when part $\mathcal{P}_{\pi[i]}$ is processed, we identify the uppermost cluster center in the last column of the alignment matrix of part $\mathcal{P}_{\pi[i]}$ that has a value between $L[i]$ and $U[i]$. The cells above this cluster center can be ignored: even when they take part in some optimal alignment path, there will always be an alternative optimal path that passes through the center cell (lemma 2.1). By ignoring these cells, the banded alignment matrix for the next part may be initialized with a reduced width and the number of edit cells to be computed in the next part is reduced. In Figure 2.4 (right), this is exemplified.

In case the search changes direction in between parts $\mathcal{P}_{\pi[i]}$ and $\mathcal{P}_{\pi[i+1]}$ the same redundancy problem arises. By reporting *all* approximate (partial) matches with an edit distance of at most $U[i]$, multiple, possibly redundant alignments will be started in the other direction. Again, this can be mitigated using the cluster concept. Two cases exist. First, assume part $\mathcal{P}_{\pi[i]}$ is either the leftmost or rightmost part of pattern $P$. This implies that the partial matches will no longer be extended in the current direction. Instead of reporting *all* partial occurrences with an edit distance of at most $U[i]$, we report only the partial match associated to the cluster center with the lowest value. If multiple such centers exist, then the uppermost one is reported, corresponding to the shortest (partial) match. In the second case, if part $\mathcal{P}_{\pi[i]}$ is neither the leftmost or rightmost part of pattern $P$, partial matches are first extended in the *other* direction before resuming the extension in the *present* direction. In this case, we cannot *a priori* assume that an optimal alignment path of the entire pattern $P$ (if such alignment exists) will pass through the cluster center with the lowest value. However, due to lemma 2.1, we know that such optimal alignment path (or an alternative, equivalent path) *will* pass through *one* of the cluster centers. Therefore, when part $\mathcal{P}_{\pi[i]}$ has been processed, we extend (in the other direction) *only* the FM-range corresponding to the *deepest* point in the present branch with the knowledge that this partial match will ultimately turn out to have an edit distance between the lowest and highest cluster center values. This information is valuable when checking the $L[i + 1]$ and $U[i + 1]$ bounds during the processing of part $\mathcal{P}_{\pi[i+1]}$.

## 2.5 Results and Discussion

### 2.5.1 Material and Experimental Setup

In all benchmarks, all occurrences of both strands of search patterns up to an edit distance (allowing substitutions and/or indels) of $k = \{1, 2, 3, 4\}$ were identified in the human reference genome (GRCh38) [6]. Non-ACGT characters (e.g., Ns) in the reference sequence were replaced by a randomly chosen nucleotide. Chromosomes were concatenated into a single string. It is therefore possible that a pattern is mapped

across the boundary of adjacent chromosomes. Such matches can easily be filtered during post-processing.

We sampled two sets of each 100 000 Illumina HiSeq 2000 reads (101 bp ) from a whole genome sequencing dataset (accession no. ERR194147). The first set of reads was used only to determine suitable parameters (as described in Section 2.2) for the different partitioning strategies, for the different search schemes and different values of $k$. All benchmark results were obtained using the second set of reads, demonstrating that the empirically derived parameters generalize well to other datasets with similar characteristics. We also provide results for 100 000 search patterns of length 50 bp, randomly sampled from long Pacific Biosciences reads (accession no. SRR1304331).

All results were obtained using a single core of a 24-core AMD EPYC 7451 CPU running at a base clock frequency of 3.1 GHz. Each simulation was repeated 20 times. We report both the average wall clock time as well as the standard deviation. We report the fraction of search patterns with at least one match as well as the total number of non-redundant occurrences. We label an occurrence as redundant when its starting position is at most $2k + 1$ nucleotides away from another match with an equal or lower edit distance.

### 2.5.2 Dynamic Partitioning of Search Patterns Reduces the Search Space

Table 2.1 shows the impact of different partitioning strategies for different values of the maximum allowed edit distance $k = \{1, 2, 3, 4\}$, using the search schemes proposed by Kucherov et al. with $k + 1$ parts [2]. Besides runtime, the number of nodes visited in the search trie as well as the number of computed elements in the banded edit distance matrix are shown. The former equals the number of times a partial match is extended by a single character $c$ (in either direction). In practice, this involves expensive random memory access and it is therefore a clear indication of intrinsic performance, regardless of the quality of implementation.

For $k = 1$ errors, the search scheme consists of two searches which are symmetric with respect to each other. Hence, the uniform and optimal static partitioning strategies both partition $P$ into two equally-sized parts and both show equal performance. The dynamic partitioning strategy reduces the number of nodes visited by 11.6%, demonstrating its ability to reduce the search space. However, because the overhead resulting from the dynamic partitioning procedure itself is larger than the gains from this smaller search space, the runtime increases slightly.

The search scheme for $k = 2$ errors is not symmetric, thus we expect the optimal static partitioning to differ from the uniform partitioning. For $|P| = 101$, we find optimal part sizes of 41, 29 and 31, respectively. Table 2.1 shows that applying optimal static partitioning reduces both the number of nodes visited and the runtime by 9.8%. The use of dynamic partitioning yields a notable reduction in search space and runtime of respectively 28.2% and 20.4%, both w.r.t. uniform partitioning. Clearly, the ability to partition patterns based on their sequence content results in a significant computational gain.

For $k = 3$ errors, the search scheme is only asymmetric in the lower bounds. We

***Table 2.1:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes by Kucherov et al. with $k + 1$ parts. The percentage values indicate the relative increase or decrease with respect to uniform partitioning. See also Table B.1- B.5.*

| Partitioning strategy | Wall clock time ± SD | | No. of nodes visited (search space) | | No. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Uniform | 6.73 ± 0.18 s | | 29 212 674 | | 45 697 288 | |
| Optimal static | 6.73 ± 0.18 s | (+0.0%) | 29 212 674 | (+0.0%) | 45 697 288 | (+0.0%) |
| Dynamic | 7.23 ± 0.14 s | (+7.4%) | 25 829 974 | (−11.6%) | 39 391 011 | (−13.8%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Uniform | 22.32 ± 0.15 s | | 104 476 617 | | 339 010 141 | |
| Optimal static | 20.14 ± 0.12 s | (−9.8%) | 94 289 677 | (−9.8%) | 282 148 209 | (−16.8%) |
| Dynamic | 17.76 ± 0.13 s | (−20.4%) | 74 978 120 | (−28.2%) | 228 389 750 | (−32.6%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Uniform | 55.19 ± 0.44 s | | 276 027 753 | | 1 068 947 201 | |
| Optimal static | 55.27 ± 0.40 s | (+0.1%) | 276 027 753 | (+0.0%) | 1 068 947 201 | (+0.0%) |
| Dynamic | 45.73 ± 0.39 s | (−17.1%) | 212 965 491 | (−22.8%) | 816 519 456 | (−23.6%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 215.37 ± 0.96 s | | 1 154 125 425 | | 4 941 009 260 | |
| Optimal static | 191.45 ± 1.46 s | (−11.1%) | 1 012 354 507 | (−12.3%) | 4 272 355 550 | (−13.5%) |
| Dynamic | 162.14 ± 0.99 s | (−24.7%) | 827 521 618 | (−28.3%) | 3 607 583 286 | (−27.0%) |

find that the optimal part sizes are uniform for $|P| = 101$, hence optimal static partitioning and uniform partitioning yield identical results. However, dynamic partitioning again reveals superior performance with a runtime reduction of 17.1%.

The search scheme for $k = 4$ is highly asymmetric. It consists of eight searches, three of which start from an exact match of part $\mathcal{P}_0$. Therefore, it is beneficial to increase the size of $\mathcal{P}_0$, such that the number of exact matches of this part is reduced. We find optimal part sizes of 27, 20, 15, 19 and 19 that yield a reduction in runtime of 11.1%. Dynamic partitioning, however, is able to further reduce runtime by 24.7% while the number of nodes visited is reduced by 28.3%, both w.r.t. uniform partitioning. Also for other search schemes, we found that dynamic partitioning reduces the search space and runtime to a similar extent (Tables B.1- B.5). The same holds for the search patterns sampled from Pacific Biosciences reads (Table B.6). We conclude that dynamic partitioning can be universally applied to boost the performance of search schemes.

***Table 2.2:*** *The wall clock time for mapping both strands of 100 000 Illumina patterns using interleaved and non-interleaved bit vectors for $k = 4$ errors and the search scheme by Kucherov et al. with $k + 1 = 5$ parts.*

| Partitioning type | Non-interleaved bit-vectors | Interleaved bit-vectors | |
|---|---|---|---|
| Uniform | 342.02 ± 2.71 s | 215.37 ± 0.96 s | (−37.2%) |
| Optimal static | 295.15 ± 2.93 s | 191.45 ± 1.46 s | (−35.1%) |
| Dynamic | 249.49 ± 2.19 s | 162.14 ± 0.99 s | (−35.0%) |

### 2.5.3 Memory Interleaving of Bit-vectors Reduces Runtime

We compare our proposed memory storage scheme for bit vectors to the scheme proposed by Vigna. For the sake of simplicity, we refer to these methods as *interleaved* and *non-interleaved*, respectively, even though also in Vigna's scheme, first- and second-level counts are interleaved. Table 2.2 shows the runtime for both approaches for the different partitioning strategies using the search scheme by Kucherov et al. for $k = 4$ errors and $k + 1 = 5$ parts. Runtime is considerably reduced by 35% to 37%. Note that the memory storage scheme is independent from the partitioning scheme and hence, both techniques can be combined. Collectively, for $k = 4$, they reduce runtime from 342 s (uniform partitioning with non-interleaved bit vectors) to 162 s (dynamic partitioning with interleaved bit vectors), a reduction of 53%.

### 2.5.4 Reducing redundancy for the Edit Distance Metric Reduces the Search Space

In Section 2.4, we provided techniques to reduce redundant computations and thus improve performance.

We compare an optimized implementation, which implements the ideas from Section 2.4, to a naive implementation. In this naive implementation an alignment procedure is started for each partial occurrence, even though some of them may be redundant. In Table 2.3, both approaches are compared for different values of $k$. For $k = 1$, the difference in runtime, number of nodes and matrix elements is negligible. For $k = 2$, the search space is reduced by 26%, leading to a reduction in runtime by 13%. For $k = 3$, even larger reductions of respectively 46% (search space) and 33% (runtime) are found. This is to be expected, as the redundancy grows quadratically with $k$. For $k = 4$, the optimized implementation reduces the search space by 63% and the runtime by 52%. We conclude that for higher values of $k$, an optimized implementation to handle redundancy associated with edit distance metric computations is essential to achieve high computational performance.

***Table 2.3:*** *Comparison of a naive and optimized strategy for handling the redundancy associated to the edit distance metric for different values of k. Both strands or 100 000 Illumina reads are mapped, using the search schemes by Kucherov et al. with $k+1$ parts, dynamic partitioning and interleaved bit vectors.*

| Edit distance redundancy | Wall clock time ± SD | | No. of nodes visited (search space) | | No. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Naive | 7.19 ± 0.10 s | | 26 102 601 | | 39 697 328 | |
| Optimized | 7.23 ± 0.14 s | (+0.6%) | 25 829 974 | (−1.0%) | 39 391 011 | (−0.8%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Naive | 20.50 ± 0.18 s | | 101 087 547 | | 297 657 004 | |
| Optimized | 17.76 ± 0.13 s | (−13.4%) | 75 164 043 | (−25.6%) | 227 670 435 | (−23.5%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Naive | 67.72 ± 0.52 s | | 393 769 104 | | 1 389 115 919 | |
| Optimized | 45.73 ± 0.39 s | (−32.5%) | 212 965 491 | (−45.9%) | 816 519 456 | (−41.2%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Naive | 335.67 ± 1.94 s | | 2 247 115 246 | | 8 250 593 095 | |
| Optimized | 162.14 ± 0.99 s | (−51.7%) | 827 521 618 | (−63.2%) | 3 607 583 286 | (−56.3%) |

### 2.5.5 Columba has Higher Computational Performance than State-of-the-art Tools

We benchmarked the different search schemes that have been proposed in literature in our implementation (called Columba). Kucherov et al. proposed schemes with $k+1$ and $k+2$ parts [2]. Kianfar et al. generated optimal search schemes for the Hamming distance metric [7]. For $k = 3$ and $k = 4$ errors, they contain searches that already allow one or two errors in the first part of $P$ that is matched. Hence, we found that these search schemes are not competitive when the edit distance metric is used. From the same research group, Pockrandt et al. derived an alternative search scheme for $k = 4$ errors referred to as $\text{Man}_{Best}$ [8]. We also include the search schemes based on the pigeonhole principle as well as those based on the 01*0 seeds [1, 7]. For the latter, a dedicated tool called Bwolo was developed. In other papers, Bwolo was found to be the fastest available method for the edit distance metric [7, 8]. For all search schemes, dynamic partitioning of search patterns was used with optimal parameters. We also compared our implementation to the implementation of search schemes in SeqAn 3. In our hands, the runtime was found to be orders of magnitude larger than our implementation. Therefore, no results for SeqAn 3 are reported.

Table 2.4 lists the runtimes. For efficient search schemes, Columba shows superior performance to Bwolo: for $k = 4$ errors and using the Kucherov $k+1$ search scheme, Columba is almost four times faster than Bwolo. Particularly, even when Bwolo's 01*0 search strategy is used within Columba, a significant performance difference is revealed. Bwolo uses a unidirectional search index whereas Columba relies on a bidirectional search functionality. Therefore, Bwolo has to perform in-text verification to

***Table 2.4:*** *The runtime for different search schemes that we support in our tool Columba versus a state-of-the-art tool Bwolo for different values of k. For all search schemes, dynamic partitioning was used.*

| Search scheme/ Tool | $k = 1$ | $k = 2$ | $k = 3$ | $k = 4$ |
|---|---|---|---|---|
| Pigeonhole princ. | 7.13 ± 0.04 s | 24.13 ± 0.15 s | 144.88 ± 0.99 s | 722.21 ± 5.72 s |
| Kucherov $k + 1$ | 7.23 ± 0.14 s | 17.76 ± 0.13 s | 45.73 ± 0.39 s | 162.14 ± 0.99 s |
| Kucherov $k + 2$ | 7.23 ± 0.09 s | 20.93 ± 0.12 s | 58.61 ± 0.35 s | 195.52 ± 1.57 s |
| 01*0 principle | 7.22 ± 0.09 s | 18.50 ± 0.10 s | 68.67 ± 0.38 s | 241.17 ± 2.07 s |
| Kianfar | 7.11 ± 0.06 s | 17.05 ± 0.11 s | 152.54 ± 1.19 s | 1994.92 ± 19.69 s |
| $\text{Man}_{Best}$ | n/a | n/a | n/a | 192.56 ± 1.35 s |
| Bwolo | 12.68 ± 0.37 s | 35.87 ± 0.25 s | 123.60 ± 0.33 s | 602.14 ± 0.77 s |

validate candidate matches, whereas Columba performs the matching entirely within the index structure itself. Nevertheless, as also noted in [7], in-text verification can in certain cases be faster.

Interestingly, in our implementation and using our dataset, the original $k+1$ search schemes proposed by Kucherov et al. yield the highest performance. Nevertheless, the performance difference with the $\text{Man}_{Best}$, Kucherov $k + 2$ and 01*0 schemes is relatively small and other datasets may yield different conclusions. Remarkably, the performance differences between these four efficient search schemes are about as large as the performance differences between partitioning strategies. Therefore, it might be worthwhile to further investigate novel partitioning strategies as well as novel search schemes.

We also benchmarked the runtime of Columba with uniform partitioning, non-interleaved bit vectors and a naive edit distance implementation (i.e., without any of the improvements proposed in this paper) for $k = 4$ and using the search scheme by Kucherov et al. with $k + 1 = 5$ parts. This resulted in a wall clock time of 560.64 s, almost 3.5 times slower than the runtime of 162.14 s of Columba with all improvements in place (see Table 2.1). Note that Columba, like most alignment tools, can easily take advantage of multi-core and/or multi-CPU architectures by aligning reads on different cores in parallel.

## 2.5.6 Summary

In this chapter we made three contributions, which together form Columba. We proposed an algorithm for the *dynamic partitioning* of a search pattern $P$ based on its sequence content. We demonstrate that for the task of mapping 100 bp Illumina reads to the human reference genome, this technique reduces the search space and runtime by up to 28% and 25%, respectively, compared to a uniform partitioning of $P$. Dynamic partitioning can be universally applied to any kind of search scheme to boost its performance. Additionally, we proposed a new strategy to interleave bit vector representations of the Burrows-Wheeler transform in memory that is specifically tailored to

search schemes. This leads to fewer cache misses and reduces runtime by 35%. Again, this technique can be universally applied to all search schemes. Finally, we provided an analysis and discussion on how to maximally avoid redundancy that is inherently associated with edit distance computations. Avoiding this redundancy reduces the search space by up to 63% and the runtime by 52%. Collectively, these techniques reduce the runtime with a factor of almost 3.5. Columba supports different search schemes: the pigeonhole-based schemes, the search schemes with $k + 1$ and $k + 2$ parts as proposed by Kucherov et al. [2], the schemes proposed by Kianfar et al. [7] and the 01*0 seeds by Vroland et al. [1]. We demonstrated that our implementation is almost four times faster than Bwolo [1] for the task of identifying all occurrences of 100 bp Illumina reads in the human reference genome within an edit distance of $k = 4$ errors. Columba is available at https://github.com/biointec/columba under AGPL-3.0 license.

## 2.6 Limitations of the Study

This work considers only lossless approximate pattern matching algorithms. Columba is not compared against state-of-the-art tools that use lossy approximate pattern matching algorithms. Only the edit distance (i.e., Levenshtein) metric is considered and not the Hamming distance metric. Columba cannot find occurrences under a more generic scoring scheme with arbitrary match, mismatch and gap scores/penalties. At the time of this research, search schemes had been proposed in literature for up to $k = 4$ errors. It remained still an open research question how efficient search schemes for higher values of $k$ could be designed. We have benchmarked Columba using only patterns of length 101 (Illumina) and length 50 (Pacific Biosciences). Performance and relative performance differences w.r.t. other tools may vary for other pattern lengths. We have benchmarked only using the human reference genome as a search text.

## Acknowledgments

The authors received no specific funding for this work.

### Data and Code Availability

The datasets used in this study are derived from publicly available data. The first dataset consists of 100 000 randomly sampled reads from an Illumina sequencing experiment (EBI ENA, accession no. ERR194147)[1]. The second dataset consists of 100 000 subreads of length 50 randomly sampled from a Pacific Biosciences sequencing experiment (EBI ENA, accession no. SRR1304331)[2]. The code generated during this study is available at https://github.com/biointec/columba.

[1] https://github.com/biointec/columba/releases/download/example/sampled_illumina_reads.fastq
[2] https://github.com/biointec/columba/releases/download/example/sampled_pacbio_seeds.fastq

## Author Contributions

L.R. and J.F. designed and implemented the algorithms. L.R. performed all benchmarks. K.M. and J.F. supervised the study. All authors have written and approved the manuscript.

## Declaration of interests

The authors declare no competing interests.

# References

[1] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[2] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[3] S. Vigna. *Broadword Implementation of Rank/Select Queries*. In C. C. McGeoch, editor, Experimental Algorithms, pages 154–168, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-68552-4_12.

[4] S. Gog and M. Petri. *Optimized Succinct Data Structures for Massive Data*. Softw. Pract. Exper., 44(11):1287–1314, November 2014. doi:10.1002/spe.2198.

[5] W. J. Masek and M. S. Paterson. *A faster algorithm computing string edit distances*. Journal of Computer and System Sciences, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

[6] V. Schneider, T. Graves-Lindsay, K. Howe, N. Bouk, H.-C. Chen, P. Kitts, T. Murphy, K. Pruitt, F. Thibaud-Nissen, D. Albracht, R. Fulton, M. Kremitzki, V. Magrini, C. Markovic, S. McGrath, K. Steinberg, K. Auger, W. Chow, J. Collins, and D. Church. *Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly*. Genome Research, 27, 2017. doi:10.1101/gr.213611.116.

[7] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert. *Optimum Search Schemes for approximate string matching using bidirectional FM-index*. arXiv preprint arXiv:1711.02035, 2017. doi:10.48550/arXiv.1711.02035.

[8] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

# 3

# Approximate Pattern Matching Using Hybrid In-index Matching/In-text Verification

*In this chapter, we expand on Columba, introduced in the previous chapter, with the presentation of Columba 1.1. We delve into the implementation of bit-parallel, pairwise alignment and its application for in-text verification, briefly touched upon at the end of the previous chapter. Additionally, we explore a manual adaption of search schemes for enhanced performance. Columba 1.1's speed is benchmarked against both lossy and lossless alignment tools. The material in this chapter was first presented at the 2022 International Workshop of Bioinformatics and Biomedical Engineering in Gran Canaria, Spain, and is based on the accompanying conference paper.*

**L. Renders, L. Depuydt, and J. Fostier.**

**Published in Lecture Notes in Computer Science (LNBI, volume 13347), doi:10.1007/978-3-031-07802-6_36**

**Abstract** Search schemes enable the efficient identification of all approximate occurrences of a search pattern in a text. Using a bidirectional FM-index, search schemes describe how to explore the search space in such a way that runtime is minimized. Even though in-index matching has an optimal time complexity, relatively expensive random memory access is required for elementary operations on the FM-index. We

analyze to what extent in-index matching can be complemented with in-text verification where a candidate occurrence is directly validated in the text using a bit-parallel, pairwise alignment procedure. We find that hybrid in-index/in-text matching can reduce the runtime by more than a factor of two, compared to pure in-index matching. We present Columba 1.1, an open-source (AGPL-3.0 license) software tool written in C++ that efficiently implements these ideas. Using a single CPU core, Columba 1.1 can identify, within a maximum edit distance of four, all occurrences of 100 000 Illumina reads (150 base-pair (bp)) in the human reference genome in roughly half a minute. This significantly outperforms existing, state-of-the-art tools.

## 3.1 Introduction

In Chapter 2, we proposed Columba, an efficient software tool for lossless approximate pattern matching using arbitrary search schemes. We proposed an algorithm for the dynamic partitioning of search patterns to further reduce the search space and used an efficient memory layout for the data structures that underlie the FM-index. In this chapter, we further build upon this work and we make the following contributions:

1. We adapted the search schemes by Kucherov et al. with $k + 1$ parts by imposing more stringent lower bounds on the cumulative number of errors in the different parts of the search pattern while maintaining the guarantee that all possible error distributions are covered. These adapted search schemes reduce the runtime by nearly 15%.

2. We adopt the bit-parallel, pairwise alignment algorithm by Hyyrö [1]. This algorithm is used to accelerate edit distance computations during in-index matching. Additionally, it is applied to in-text verification where a candidate occurrence of the search pattern is assessed directly in the text $T$. We show that using hybrid in-index matching/in-text verification can reduce the runtime by half compared to using only in-index matching.

3. We developed Columba 1.1, an open-source implementation in standard C++11 in which the above techniques were implemented. We demonstrate that our implementation is several times faster than other state-of-the-art lossless alignment algorithms such as GEM [2] and Bwolo [3] for the task of identifying all occurrences of 150 bp Illumina reads in the human reference genome within an edit distance of $k = 4$. We show that Columba 1.1 is faster than Burrows-Wheeler Aligner (BWA) in mem mode for $k = 1, 2$ and 3 and has a similar runtime for $k = 4$. Columba 1.1 is available at https://github.com/biointec/columba under AGPL-3.0 license.

This chapter is organized as follows. In Section 3.2 we introduce the manually adapted search schemes that are used in this chapter. In Sections 3.3 and 3.4, we provide the key algorithms for bit-parallel edit distance computations and their application to in-text verification, respectively. Finally, Section 3.5 provides performance

benchmarks of Columba 1.1 as well as existing state-of-the-art tools. Sections 3.2, 3.3 and 3.4 rely on the definitions of search schemes and a (bidirectional) FM-index. We advise the reader to refer to respectively Section 1.4 and 1.3 for more details about these concepts.

## 3.2 Adapted Search Schemes

In the previous chapter, we concluded that the search schemes by Kucherov et al. with $p = k + 1$ parts showed the best performance for the task of identifying occurrences of Illumina reads in the human reference genome under an edit distance constraint. However, it appears that for some searches $S = (\pi, L, U)$, the lower bound array $L$ can be made more stringent, while maintaining the guarantee that collectively, all searches within the search scheme cover all possible error distributions over a pattern. Recall that when part $\mathcal{P}_i$ has been processed, the cumulative number of errors must be between $L[i]$ and $U[i]$. The benefit of the adapted search schemes is twofold: 1) if fewer error distributions of a search pattern are covered by multiple searches, the number of redundant occurrences decreases, reducing the time to filter them and 2) by making the lower bounds more stringent, the search space that needs to be explored decreases. The original and adapted search schemes are presented in Table 3.1.

***Table 3.1:*** *The original search schemes by Kucherov et al. for $p = k + 1$ parts and our adapted search schemes for $k = \{1, 2, 3, 4\}$ errors. Changes are highlighted in bold.*

| $k$ | Original | Adapted |
|---|---|---|
| 1 | (01, 00, 01); (10, 01, 01) | (01, 00, 01); (10, 01, 01) |
| 2 | (012, 000, 022); (210, 000, 012); (102, 001, 012) | (012, 0**12**, 022); (210, 000, 012); (102, 001, 012) |
| 3 | (0123, 0000, 0133); (1023, 0011, 0133) (2310, 0000, 0133); (3210, 0011, 0133) | (0123, 000**2**, 0133); (1023, 0**11**3, 0133) (2310, 0000, 0133); (3210, 0**1**11, 0133) |
| 4 | (01234, 00000, 02244); (43210, 00000, 01344); (10234, 00133, 01334); (01234, 00133, 01334); (32410, 00011, 01244); (21034, 00013, 01244); (10234, 00124, 01244); (01234, 00034, 00444); | (01234, 0000**2**, 02244); (43210, 00000, 01344); (10234, 0**133**4, 01334); (01234, 00**33**4, 01334); (32410, 00**11**1, 01244); (21034, 001**1**3, 01244); (10234, 0**12**24, 01244); (01234, 00**34**4, 00444) |

## 3.3 Bit-parallel Edit Distance Computation

In Section 1.2.3.1, we introduced the concept of edit distance computations. We now provide a brief recap: the edit distance between two sequences $S_1$ and $S_2$ of lengths $m$ and $n$, respectively, is the minimum number of insertions, deletions, or substitutions required to transform one sequence into the other. This distance can be computed in $O(mn)$ time using a dynamic programming algorithm that constructs an $(m+1)\times(n+1)$ matrix $D$. Each element $D[i, j]$ in this matrix represents the edit distance between the prefixes $\text{pre}_{S_1}(i)$ and $\text{pre}_{S_2}(j)$. The values $D[i, j]$ are then determined using the following recurrence relation, which uses the indicator function as defined on page 20:

$$D[m,n] = \begin{cases} \max(m,n) & \text{If } \min(m,n) = 0 \\ \min \begin{cases} D[m-1,n]+1 & \text{(deletion)} \\ D[m-1,n-1]+1 & \text{(insertion)} \\ D[m-1,n-1]+I & \text{(match/subst.)} \end{cases} & \text{Otherwise} \end{cases}$$

The oldest description of this algorithm is by Vintsyuk [4] in 1968; it has been independently rediscovered by others (see e.g. [5] and the references therein). Myers [6] improved the time complexity to $O(mn/w)$, where $w$ denotes the computer word size ($w = 64$ for most CPU architectures). The core idea is to leverage bit-level parallelism to compute multiple values of matrix $D$ simultaneously. Inspired by Myers work, Hyyrö [7] proposed a slightly more efficient bit-parallel algorithm. We first provide a brief description of this algorithm. Next, we describe our specific adaptations.

### 3.3.1 Hyyrö's Bit-Parallel Algorithm

Adjacent elements within any row or column of matrix $D$ differ by at most a value of 1, i.e., for all $i, j$: $D[i,j] - D[i,j-1] \in \{-1,0,1\}$ and $D[i,j] - D[i-1,j] \in \{-1,0,1\}$ (see [8], lemma 3). Similarly, for adjacent elements on a diagonal, it holds that $D[i,j] - D(i-1,j-1) \in \{0,1\}$. Rather than computing the values of $D$ directly, each row $i$ is encoded by five delta vectors $\mathrm{VP}_i$, $\mathrm{VN}_i$, $\mathrm{HP}_i$, $\mathrm{HN}_i$, and $\mathrm{D0}_i$. These delta vectors are stored as bit vectors (i.e., a sequence of 0s and 1s) and are defined as follows:

1. The vertical positive delta vector: $\mathrm{VP}_i[j] = 1 \iff D[i,j] - D[i-1,j] = 1$
2. The vertical negative delta vector: $\mathrm{VN}_i[j] = 1 \iff D[i,j] - D[i-1,j] = -1$
3. The horizontal positive delta vector: $\mathrm{HP}_i[j] = 1 \iff D[i,j] - D[i,j-1] = 1$
4. The horizontal negative delta vector: $\mathrm{HN}_i[j] = 1 \iff D[i,j] - D[i,j-1] = -1$
5. The diagonal zero delta vector: $\mathrm{D0}_i[j] = 1 \iff D[i,j] - D[i-1,j-1] = 0$

The bits $\mathrm{HP}_i[j]$ and $\mathrm{HN}_i[j]$ encode the value $D[i,j] - D[i,j-1]$. The latter equals either 1 (when $\mathrm{HP}_i[j] = 1$), -1 (when $\mathrm{HN}_i[j] = 1$), or 0 (when both $\mathrm{HP}_i[j] = 0$ and $\mathrm{HN}_i[j] = 0$). Similarly, $\mathrm{VP}_i[j]$ and $\mathrm{VN}_i[j]$ encode the value $D[i,j] - D[i-1,j]$. Therefore, because $D[0,0]$ is known (often 0), all other values $D[i,j]$ can be inferred from the delta vectors.

The key advantage of using the delta vectors is that they can be computed in a bit-parallel manner as shown in Algorithm 8:

**Algorithm 8:** Bit-parallel computation of the delta vectors at row $i$ from those at row $i-1$

$\mathrm{D0}_i \leftarrow (((\mathrm{M}_{S_1[i-1]} \,\&\, \mathrm{HP}_{i-1}) + \mathrm{HP}_{i-1}) \,\hat{}\, \mathrm{HP}_{i-1}) \mid \mathrm{M}_{S_1[i-1]} \mid \mathrm{HN}_{i-1}$
$\mathrm{VP}_i \leftarrow \mathrm{HN}_{i-1} \mid \sim(\mathrm{D0}_i \mid \mathrm{HP}_{i-1})$
$\mathrm{VN}_i \leftarrow \mathrm{D0}_i \,\&\, \mathrm{HP}_{i-1}$
$\mathrm{HP}_i \leftarrow (\mathrm{VN}_i << 1) \mid \sim(\mathrm{D0}_i \mid (\mathrm{VP}_i << 1))$
$\mathrm{HN}_i \leftarrow (\mathrm{D0}_i \,\&\, (\mathrm{VP}_i << 1))$

Here, the symbols &, |, ˆ, ∼ and << denote the bitwise AND, OR, XOR, NOT and left shift operators. $\mathrm{M}_{S_1[i-1]}$ is a match vector (again a bit vector) that indicates which positions in $S_2$ match character $S_1[i-1]$. The four match vectors $M_A, M_C, M_G$ and $M_T$ are pre-computed. For the exact details of Algorithm 8, we refer to [7].

### 3.3.2 Bit-Parallel Banded Alignment

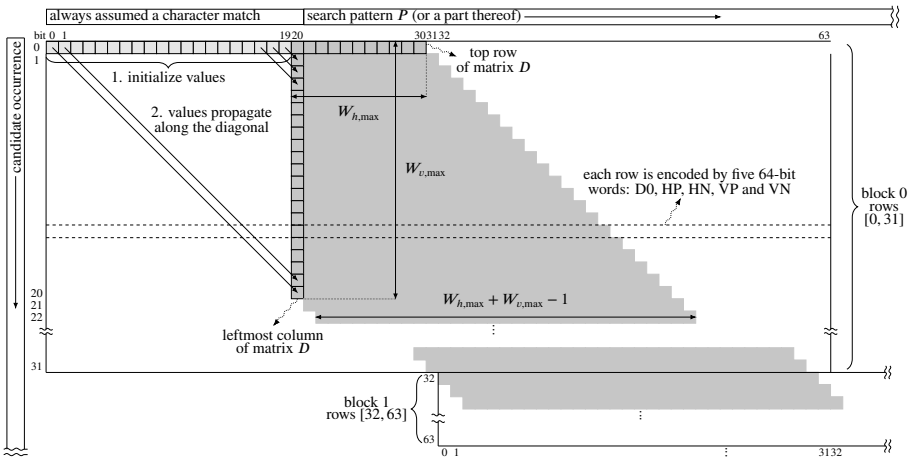


***Figure 3.1:*** *Layout of the banded dynamic programming matrix D as 64-bit words.*

In the context of this work, we want to identify approximate occurrences within a distance of at most $k$ edit operations of search pattern $P$. As explained in Section 1.3.3.1, computations can be restricted to those elements $D[i, j]$ for which $|i-j| \leq k$, i.e., within a band along the diagonal. Each row (or column) of matrix $D$ thus contains at most $2k + 1$ values to compute. For this problem of banded alignment, Hyyrö proposed a bit-parallel algorithm [7]. Our implementation is heavily influenced by these ideas but uses a different layout of bit vectors. It is described below.

The global layout of the banded dynamic programming matrix $D$ is depicted in Figure 3.1. Search pattern $P$ is the 'horizontal' sequence while candidate occurrence $O$ is the 'vertical' sequence. The FM-index spells out candidate occurrences character by character, therefore, we leverage bit-parallel computations at the level of *rows* of $D$. During in-index searching, candidate occurrences are generated by a depth-first exploration of the search trie. To support backtracking, the delta vectors of each row are kept in a stack data structure.

Our implementation can compute edit distance values up to $k = 10$ for sequences of arbitrary length. Because $k$ is sufficiently small, a single 64-bits word can be used to represent a delta vector and all computations per row are done in $O(1)$ time. Support for larger values of $k$ could easily be achieved by representing a delta vector by multiple words, at the cost of some loss of performance. Rows are grouped into *blocks* of 32 rows each. At each next block, the delta vectors are shifted by 32 bit positions such

that they overlap all relevant values of the banded dynamic programming matrix (gray-shaded cells in Figure 3.1). For each block, four match vectors $M_c$ ($c = \{\text{A, C, G, T}\}$) are pre-computed to indicate character matches between $c$ and the overlapping positions of $P$. At each row $i$, we also keep track of the value $D[i, i]$. Using the $\text{D0}_i$ delta vector, $D[i, i]$ can easily be computed from $D[i-1, j-1]$. The knowledge of $D[i, i]$ and the $\text{HP}_i$ and $\text{HN}_i$ delta vectors allows for the computation of any value $D[i, j]$. By using population count ('`popcount`') instructions, this can be achieved in $O(1)$ time. Finally, we adopted Hyyrö's algorithm to evaluate in a bit-parallel manner whether all values in a row exceed the maximum edit distance threshold $k$. This signals the backtracking algorithm that the current candidate occurrence $O$ should no longer be extended and that the search procedure should backtrack and explore a different branch of the search trie. For details on this algorithm, we refer to [1].

### 3.3.3 Matrix Initialization

Traditionally, the first row and column of matrix $D$ are initialized with gap penalties (i.e., $D[i, 0] = i$ and $D[0, j] = j$) in the case of global alignment, or with zero values (i.e., $D[i, 0] = 0$ and/or $D[0, j] = 0$) in case of semi-global alignment. For our use case of search schemes, we need to be able to initialize the leftmost column of $D$ with $2k + 1$ *arbitrary* values. Indeed, using search schemes, search pattern $P$ is matched part by part. Therefore, assuming left-to-right matching, when matching part $\mathcal{P}_i$, the first column of $D$ should be initialized with the values from the last column of the matrix of part $\mathcal{P}_{i-1}$ in order to continue the alignment.

In the bit-parallel implementation, the initialization of the first row of $D$ is straightforward: we set the appropriate value for $D(0, 0)$ (e.g., $D(0, 0) = 0$) and encode the other values $D[0, j]$ using the $\text{HP}_0$ and $\text{HN}_0$ delta vectors. For example, to encode $D[0, j] = j$, we set $\text{HP}_0[j] = 1$ and $\text{HN}_0[j] = 0$ for $j = 1 \dots k$.

To initialize the first column of $D$ with arbitrary values, we append dummy columns with a 'negative' column index to $D$ (illustrated in a lighter shade of gray in Figure 3.1). Again, we use the $\text{HP}_0$ and $\text{HN}_0$ delta vectors to encode the part of the first row of $D$ with negative column indexes such that $D[0, i]$ equals the desired value for $D[i, 0]$. By consistently assuming a character match at negative column indexes, each value $D(0, -i)$ propagates along a diagonal, eventually ensuring that $D[i, 0]$ is assigned its correct value. This is easily achieved by setting 1-bits in the corresponding part of $M_c$ for all $c = \{\text{A, C, G, T}\}$. Even in the presence of backtracking, the elements $D[i, 0]$ will always be computed correctly. Because the computations for the negative column indexes are handled within the same 64-bit word as the regular column indexes, this procedure imposes no computational overhead.

Because we support a maximum allowed edit distance of 10, we require at most $W_{h,\max} = 11$ elements at the top row of $D$ (e.g., to encode the values $\{0, 1, 2, \dots, 10\}$ and at most $W_{v,\max} = 21$ elements at the leftmost column of $D$ (e.g., to encode the values $\{10, \dots, 1, 0, 1, \dots, 10\}$). Thus, the parts of the delta vectors that *could* contain relevant values are indicated in a darker shade of gray in Figure 3.1. Depending on the use-case (the actual allowed edit distance $k \leq 10$, and how precisely matrix $D$ is initialized) only a subset of these cells will effectively contain relevant data.

## 3.4 In-Text Verification

In principle, search schemes rely purely on in-index matching: using the bidirectional FM-index, candidate occurrences $O$ of a search pattern $P$ are spelled out character by character. Extending a candidate occurrence by a single character ultimately translates into rank operations on bitvectors (see Section 1.3.4.1 and Section 2.3 for more details). Collectively, these rank operations lead to a random memory access pattern. The expression *random memory access* refers to the fact that the memory access pattern is inherently unpredictable, and hence, may suffer from a large number of cache misses. Consequently, extending a candidate occurrence by a single character is a relatively expensive operation: Pockrandt et al. estimated at least 100 CPU clock cycles per character [9].

At all times during the spelling of a candidate occurrence $O$, a range $[b, e[$ over the suffix array is maintained that refers to the starting positions of each instance of $O$ in $T$. Thus, at any point, the size of the range $e - b$ corresponds to the number of times $O$ occurs in $T$. This number of instances decreases monotonically when more characters are added to $O$.

When the value $e - b$ becomes small, it can be beneficial to abandon the in-index matching procedure and to verify each of the instances of $O$ directly in $T$ using the previously described pairwise alignment procedure. As detailed in Section 3.3, pairwise alignment can be performed efficiently using bit-parallel techniques in a cache-friendly manner. In contrast, when the value $e - b$ is large, in-index matching is more computationally advantageous, because all instances of $O$ of in $T$ are handled simultaneously by the FM-index.

In our implementation, part $\mathcal{P}_{\pi[0]}$ is always matched using the FM-index. In practice, with efficient search schemes, matching $\mathcal{P}_{\pi[0]}$ always entails an exact pattern matching procedure (see for example the search schemes in Table 3.1). From that point onwards, whenever the value $e - b$ becomes smaller than or equal to a pre-defined threshold $t$ (referred to as the 'tipping point'), candidate occurrence $O$ is no longer extended using the index and the search procedure switches to in-text verification, via (a) look-up(s) in the suffix array. When all instances of $O$ in $T$ have been fully evaluated, the search procedure will backtrack and explore other candidate occurrences, again using the FM-index.

This idea of hybrid in-index matching/in-text verification within the context of search schemes has been explored previously by Pockrandt for the Hamming distance metric. The author reports speed-ups between 1.6× and 2.1× and an optimal tipping point of 25 [9].

Performing in-text verification for the edit distance metric is more complex because pairwise alignment is computationally more expensive and thus needs to be highly optimized to have overall performance gains. Additionally, the precise start and end positions of each approximate occurrence of $P$ (which contains $O$) in $T$ are not known in advance. To this end, the bit-parallel alignment algorithm from section 3.3 is easily modified to support semi-global alignment.

## 3.5 Results and Discussion

All benchmarks were performed using a dataset of 100 000 Illumina NovaSeq 6000 reads (150 bp), randomly sampled from a larger whole genome sequencing dataset (accession no. SRR9091899). We identified all approximate read occurrences up to an edit distance of $k = \{1, 2, 3, 4\}$ on both strands of the human reference genome (GRCh38) [10]. We recall that we consider only lossless algorithms that are guaranteed to report all occurrences. We replaced non-ACGT characters in the reference genome (e.g., Ns) by a randomly chosen nucleotide. The different chromosomes were concatenated into a single string. As such, a read can be mapped across the borders of adjacent chromosomes. Such spurious matches can easily be filtered during post-processing.

All results were obtained using a single core of a 32-core Intel® Xeon® E5-2698 v3 CPU running at a base clock frequency of 2.30 GHz. To quantify variability in runtime, each benchmark run was repeated 20 times. We report both the average wall clock time as well as the standard deviation. Redundant occurrences (as defined in Chapter 2) were filtered.

### 3.5.1 Original vs. Adapted Search Schemes

***Table 3.2:*** *Comparison of the original search schemes by Kucherov et al. and our adapted search schemes, for different values of the maximum allowed edit distance k. In both cases, 100 000 Illumina reads of length 150 bp are mapped to both strands of the human reference genome.*

| Search scheme | Wall clock time ± SD | | No. of nodes visited (search space) | | No. of redundant occurrences | |
|---|---|---|---|---|---|---|
| | $k = 2$, unique occurrences = 676 528, reads mapped 90.5% | | | | | |
| Original | 15.91 ± 1.58 s | | 62 035 887 | | 267 541 | |
| Adapted | 14.73 ± 1.44 s | (−7.4%) | 57 263 477 | (−7.7%) | 264 671 | (−1.1%) |
| | $k = 3$, unique occurrences = 1 416 632, reads mapped 93.1% | | | | | |
| Original | 30.89 ± 1.80 s | | 128 708 469 | | 719 576 | |
| Adapted | 26.82 ± 0.60 s | (−13.2%) | 116 965 983 | (−9.1%) | 648 817 | (−9.8%) |
| | $k = 4$, unique occurrences = 2 579 745, reads mapped 94.8% | | | | | |
| Original | 72.07 ± 2.54 s | | 364 385 491 | | 1 492 806 | |
| Adapted | 61.35 ± 0.59 s | (−14.9%) | 305 476 323 | (−16.2%) | 1 420 668 | (−4.8%) |

In Table 3.2, the original and adapted search schemes (as defined in Table 3.1) are compared for edit distance values of $k = \{2, 3, 4\}$ as for $k = 1$, both search schemes are identical. We report the average runtime and standard deviation on a single CPU core and the number of nodes visited in the search trie. The latter equals the number of times a partial match is extended by a single character $c$ (in either direction). In practice, this involves expensive random memory access that largely determines the runtime. It is therefore a clear indication of intrinsic performance, regardless of the quality of implementation. It is clear that both in the size of search space (number

of nodes visited) and runtime the adapted search schemes are superior. This is no surprise, as the adapted schemes have tighter bounds and thus reduce the search space.

Table 3.2 also reports the total number of unique and redundant (filtered out) occurrences for the different values of $k$. Because search schemes are lossless, the number of unique occurrences does not differ between the original and adapted search scheme. Clearly, the tighter lower bounds also reduce the number of redundant occurrences (i.e., occurrences reported by multiple searches in the search scheme).

Finally, Table 3.2 reports the fraction of reads that have at least one occurrence in the reference genome ('reads mapped'), for the different values of $k$.

### 3.5.2 In-Index vs. In-Text Verification

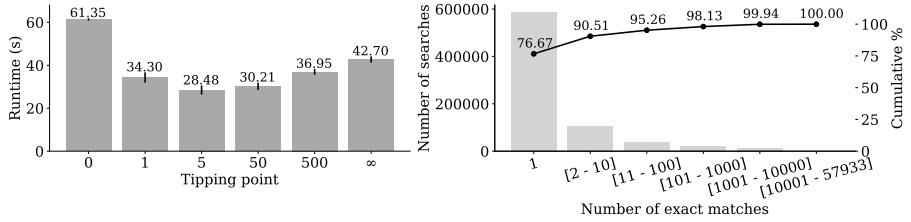


***Figure 3.2:*** *Left: the runtime for mapping 100 000 Illumina reads of length 150 bp to both strands of the human reference genome ($k = 4$) as a function of the tipping point $t$. Right: histogram of the number of matches for part $\mathcal{P}_{\pi[0]}$ across all searches.*

We compared the runtime for matching 100 000 Illumina patterns to both strands of the human reference genome with up to $k = 4$ edit operations for different values of the tipping point $t = 0$, 1, 5, 50, 500 and $\infty$. A value of $t = 0$ means that all patterns are entirely matched using the FM-index and that no in-text verification is performed whereas $t = \infty$ denotes that after the initial matching of the first part $\mathcal{P}_{\pi[0]}$, all candidate occurrences are verified directly in $T$ and that no further in-index extension takes place. For the intermediate tipping point values, the search procedure switches to in-text verification when $e - b \leq t$.

Figure 3.2 (left) shows the runtime as a function of tipping point $t$. Clearly, using purely in-index matching shows the worst performance for this particular dataset. This is because in-index matching involves expensive random memory access in the FM-index for each character that is added to a candidate occurrence. Switching to in-text verification when there is only a single candidate occurrence in $T$ ($t = 1$) reduces runtime by almost half. This is because bit-parallel, pairwise alignment between the appropriate substring of $T$ and $P$ can be performed very efficiently. This effect increases with larger tipping point values and for $t \approx 5$, runtime is minimized. For larger tipping point values ($t \geq 50$), the increasing overhead of suffix array lookup operations and pairwise alignments associated with in-text verification (that often turn out to be unsuccessful) dominates the gains. Remarkably, for this dataset, never performing in-index extension beyond the exact matching of the first part $\mathcal{P}_{\pi[0]}$ ($t = \infty$) is

still significantly faster than pure in-index matching ($t = 0$). For $t = \infty$, the matching process degenerates to a very simple procedure: exact pattern matching of part $\mathcal{P}_{\pi[0]}$ followed by in-text verification of each of the candidate occurrences. For our dataset, the largest suffix array range size encountered was 57 933. This range was encountered for a single read for which $\mathcal{P}_{\pi[0]}$ consists of 29 consecutive characters A.

Collectively over all reads, a tipping point $t$ between 2 and 10 yields the best performance. Within this range and for our dataset, the runtime is largely insensitive to the precise choice of $t$ (data not shown). Only for larger values of the tipping point ($t \geq 10$), we again observe an increase in runtime. For other values of $k$, a similar conclusion is reached: hybrid in-index matching/in-text verification reduces runtime by 38.43% for $k = 1$, 45.24% for $k = 2$ and 51.30% for $k = 3$.

#### 3.5.2.1 Breakdown of Reads

The search scheme for $k = 4$ errors consists of eight searches (see Table 3.1). Therefore, for the task of identifying all approximate occurrences of 100 000 reads on both strands of the reference genome, 1 600 000 searches are executed in total. For more than half of these searches (834 198), the first part $\mathcal{P}_{\pi[0]}$ has no exact match in $T$ and, hence, the search will immediately be terminated. This is no surprise, as most reads have approximate occurrences on only one strand of the reference genome. For the remaining 765 802 searches, Figure 3.2 (right) shows a breakdown as a function of the number of (exact) occurrences of part $\mathcal{P}_{\pi[0]}$. Remarkably, 76.67% (587 103) of those searches yield only a single occurrence in $T$ for $\mathcal{P}_{\pi[0]}$. In other words, for most reads, matching only a single part of $P$ already suffices to point to a unique position in $T$. For such cases, in-text verification of that sole candidate occurrence outperforms a further in-index character-by-character extension. This explains the large performance difference between tipping point values $t = 0$ and $t = 1$. Additionally, 13.84% (106 022) of the searches yield between 2 and 10 occurrences in $T$ for part $\mathcal{P}_{\pi[0]}$. Also for these cases, in-text verification at each of these candidate positions in $T$ is superior to in-index matching.

In contrast, only a relatively small fraction of 9.49% (72 677) of the searches deal with patterns for which $\mathcal{P}_{\pi[0]}$ has more than 10 occurrences in $T$. In certain cases, this number of instances is vast. For example, 14 329 searches yield more than 1 000 instances of $\mathcal{P}_{\pi[0]}$ in $T$, seven of which amount to more than 50 000 instances. The latter all correspond to low-complexity poly-A/T or poly-CA/GT patterns which are highly repeated in the human genome. Here, in-index matching has a clear advantage as all repeated candidate occurrences are handled simultaneously by the FM-index.

We conclude that in-text verification is beneficial for those searches for which the number of occurrences of $\mathcal{P}_{\pi[0]}$ in $T$ (and hence, the number of candidate occurrences of $P$ itself), is limited ($\leq 10$). For our dataset, this holds for roughly 90% of the searches. In contrast, the remaining searches (10%) deal with search patterns with many potential occurrences in $T$, a task which is best performed using in-index matching and the search scheme. We find that these 'difficult' searches, although limited in number, account for roughly two-thirds of the total runtime. In total, these complex searches account for 96.0% of unique matches over the entire dataset.

### 3.5.2.2 Suffix Array (SA) Space-Time Trade-off

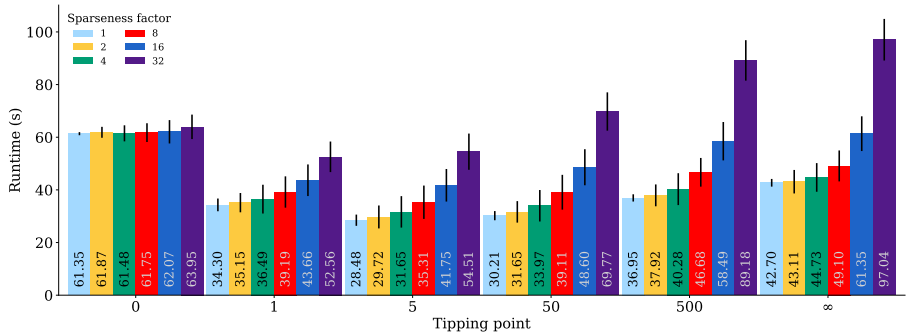


***Figure 3.3:*** *Runtime for mapping 100 000 Illumina reads (150 bp ) to both strands of the human reference genome as a function of the tipping point $t$ and sparseness factor $f$.*

In-text verification requires a lookup operation in the suffix array (SA) to retrieve, for each candidate occurrence, its position in $T$. The number of candidate occurrences for which in-text verification is performed, and hence, the number of required lookup operations in the SA, increases with higher values of the tipping point $t$.

To reduce the memory footprint of the FM-index, a sparse version of the SA is often used. In our implementation, every $f$-th suffix of the SA is stored, where $f$ denotes the sparseness factor, i.e., SA[i] is stored if and only if SA[$i$] $\mod f = 0$. It is well-known that a suffix at an arbitrary index $i$ can then be inferred in $O(f)$ time (see Section 1.3.4.1). Thus, the sparseness factor $f$ controls the space-time trade-off. As each in-text verification requires a lookup operation in the SA, a larger sparseness factor $f$ will diminish the gains in the runtime of in-text verification.

Figure 3.3 shows the runtime for different sparseness factors $f$ and tipping points $t$. The results for $f = 1$ (dense SA) are identical to those of Figure 3.2 (left). For all values of $t$, the runtime increases with the sparseness factor $f$, as lookup operations in the SA become more expensive. For $t = 0$, the increase in runtime from $f = 1$ to $f = 32$ is limited to only 4.2% whereas for $t = \infty$, the runtime more than doubles.

Therefore, especially for larger values of the sparseness factor $f$, the tipping point $t$ should not be set to (too) high values for good performance. In our experience, up to $f = 16$, a choice of $t \approx 5$ appears appropriate. For sparseness factors of $f = 32$ and larger, a tipping point of $t = 1$ or $t = 2$ showed the best performance.

## 3.5.3 Comparison To State-Of-The-Art Tools

In Chapter 2, we presented Columba 1.0, a fast software implementation for lossless approximate pattern matching using search schemes. Columba 1.0 implements the ideas outlined in that chapter such as a cache-friendly Burrows-Wheeler Transform (BWT) representation and dynamic partitioning of search schemes.

***Table 3.3:*** *Runtime comparison of state-of-the-art lossless alignment tools, with the exception of BWA in 'mem' mode, which is a lossy alignment algorithm. DNC stands for Did Not Complete within time limit (> 3h).*

| Tool | Language | Reference | $k = 1$ | $k = 2$ | $k = 3$ | $k = 4$ |
|---|---|---|---|---|---|---|
| Columba 1.1[1] | C++ | This chapter | 5.15 ± 0.44 s | 8.66 ± 1.00 s | 13.06 ± 1.31 s | 28.48 ± 2.13 s |
| Columba 1.0[2] | C++ | Chapter 2 | 7.05 ± 0.16 s | 13.10 ± 0.26 s | 25.62 ± 0.33 s | 67.75 ± 0.51 s |
| BWA[3] | C | [11] | 14.73 ± 0.23 s | 133.11 ± 2.39 s | 1454.40 ± 24.64 s | DNC |
| Bwolo | C++ | [3] | 12.53 ± 0.55 s | 25.24 ± 0.86 s | 63.67 ± 1.32 s | 189.78 ± 2.25 s |
| GEMv3[4] | C | [2] | 9.00 ± 1.50 s | 18.60 ± 2.40 s | 38.50 ± 4.60 s | 84.60 ± 4.90 s |
| Yara v0.9.11[5] | C++ | [12] | 4.49 ± 0.13 s | N/A | 21.00 ± 0.34 s | 81.90 ± 0.84 s |
| BWA-MEM (lossy) | C | [11] | 32.42 ± 0.67 s (independent of $k$) | | | |

The techniques described in the current chapter (bit-parallel edit distance computations, in-text verification, and the adapted search schemes) are implemented in Columba 1.1. In this section, we benchmark Columba 1.1 against state-of-the-art lossless pattern matching tools, including Columba 1.0. We use the adapted search schemes proposed in Table 3.1, a tipping point $t = 5$ and a SA sparseness factor $f = 1$ (dense SA).

In Table 3.3, we compare the performance of Columba 1.1 to Columba 1.0, Bwolo [3], GEM [2], Yara [12] and BWA [11] in all-mapping mode. Note that Columba 1.0 and Bwolo do not report the CIGAR string of the alignments in their output whereas the other tools do (including Columba 1.1). For the GEM aligner, not all occurrences could be reported as the tool failed when using the `all` parameter. Therefore, GEM was configured to report at most 1 000 occurrences per read. In Yara, the parameter for specifying the number of allowed errors is unique compared to the other aligners. Instead of directly setting a fixed number of allowed errors, Yara lets the user define this number as a percentage of the length of the read being analyzed. Only integer percentages from 0 to 10 are allowed. The underlying algorithm then multiplies this percentage with the read length and casts it down to an integer value. For this reason, with the reads of length 150 it is impossible to get results for 2 allowed errors.

Columba 1.1 outperforms Columba 1.0 for all values of $k$, even though Columba 1.0 does not compute the Compact Idiosyncratic Gapped Alignment Report (CIGAR) string. Gains are achieved through the tighter lower bounds as specified in the adapted search schemes and bit-parallel, in-text verification. Clearly, these gains outweigh the extra computations required to generate the CIGAR string.

Columba 1.1 and 1.0 outperform all other lossless alignment tools when $k \geq 3$. However, for $k = 1$, both versions are slightly slower than Yara, likely due to the overhead introduced by the use of the bidirectional FM-index, as opposed to Yara's reliance on a unidirectional index. At $k = 2$, Columba 1.1 stands out as the fastest tool, surpassing all competitors, whereas its predecessor lags behind Yara. For $k \geq 2$, Columba 1.1 demonstrates remarkable performance, being at least twice as fast as

[1] -e $k$ -i 5 -ss ../search_schemes/kuch_k+1_adapted/

[2] -e $k$ -ss ../search_schemes/kuch_k+1/

[3] aln -N -n $k$ -i 0 -l 150 -k $k$

[4] -t 1 -e [$k$] -s [$k$] –alignment-model edit –mapping-mode complete -M 1000

[5] -e [$k$] -s [$k$] -y full -t 1

other tools. Notably, at $k = 4$, it achieves a speed approximately three times greater than GEM and Yara, and six times faster than Bwolo. It is evident that BWA was not designed for lossless mode operation at higher values of $k$.

We also compare Columba 1.1 with BWA in (lossy) mem mapping mode. In mem mode, BWA does not require a maximum number of errors $k$ to be specified and it will typically report only a single candidate alignment position for each read. Note that the time to read the index structure from disk is included in BWA's runtime, which is not the case for Columba 1.1. Also note that BWA outputs Sequence Alignment/Map (SAM) format and is able to handle paired-end reads, which is not the case for Columba 1.1. Columba 1.1 appears faster than BWA for $k = 1, 2$ and 3. For $k = 4$, the runtime of Columba 1.1 is similar to that of BWA This indicates that the performance gap between lossless and lossy alignment tools is closing for practical bioinformatics applications such as read mapping.

## 3.6 Conclusion

We introduced Columba 1.1, a tool for lossless approximate pattern matching using search schemes under the edit distance metric. Columba 1.1 implements hybrid in-index matching/in-text verification using a bit-parallel, pairwise alignment algorithm. It is demonstrated that this technique reduces runtime by more than a factor of two, compared to pure in-index matching. We provided an analysis of the effect of in-text verification for different types of reads. For reads with a limited number of occurrences, switching to in-text verification greatly reduces the runtime. In contrast, for reads with many potential occurrences, in-index matching appears the better option. We showed that the use of a sparse suffix array somewhat diminishes the performance gains of using in-text verification. Nevertheless, for all practical values of the suffix array sparseness factor, in-text verification proves beneficial. Finally, Columba 1.1 shows superior performance to state-of-the-art lossless aligners.

**Acknowledgments.**

Luca Renders and Lore Depuydt are funded by the Research Foundation – Flanders (FWO), through a PhD Fellowship SB (1SE7822N) and a PhD Fellowship FR (1117322N), respectively.

# References

[1] H. Hyyrö and G. Navarro. *Faster Bit-Parallel Approximate String Matching*. In A. Apostolico and M. Takeda, editors, Combinatorial Pattern Matching, pages 203–224, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45452-7_18.

[2] S. Marco-Sola, M. Sammeth, G. R, and P. Ribeca. *The GEM mapper: fast, accurate and versatile alignment by filtration*. Nature Methods, 9(December):1185–1188, 2012. doi:10.1028/nmeth.2221.

[3] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[4] T. K. Vintsyuk. *Speech discrimination by dynamic programming*. Cybernetics, 4(1):52–57, 1968. doi:10.1007/bf01074755.

[5] G. Navarro. *A Guided Tour to Approximate String Matching*. ACM Comput. Surv., 33(1):31–88, March 2001. doi:10.1145/375360.375365.

[6] G. Myers. *A fast bit-vector algorithm for approximate string matching based on dynamic programming*. J. ACM, 46(3):395–415, May 1999. doi:10.1145/316542.316550.

[7] H. Hyyrö. *A bit-vector algorithm for computing Levenshtein and Damerau edit distances*. Nordic J. of Computing, 10(1):29–39, March 2003. doi:10.5555/846090.846095.

[8] W. J. Masek and M. S. Paterson. *A faster algorithm computing string edit distances*. Journal of Computer and System Sciences, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

[9] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

[10] V. Schneider, T. Graves-Lindsay, K. Howe, N. Bouk, H.-C. Chen, P. Kitts, T. Murphy, K. Pruitt, F. Thibaud-Nissen, D. Albracht, R. Fulton, M. Kremitzki, V. Magrini, C. Markovic, S. McGrath, K. Steinberg, K. Auger, W. Chow, J. Collins, and D. Church. *Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly*. Genome Research, 27, 2017. doi:10.1101/gr.213611.116.

[11] H. Li and R. Durbin. *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

[12] E. Siragusa. *Approximate string matching for high-throughput sequencing*. PhD thesis, Freie Universität Berlin, 2015. doi:10.17169/refubium-15562.

# 4

# Automated Design of Efficient Search Schemes

*In this chapter, we introduce new search schemes inspired by the manual adaptations discussed in the previous chapter and the insights on efficient search schemes from Chapter 2. We present and implement two novel approaches for designing these search schemes, which are incorporated into a new tool, Hato. We then compare the search spaces generated by these new schemes with those of established schemes in the literature. Additionally, we introduce a dynamic selection technique, which resembles dynamic partitioning from Chapter 2 in some respects but serves as a complementary approach. This technique further reduces the search space, leading to the development of Columba 1.2. We assess Columba 1.2 for both runtime and memory efficiency against a variety of other alignment tools. This chapter is based on material first presented at RECOMB 2024 in Cambridge, Massachusetts, and subsequently selected and expanded for publication in the Journal of Computational Biology.*

**L. Renders, L. Depuydt, S. Rahmann, and J. Fostier.**

**Published in *Journal of Computational Biology*, Volume 31, Issue 10, October 2024, pg. 975–989, https://doi.org/10.1089/cmb.2024.06**

**Abstract** This study introduces a pioneering approach to automate the creation of search schemes for lossless approximate pattern matching. Search schemes are combinatorial structures, that define a series of searches over a partitioned pattern. Each search specifies the processing order of these parts and the cumulative lower and upper

bounds on the number of errors in each part of the pattern. Together, these searches ensure the identification of all approximate occurrences of a search pattern within a predefined limit of $k$ errors.

While existing literature offers designed schemes for up to $k = 4$ errors, designing search schemes for larger $k$ values incurs escalating computational costs.

Our method integrates a greedy algorithm and a novel Integer Linear Programming (ILP) formulation to design efficient search schemes for up to $k = 7$ errors. Comparative analyses demonstrate the superiority of our ILP-optimal schemes over alternative strategies in both theoretical and practical contexts. Additionally, we propose a dynamic scheme selection technique tailored to specific search patterns, further enhancing efficiency. Combined, this yields runtime reductions of up to 53% for higher $k$ values. Furthermore, we adapt existing search schemes for $k > 7$ (like the ones based on the pigeonhole principle), using the greedy algorithm, presenting more efficient, but not optimal, search schemes for $k \leq 13$.

To facilitate search scheme generation, we present Hato, an open-source software tool (AGPL-3.0 license) employing the greedy algorithm and utilizing CPLEX for ILP solving. Furthermore, we introduce Columba 1.2, an open-source lossless read-mapper (AGPL-3.0 license) implemented in C++. Columba surpasses existing state-of-the-art tools by identifying all approximate occurrences of 100 000 Illumina reads (150 base-pair (bp)) in the human reference genome within 24 seconds (maximum edit distance of 4) and 75 seconds (maximum edit distance of 6) using a single CPU core. Notably, our study showcases Columba's capability to align 100 000 reads of length 50, with high error rates and up to an edit distance of 7, in a mere 2 hours and 15 minutes. This achievement is unmatched by other lossless aligners, which require over 3 hours for edit distance 5 alignments. Moreover, Columba exhibits a mapping rate four times higher than that of a lossy tool for this dataset.

## 4.1 Introduction

Existing search schemes for lossless approximate pattern matching (APM), as previously discussed in Section 1.4, have been developed to improve computational efficiency by systematically reducing the search space. These schemes often leverage bidirectional indexes and divide the pattern $P$ into $k + 1$ or $k + 2$ non-overlapping parts, where $k$ is the maximum number of allowed errors. A prominent example is the pigeonhole-based approach, which ensures that at least one part is error-free, allowing for an exact search on these parts before extending matches with branching and backtracking. Kucherov et al. formalized the concept of search schemes, introducing methods that efficiently manage the matching process using bidirectional indexes and provided designed schemes for up to four errors [1]. Building on this, Kianfar et al. employed a Mixed Integer Linear Programming (MILP) formulation to optimize search schemes specifically for the Hamming distance, also covering cases with up to $k = 4$ [2]. Other generalizable approaches include Pockrandt's schemes, which extended the pigeonhole principle to any $k$, and the 01*0 seed-based schemes designed for arbitrary $k$ [3]. However, as seen in Chapter 2, both these general approaches are

not computationally optimal. Hence, the design of efficient search schemes for $k > 4$ remains an open problem, paving the way for further research to address the growing demand for robust and scalable lossless APM solutions.

In Chapters 2 and 3, we introduced Columba, an efficient implementation of a lossless read-mapper that supports arbitrary search schemes for both the edit and Hamming distance. In present work, we make following new contributions:

1. We propose a greedy algorithm to improve the computational performance of existing search schemes. Even though this algorithm cannot guarantee optimality, it is able to quickly generate lossless search schemes for up to $k = 13$ errors with good computational performance. We show that for $k = 5, 6$ and 7 errors, the search space (and thus: runtime) is reduced by respectively 55%, 63% and 71%, compared to the best previously known search schemes (i.e., either pigeonhole principle based or 01*0 seed based)

2. We present a novel ILP formulation to generate search schemes. In contrast to the greedy algorithm, these search schemes are guaranteed to be optimal under the set of used constraints. Using the ILP algorithm, we generated for the first time efficient search schemes for $k = 5, 6$ and 7 errors that demonstrate superior performance in a practical scenario. For $k = 7$ errors, the search space is further reduced with up to 16% as compared to the search scheme generated using our greedy algorithm.

3. For a given value of $k$, multiple, co-optimal ILP solutions can be generated. Even though co-optimal search schemes are equally fast *on average*, their individual performance may vary among different search patterns $P$. We propose a *dynamic selection* method where, for each pattern $P$, the expected best performing search scheme is chosen. We show that this heuristic can reduce the search space by up to 11% as compared to the static use of a search scheme.

4. We present Hato (Japanese for "pigeon") and Columba (Latin for "pigeon") 1.2, two open-source C++11 tools [1] under AGPL-3.0 license. Hato implements the greedy algorithm and the ILP solver using CPLEX 22.1.1 [4]. Columba 1.2 is a lossless read-mapper that implements dynamic selection, on top of other improvements described in Chapters 2 and 3. We demonstrate that Columba, using dynamic selection and the novel search schemes generated by Hato, significantly outperforms other state-of-the-art lossless alignment implementations, such as GEM [5], Yara [6], Burrows-Wheeler Aligner (BWA) aln (in lossless mode) [7] and Bwolo [8], in the task of identifying all occurrences of 150 bp Illumina reads and in identifying all occurrences of 50 bp subreads of Pacific Biosciences (PacBio) reads in the human reference genome. Additionally, we demonstrate superior performance to BWA in lossy mem mode for $k \leq 3$ and a similar runtime for $k = 4$ with Illumina reads.

Section 4.2 briefly recaps the concept of search schemes and provides an analysis on how to compare search schemes. In Section 4.3, we introduce a greedy heuristic

[1] available at https://github.com/biointec/hato and https://github.com/biointec/columba

that, given an initial search scheme, improves the computational performance by adapting the original search scheme into a more efficient one. In Section 4.4, we present an ILP formulation that incorporates a new objective function to create efficient search schemes in practice. Section 4.5 discusses the dynamic selection method, which identifies the expected best performing search scheme for a given pattern $P$ from a set of co-optimal schemes. The practical performance of the newly designed search schemes and the dynamic selection method is evaluated and compared to the current state of the art in Section 4.6. Finally, Section 4.7 provides a summary of the overall findings.

## 4.2 Search Schemes

In this section, we briefly recap the key concepts of search schemes, which were introduced in Section 1.4 of the introductory chapter. We assume that up to $k$ errors are allowed over pattern $P$ and that $P$ is divided into a fixed number of parts $p$. It is useful, but not strictly required, to choose $p \geq k+1$ to have at least one error-free part. We use square brackets for indexing arrays, and indexing starts at 0, i.e. $s[0]$ is the first character of array $s$. Sequences of single-digit integers are written without interpunctuation, e.g. $s = (0, 0, 1)$ is written as $s = 001$. To recap: a search scheme $\mathbb{S}$ is a collection of searches. Each search is defined by three arrays of length $p$, the $\pi$, $L$ and $U$-arrays, that respectively define the order in which the parts are processed and the cumulative lower and upper bounds to the number of allowed errors. An error distributions is an integer array that distributes up to $k$ errors over the parts of the pattern.

**Example 4.1.** *The most intuitive search schemes are based on the* ***pigeonhole principle****. Given k, these schemes consist of $p = k + 1$ parts and $k + 1$ searches. Since there are only k errors to distribute, each possible error distribution must leave at least one of the parts error-free. Hence each search starts by matching a different part exactly. The other parts can then be processed allowing at most k errors. All searches have lower and upper bounds $L = 00 \dots 0$ and $U = 0k \dots k$. For $k = 2$, this results in $\mathbb{S} = \{S_0, S_1, S_2\}$ with $S_0 = (012, 000, 022)$, $S_1 = (102, 000, 022)$, $S_2 = (210, 000, 022)$.*

**Example 4.2.** *For the same $k = 2$, $p = 3$, Kucherov et al. [1] proposed the following scheme: $S_0 = (012, 000, 022)$; $S_1 = (102, 001, 012)$; $S_2 = (210, 000, 012)$. While $S_0$ is identical to the previous example, the lower and upper bounds are tighter for the subsequent searches, making them more efficient than the search scheme based on the pigeonhole principle. By checking all error distributions, it is shown that this search scheme is still valid.*

These definitions were first formalized by [1]. Search schemes are designed to systematically manage the number of allowed errors during the search process. The primary idea is to incrementally increase the number of permissible errors, a notion reflected in the $U$-sequence. By doing so, search schemes enable the efficient elimination of unsuccessful branches within the search trie. Additionally, the $L$-sequences

serve a crucial role in preventing redundant coverage of error distributions. A well-constructed search scheme aims to include just one covering search for each error distribution, ensuring efficiency and effectiveness.

### 4.2.1 Comparing Search Schemes

Kucherov et al. [1] proposed a method to evaluate the efficiency of a search scheme. Their approach considers the Hamming or Levenshtein distance for random texts and independent reads. The efficiency is estimated by enumerating the number of strings generated by all searches within the scheme. Each enumerated substring corresponds to a character extension (either forward or backward) in the bidirectional index. This can be visualized as a search trie, where each extension is a node in the trie. As such, this number reflects the search space. A smaller search space indicates a more efficient search scheme. Each extension in the index is executed in constant time. Hence, the number of enumerated substrings correlates with the runtime and thus the efficiency.

Kianfar et al. [2] designed search schemes based on the formula proposed by Kucherov et al. for the Hamming distance. However, in Chapter 2 we showed that these search schemes do not perform effectively in genomic contexts. As noted by the authors, this discrepancy can be attributed to the false assumption of randomness and independence of the reads within genomic contexts.

To illustrate this discrepancy empirically, we conducted a comparison of the number of visited nodes in the search trie in various scenarios. We utilized two reference genomes, both of identical size. The first reference genome consists of 10 million random base pairs, while the second comprises the initial 10 million base pairs of chromosome 21 from the human reference genome. Additionally, we generated three sets of reads, each containing 100 000 reads with a length of 50. The first read set comprises random reads. The second and third read sets consist of reads sampled (with errors) from the random and genomic reference texts, respectively. We aligned the first and second read sets to the random reference text and the first and third read sets to the genomic reference text. This alignment was performed using the search scheme based on the pigeonhole principle for two errors, parts of equal size, and the Hamming distance. The number of visited nodes for each of these contexts is presented in Table 4.1

***Table 4.1:*** *The average number of nodes visited per read using the search scheme based on the pigeonhole principle for 2 errors,* 100 000 *reads of length 50 and a reference text of 10 million base pairs for different scenarios.*

| | random text | genomic text |
|---|---|---|
| random reads | 35 | 34 |
| sampled reads | 74 | 89 |

The expected number of nodes using Kucherov et al.'s formula for this search scheme, with the given read and reference lengths, was 35. The table clearly demonstrates that the formula holds true when reads are independent of the reference text. However, when reads are not independent from the reference text (i.e. sampled), the

formula no longer holds. This discrepancy is particularly pronounced in genomic contexts, where genomic texts exhibit repetition, increasing the likelihood that reads originated from repetitive regions. The discrepancy between the expected number of nodes and the empirically observed number of nodes increases with $k$.

In our work, we prioritize empirical comparisons over relying solely on theoretical formulas. While theoretical frameworks such as Kucherov et al.'s formula provide valuable insights, our emphasis lies on empirical evidence derived from real-world data and scenarios. By conducting empirical comparisons, we can directly observe and analyze the performance of search schemes in genomic contexts. This approach allows us to account for factors such as the non-random nature of reads and the repetitive nature of genomic texts, which may significantly impact the effectiveness of search schemes. Therefore, in our analyses and comparisons, we will opt for empirical metrics that accurately reflect the performance of search schemes in genomic analysis.

#### 4.2.1.1 Designing Search Schemes

In Chapters 2 and 3 we found that fast growth rates of the $U$-sequence lead to larger search spaces, resulting in increased computational requirements. On the other hand, fast-growing $L$-sequences reduce the search space. Hence, our goal is to minimize the $U$-sequences and maximize the $L$-sequences. We call the resulting schemes **minU search schemes**. First, we introduce a greedy heuristic to design search schemes with these characteristics in mind. Later, we propose an ILP formulation that generates optimal search schemes under the set of constraints.

## 4.3 A Greedy Heuristic for Improving Search Schemes

We propose a greedy heuristic for designing search schemes. This algorithm takes as input any valid search scheme, whose $U$- and $L$-values are then decreased respectively increased by greedy local changes. We may start with the search scheme based on the pigeonhole principle or from the 01*0 seeds strategy [8]. The $\pi$-sequences of the searches are not modified by this heuristic.

The algorithm tracks which searches cover which error distributions. If a distribution is covered by only one search, it is labeled as *critical* for that search. First, we aim to minimize the $U$-sequences by iterating through positions $i = 0, \dots, p-1$. Before each iteration, searches are sorted lexicographically by decreasing $U$-sequences. In case of a tie, they are sorted by increasing $L$-sequences. For each search $S_x$ in this list, we assess the possibility of reducing $U_x[i]$ without violating the monotonicity of the bounds or losing coverage of critical error distributions. If $U_x[i]$ is decreased, we update the list of covering searches for each error distribution previously covered by $S_x$. In the event that an error distribution $e$ is now exclusively covered by another search $S_y$, it will be added to $S_y$'s list of critical error distributions.

Following this, we aim to maximize the $L$-sequences. This involves iterating over decreasing positions $i = p-1, \dots, 1, 0$ (to ensure monotonicity of $L$) and determining how much $L_x[i]$ can increase without violating any properties.

**Algorithm 9:** Algorithm to Greedily Adapt a Search Scheme

**Input:** A valid search scheme $\mathbb{S}$ for $p$ parts and up to $k$ errors.
**Output:** The adapted search scheme.

```
// Generate all error distributions E
foreach error distribution e ∈ E do
    e.covering ← []
foreach search S_x ∈ 𝕊 do
    x.covering ← [], x.critical ← []
foreach error distribution e ∈ E do
    foreach search S_x ∈ 𝕊 do
        if S_x covers e then
            e.covering.add(x), x.covering.add(e)
    if |e.covering| = 1 then
        e[0].critical.add (e)
for i ← 0 to p − 1 do
    Sort the searches in 𝕊 by decreasing U-array and increasing L-array
    foreach search S_x ∈ 𝕊 do
        while true do
            U_x[i] ← U_x[i] − 1
            if U_x[i] < 0 or invalid bounds or not (all e ∈ x.critical are covered by
              S_x) then
                U_x[i] ← U_x[i] + 1
                break
        foreach error distribution e ∈ x.covering do
            if S_x does not cover e then
                Remove x from e.covering, Remove e from x.covering
                if |e.covering| = 1 then
                    Add e to e[0].critical
for i ← 0 to p − 1 do
    Sort the searches in 𝕊 by decreasing U-array and increasing L-array
    foreach search S_x ∈ 𝕊 do
        while true do
            L_x[i] ← L_x[i] + 1
            if L_x[i] > k or invalid bounds or not (all e ∈ x.critical are covered by
              S_x) then
                L_x[i] ← L_x[i] − 1
                break
        foreach error distribution e ∈ x.covering do
            if S_x does not cover e then
                Remove x from e.covering, Remove e from x.covering
                if |e.covering| = 1 then
                    Add e to e[0].critical
```

We have successfully generated search schemes for up to $k = 13$ using this technique. The pseudo-code for the greedy approach is detailed in Algorithm 9. The resulting greedily adapted search schemes can be found in supplementary Tables A.1-A.4 and Table A.6. The computational complexity of this algorithm is primarily driven by the number of error distributions, which is given by $\binom{p+k}{k}$.

## 4.4 Integer Linear Program Formulation

Kianfar et al. [2] introduced a Mixed Integer Linear Programming (MILP) model to find optimal search schemes in a specific sense: Their model minimized the (recursively defined) expected number of enumerated substrings for a given pattern length. This expected number of explored branches is computed based on the assumption that both $T$ and $P$ are random sequences. However, in the context of sequence alignment, both $P$ and $T$ are not random; and moreover, we expect $P$ to have an approximate occurrence in $T$, thus the assumption of randomness typically does not hold. As estimating the number of expected branches is computationally very expensive, Kianfar et al. only generated search schemes for up to $k = 4$ and $S = |\mathbb{S}| = 3$. For $k = 3$ and $k = 4$, this resulted in search schemes for which at least one of the searches did not start with an exact match. We showed in Chapter 2 that such search schemes result in non-competitive runtimes on repetitive and complex genomes such as the human genome.

Therefore, we propose a novel Integer Linear Programming (ILP) formulation to design practical and effective search schemes with a different objective. As stated earlier, our goal is to minimize the $U$-sequences and maximize the $L$-sequences. As an additional objective, we want to minimize the number of error distributions that are redundantly covered. Our Integer Linear Programming formulation (Table 4.2) aims to identify a search scheme that efficiently identifies all approximate matches with up to $k$ errors, considering $S$ searches and $p$ parts. The remainder of this section provides a technical explanation of the ILP model. For brevity, we abbreviate the set $\{0, 1, \ldots, n-1\}$ by $[n]$.

### 4.4.1 ILP details

The total number of possible error distributions is denoted as $Q = \binom{p+k}{k}$ (Lemma 1.3). For $q \in [Q]$, $e_q$ denotes the $q$-th error distribution (in some fixed enumeration), and $E_{q,j}$ denotes the cumulative number of errors encountered in $e_q$ up to and including index $j$. These are precomputed constants. For the *variables*, we partially adopt the modeling by Kianfar et al. for $U$, $L$ and $\lambda$, where $U_{s,i}$ and $L_{s,i}$ respectively denote the upper and lower bound of $\mathcal{S}_s$ for the $i$-th part (in $\pi_s$-order), and $\lambda_{q,s}$ is a boolean variable that indicates whether $e_q$ is covered by $\mathcal{S}_s$.

To model the permutations $\pi_s$ with the additional connectivity restriction, we use a novel direct encoding and introduce $p-1$ binary variables $y_{s,i}$ per search $\mathcal{S}_s$. The total number of zeros in the sequence $y_s$ corresponds to $\pi_s[0]$. A 1-bit in the sequence corresponds to an extension to the right, whereas a 0-bit in the sequence is an extension to the left. For example, for $p = 5$, the sequence $y = 0110$ corresponds to $\pi = 21340$: We have $\pi[0] = 2$, as there are two zeros in $y$; then the interval $[2, 2]$ is extended to the left, right, right, and left, which gives $1, 3, 4, 0$. In this way, connected permutations on $[p]$ are in a 1-to-1 correspondence with bit-vectors of length $p-1$.

Our new *objective function* consists of three components. The first component aims to minimize the upper bounds across all searches, with higher importance given to earlier upper bounds within a search. The weighting factor $(k+1)^{(p-i-1)}$ ensures

***Table 4.2:*** *ILP formulation to design search schemes with minimized $U$-sequences, maximized $L$-sequences and a small number of redundantly covered error distributions.*

**Variables:**

| | | |
|---|---|---|
| $U$- and $L$-sequences | $U_{s,i}, L_{s,i} \in [k+1]$ | $s \in [S], i \in [p]$ |
| encoded $\pi$-sequences | $y_{s,i} \in \{0,1\}$ | $s \in [S], i \in [p-1]$ |
| covering indicators | $\lambda_{q,s} \in \{0,1\}$ | $q \in [Q], s \in [S]$ |
| cumulative number of errors in $e_q$ after stage $i$ of $S_s$ | $\epsilon_{q,s,i} \in [k+1]$ | $q \in [Q], s \in [S], i \in [p]$ |
| indicators of $L_{s,i} \leq \epsilon_{q,s,i}$ and $\epsilon_{q,s,i} \leq U_{s,i}$ | $\mu^L_{q,s,i},\ \mu^U_{q,s,i} \in \{0,1\}$ | $q \in [Q], s \in [S], i \in [p]$ |
| indicator for $(\max_{i' \leq i} \pi_s[i']) = j$ | $r'_{s,i,j} \in \{0,1\}$ | $s \in [S], i \in [p], j \in [p]$ |

**Objective:**

$$\text{Minimize} \quad \sum_{s=0}^{S-1} \sum_{i=0}^{p-1} (k+1)^{(p-i-1)} \cdot U_{s,i} - \sum_{s=0}^{S-1} \sum_{i=0}^{p-1} (p-i) \cdot L_{s,i} + \sum_{q=0}^{Q-1} \sum_{s=0}^{S-1} \lambda_{q,s},$$

**subject to**

$$L_{s,i} \leq U_{s,i} \qquad s \in [S], i \in [p] \qquad (4.1)$$

$$L_{s,i} \leq L_{s,i+1} \qquad s \in [S], i \in [p-1] \qquad (4.2)$$

$$U_{s,i} \leq U_{s,i+1} \qquad s \in [S], i \in [p-1] \qquad (4.3)$$

$$\sum_{s=0}^{S-1} \lambda_{q,s} \geq 1 \qquad q \in [Q] \qquad (4.4)$$

$$\lambda_{q,s} \leq \mu^L_{q,s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.5)$$

$$\lambda_{q,s} \leq \mu^U_{q,s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.6)$$

$$\lambda_{q,s} \geq \sum_{i=0}^{p-1} (\mu^L_{q,s,i} + \mu^U_{q,s,i}) - 2p + 1 \qquad q \in [Q], s \in [S] \qquad (4.7)$$

$$(k+1) \cdot \mu^L_{q,s,i} \geq 1 + \epsilon_{q,s,i} - L_{s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.8)$$

$$k \cdot \mu^L_{q,s,i} \leq k + \epsilon_{q,s,i} - L_{s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.9)$$

$$(k+1) \cdot \mu^U_{q,s,i} \geq 1 - \epsilon_{q,s,i} + U_{s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.10)$$

$$k \cdot \mu^U_{q,s,i} \geq k - \epsilon_{q,s,i} + U_{s,i} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.11)$$

$$(p-1) - \sum_{h=0}^{p-2} y_{s,h} = \sum_{j=0}^{p-1} j \cdot r'_{s,i,j} \qquad s \in [S], i \in [p] \qquad (4.12)$$

$$\sum_{j=0}^{p-1} r'_{s,i,j} = 1 \qquad s \in [S], i \in [p] \qquad (4.13)$$

$$r'_{s,i,j} = 0 \qquad s \in [S], i \in [p], j \in [i] \qquad (4.14)$$

$$\epsilon_{q,s,i} = \sum_{j=i}^{p-1} r'_{s,i,j} \cdot E_{q,j} - \sum_{j=0}^{p-i-2} r'_{s,i,j+i+1} \cdot E_{q,j} \qquad q \in [Q], s \in [S], i \in [p] \qquad (4.15)$$

$$\sum_{i=0}^{p-2} y_{s+1,i} \cdot 2^i \geq 1 + \sum_{i=0}^{p-2} y_{s,i} \cdot 2^i \qquad s \in [S-1] \qquad (4.16)$$

that lexicographically smaller $U$-sequences are preferred over lexicographically larger ones, and give a high overall weight to the $U$-sequences. The second component maximizes the lower bounds. Here, a weighting factor of $(p-i)$ is used to give (slightly) higher importance to bounds earlier in the search. The third component aims to minimize the coverage of error distributions by multiple searches. By minimizing the coverage, the objective function encourages the utilization of fewer searches to handle each error distribution, thereby enhancing efficiency and reducing redundancy.

The *constraints* (and auxiliary variables to express the constraints) together ensure that the search scheme is valid. Constraints (4.1), (4.2) and (4.3) state the obvious properties of the $L$- and $U$-sequences.

The next constraints (4.4)–(4.7) ensure that the search scheme is valid, i.e., that each error distribution is covered by at least one search. Here, (4.4) directly models this coverage constraint using the $\lambda$-variables, and the other constraints express that $\lambda_{q,s} = 1$ if and only if $\mu^L_{q,s,i} = \mu^U_{q,s,i} = 1$ for all $i \in [p]$, which we then constrain to be indicators of $L_{s,i} \leq \sum_{j\leq i} e_q[\pi_s[j]]$ and $\sum_{j\leq i} e_q[\pi_s[j]] \leq U_{s,i}$, respectively.

Indeed, constraints (4.8)–(4.11) do exactly this using auxiliary variables $\epsilon_{q,s,i} := \sum_{j\leq i} e_q[\pi_s[j]]$ (which remain to be constrained to satisfy this definition). In particular, constraints (4.8) and (4.9) ensure $\mu^L_{q,s,i} = 1$ if and only if $L_{s,i} \leq \epsilon_{q,s,i}$, whereas constraints (4.10) and (4.11) ensure $\mu^U_{q,s,i} = 1$ if and only if $\epsilon_{q,s,i} \leq U_{s,i}$.

Now, for the hardest part constraints (4.12)–(4.15): As mentioned above, we need variables $\epsilon_{q,s,i} := \sum_{j\leq i} e_q[\pi_s[j]]$ that contain the cumulative number of errors of the $q$-th error distribution in the $i$-th part of search $s$ in $\pi_s$-order, but $\pi_s$ is encoded in the $y_s$-variables. Fortunately, we can express the right border (call it $r_{s,i}$) of the interval $\pi_s[0,\dots,i]$ using the $y$-variables. As $\pi_s[0]$ corresponds to the number of zeros in $y_s$, and each 1-bit extends the interval to the right, we have $r_{s,0} = (p-1) - \sum_{h=0}^{p-2} y_{s,h}$ and $r_{s,i+1} = r_{s,i} + y_{s,i}$ for $i \in [p-1]$. Combining these two gives $r_{s,i} = (p-1) - \sum_{h=i}^{p-2} y_{s,h}$ for $i \in [p]$ (with the empty sum taking a value of 0 as usual). For the left interval borders we have $\ell_{s,i} = r_{s,i} - i$. We can now write $\epsilon_{q,s,i}$ as the difference between the number of errors encountered up until the right border minus the number of errors encountered before the left border: $\epsilon_{q,s,i} = E_{q,r_{s,i}} - E_{q,(\ell_{s,i}-1)}$, using the precomputed constants $E_{q,j}$. To select the correct $E_{q,j}$ values to subtract from each other using linear relations, we convert the interval endpoint $r_{s,i}$ into a sequence of binary indicator variables $r'_{s,i,j}$ with $r'_{s,i,j} = 1$ if and only if $r_{s,i} = j$. This relation is modeled by (4.12) and (4.13). As we must have $r_{s,i} \geq i$, we have $r'_{s,i,j} = 0$ for $j < i$, which is (4.14). Similarly, the left boundary $\ell_{s,i}$ can be converted into a binary sequence $\ell'_{s,i,j}$. However, it is not necessary to do this explicitly because $\ell_{s,i} = r_{s,i} - i$ and hence $\ell'_{s,i,j} = r'_{s,i,j+i}$ for $j \in [p-i]$. Together, we obtain constraint (4.15): $\epsilon_{q,s,i} = E_{q,r_{s,i}} - E_{q,l_{s,i}-1} = \sum_{j=0}^{p-1} (r'_{s,i,j} - \ell'_{s,i-1,j})\, E_{q,j} = \sum_{j=i}^{p-1} r'_{s,i,j}\, E_{q,j} - \sum_{j=0}^{p-i-2} r'_{s,i,j+i+1}\, E_{q,j}$.

Finally, constraint (4.16) is a symmetry-breaking constraint enforcing a monotone order of the permutations $\pi_s$ across the $S$ searches by enforcing that the $y_s$-sequences (interpreted as binary numbers) are strictly increasing. (Recall that a search scheme is a set of searches in which the order of searches is unimportant; so we prescribe a canonical ordering to avoid symmetric solutions.)

#### 4.4.1.1 Limiting the scope of the ILP-model

The ILP in Table 4.2 is general and works in principle for any parameter combination of $k$, $p$ and $S$. In practice, we are mostly interested in the special case with $p = k + 1$ parts (one guaranteed error-free part) and $S = p$ searches, each of which starts with a distinct error-free part. Under these assumptions, we can introduce extra constraints that reduce the time needed to solve the ILP to optimality. We remove the symmetry-breaking constraints (4.16) and add the following explicit ones:

$$U_{s,0} = 0, \quad s \in [S], \tag{4.17}$$

$$U_{s,k} = k, \quad s \in [S], \tag{4.18}$$

$$(p-1) - \sum_{i=0}^{p-2} y_{s,i} = s, \quad s \in [S], \tag{4.19}$$

$$y_{0,i} = 1, \quad i \in [p-1], \tag{4.20}$$

$$y_{S-1,i} = 0, \quad i \in [p-1]. \tag{4.21}$$

Constraint (4.17) ensures that each search starts with an exact match. Constraint (4.18) ensures that each search allows up to $k$ errors in the last part; this is necessary for $S = p = k + 1$ to cover all error distributions with $k$ 1s and a single 0. Constraint (4.19) enforces that each search starts with a different part by setting $\pi_s[0] = s$. Constraints (4.20) and (4.21) then respectively model the explicit $\pi$-sequence of the first search (all extensions to the right) and of the last search (all extensions to the left), as these are the only possibilities when we start on the left respectively right end.

Kucherov et al. defined the *critical sequence* of a search scheme $\mathbb{S}$ as its lexicographically maximal $U$-sequence. They showed that the minimal critical sequence in a valid search scheme for $p = k + 1$ is $013355 \ldots kk$ for odd $k$ and $02244 \ldots kk$ for even $k$ [1], and in an optimal search scheme, some search, called the *critical search*, has this $U$-sequence. Thus, we add the following constraints if $k$ is odd, as the critical sequence starts with $01\ldots$ :

$$U_{s,1} \leq 1, \qquad s \in [S]; \tag{4.22}$$

If $k$ is even, we distinguish between the critical search $s'$ with $U$-sequence $U_{s'} = 02 \ldots$ and the other (non-critical) searches $s$, starting with $U_s = 01 \ldots$. By looping over the different non-redundant possibilities for $s'$ (see below), we define a sequence of ILPs (one for each $s'$) with constraints

$$U_{s,1} \leq 1, \qquad s \in [S],\ s \neq s', \tag{4.23}$$

$$U_{s',1} = 2. \tag{4.24}$$

It is sufficient to find those schemes where the critical search $S_{s'}$ starts with a part in the first half (including part $k/2$) of the pattern, as the other schemes can be constructed by mirroring the permutations, i.e., using $\bar{\pi}[i] := (p-1) - \pi[i]$, but keeping the same $U$- and $L$-sequences.

#### 4.4.1.2 Implementation

We use the CPLEX 22.1.1 solver, implementing the ILP model in C++ with the Concert Technology API [4]. The solver uses up to 32 threads (the maximum for CPLEX) on a 64-core Intel® Xeon® E5-2698 v3 CPU, running at a base clock frequency of 2.30 GHz CPU.

To speed up solving the model to optimality, we provide a warm start. This start can be any valid search scheme. The given scheme is then first greedily improved (using only the upper bounds loop) before starting the ILP. As a first step, the ILP is solved with all lower bounds fixed to zero. Afterwards, the found search scheme is again greedily adapted. This solution with improved lower bounds is then given as initial solution (warm start) to the main ILP, which then computes the overall optimum. This last step, solving the full ILP, may be omitted for large values of $k$ to get good (but sub-optimal) solutions fast, as solving the full ILP takes the majority of the running time. Resulting search schemes are presented in Section 4.6.2. For $k = 6$, the program finished within 1 hour, for $k = 7$ the optimal solution was found within 1.5 hours.

## 4.5 Dynamic Selection

In scenarios where there are multiple co-optimal search schemes that each have a notable load imbalance among their searches, dynamic selection of search schemes proves to be a valuable strategy. See the table in Example 4.3 for two co-optimal search schemes. These co-optimal solutions are equivalent (from the ILP's point of view), but differ in which search is critical, which may lead to different performance on specific patterns and texts. This variability arises from the fact that (for practically usable search schemes) each search starts with an exact search of one of the parts (each $U_s$ starts with 0), and the number of exact matches for each part can be very different depending on the contents of $P$ and $T$, while the ILP does not consider these contents. Hence, the size of the search space for the corresponding search also fluctuates.

The crucial insight here is that when the critical search, determined by the lexicographically highest $U$-sequence, notably differs from the other searches, it typically dominates the overall search time, as it results in an expected broader search trie structure and consequently a larger search space (i.e. more nodes visited). By preemptively identifying the part of $P$ with the fewest exact matches, we can select the co-optimal search scheme for which the critical search starts with this part, to reduce the expected search space size, as only very few exact matches have to be extended by a large number of errors in the next part.

One notable instance is when $k$ is even and $k+1 = p = S$. In this specific scenario, co-optimal solutions arise, for which the critical search's $U$-string starts with 02, while all other $U$-strings start with 01.

---

**Example 4.3.** *Consider a DNA read that starts with a low-complexity part (say, a repeat of* `CA`*) with* many *exact matches in the reference genome, but all other parts are highly specific and contain differences with respect to the reference genome, so there are* no *exact matches. Assume that we want to find all occurrences of the read within*

*edit distance 4, using 5 parts and 5 searches, where each search starts with a different part. Consider the following two minU search schemes (co-optimal with equal ILP objective value), in which the critical search is marked with an asterisk:*

| | *Scheme A* | | | *Scheme B* | | |
|---|---|---|---|---|---|---|
| $s$ | $\pi_s$ | $L_s$ | $U_s$ | $\pi_s$ | $L_s$ | $U_s$ |
| *0* | *01234* | *01114* | *01444* | *01234* | *00222* | *02244** |
| *1* | *10234* | *00003* | *01444* | *12034* | *00000* | *01244* |
| *2* | *23410* | *01111* | *02244** | *21034* | *01111* | *01244* |
| *3* | *32410* | *00000* | *01244* | *34210* | *00003* | *01444* |
| *4* | *43210* | *00222* | *01244* | *43210* | *01114* | *01444* |

*In Scheme A, the critical search (which covers error distribution* 02020*) has index* $s = 2$*. It first searches part 2 exactly and finds zero exact matches (according to our scenario) and immediately stops. Search* $s = 0$ *that first searches part 0 exactly and finds many exact matches, extends all of these to part 1, allowing only 1 error, which creates much less work than extending them allowing 2 errors. In contrast, in Scheme B, the critical search has index* $s = 0$*. It first searches part 0 exactly and finds many exact matches, all of which have to be extended allowing up to 2 errors in part 1. With dynamic selection scheme A will be chosen. We observed search space reductions of up to 88% in practice in such scenarios using dynamic selection.*

## 4.6 Experiments and Results

### 4.6.1 Dataset and Computational Environment

All benchmarks were done using a dataset of 100 000 Illumina NovaSeq 6000 reads (151 bp) randomly sampled from a larger whole genome sequencing dataset (SRR9091899). We exhaustively identified approximate read occurrences up to an edit distance of $k = 2, 3, \ldots, 7$ on both strands of the human reference genome (GRCh38, [9]) where non-ACGT characters (e.g., Ns) were replaced with a randomly selected nucleotide. The chromosomes were concatenated into a single sequence. Spurious occurrences that span the borders of adjacent chromosomes can be easily filtered during post-processing.

Performance benchmarks were obtained using a single core of a 64-core Intel® Xeon® E5-2698 v3 CPU, operating at a base clock frequency of 2.30 GHz. Each benchmark run was repeated 20 times. We report the average wall clock time and the standard deviation.

### 4.6.2 Better Search Schemes

In our comparative analysis of search schemes we explore an array of approaches. From literature, we included Kucherov et al.'s schemes with $k+1$ and $k+2$ parts [1], the pigeonhole principle-based schemes [10], the 01*0-seeds-based schemes [8], Kianfar et al.'s scheme for $k+1$ parts [2], and $\text{Man}_{\text{Best}}$, a manual adaptation of the scheme by

***Table 4.3:*** *Overview of search schemes proposed in the literature and in this work for $k = 4$ errors. Each scheme is given as a sequence of searches; each search $s$ is given as $(\pi_s, L_s, U_s)$. Different constructions yield schemes with different number of parts (e.g., $p = k+1$ or $p = k+2$) and different number of searches $S$. Our minU schemes with $S = p = k + 1$ have multiple co-optimal versions (see also Section 4.5); only one is shown.*

| Name | Search Scheme |
|---|---|
| Kucherov $k + 1$ [1] | (01234, 00000, 02244), (43210, 00000, 01344), (10234, 00133, 01334), (01234, 00133, 01334), (32410, 00011, 01244), (21034, 00013, 01244), (10234, 00124, 01244), (01234, 00034, 00444) |
| Kucherov $k + 2$ [1] | (012345, 000000, 012344), (123450, 000000, 012344), (543210, 000001, 012244), (345210, 000012, 011344), (234510, 000023, 011244), (453210, 000133, 003344), (012345, 000333, 003344), (012345, 000044, 002444), (231045, 000124, 002244), (453210, 000044, 001444) |
| Pigeonhole principle [10] | (01234, 00000, 04444), (12340, 00000, 04444), (23410, 00000, 04444), (34210, 00000, 04444), (43210, 00000, 04444) |
| 01*0 [3, 8] | (012345, 000000, 014444), (123450, 000000, 014444), (234510, 000000, 014444), (345210, 000000, 014444), (453210, 000000, 004444) |
| $\text{Man}_{\text{Best}}$ [3] | (012345, 000004, 033344), (123450, 000000, 022334), (213450, 011111, 022334), (321450, 012222, 012334), (543210, 000033, 004444) |
| Kianfar [2] | (01234, 00004, 03344), (12340, 00000, 22334), (43210, 00033, 00444) |
| Greedy Kuch. $k + 1$ | (01234, 00002, 02244), (01234, 00334, 01334), (01234, 00344, 00444), (10234, 01334, 01334), (10234, 01224, 01244), (21034, 00113, 01244), (32410, 00111, 01244), (43210, 00000, 01344) |
| Greedy Pigeonhole | (01234, 01234, 02344), (10234, 00123, 01444), (21034, 00000, 01244), (32104, 00011, 01334), (43210, 00002, 01344) |
| Greedy 01*0 | (012345, 000004, 012444), (123450, 000033, 012334), (234510, 000222, 012244), (345210, 001111, 011444), (453210, 000000, 003444) |
| minU (ILP) | (01234, 01114, 01444), (10234, 00003, 01444), (23410, 01111, 02244), (32410, 00000, 01244), (43210, 00222, 01244) |

Kianfar et al. for $k = 4$ [3]. Additionally, we include the search schemes created by our greedy algorithm, namely the greedy adaptions of the search schemes by Kucherov et al. with $k + 1$ parts, the search schemes based on the pigeonhole principle ($p = k + 1$) and the search schemes based on 01*0-seeds ($p = k + 2$). Finally, we include the minU schemes created by the ILP. All search schemes, including the co-optimal minU schemes, are listed in the Appendix in tables A.1 to A.8. Additionally, Table 4.3 shows the search schemes for $k = 4$ (only one of the co-optimal solutions is shown for the minU search schemes).

To assess the different search schemes, Table 4.4 lists, for different values of $k$, the average number of nodes visited in the search trie across the approximate matching procedure of all sequences and their reverse complements. This metric directly reflects the size of the search space and serves as an objective measure of efficiency, regardless of implementation quality. Smaller search spaces indicate more efficient search schemes. Each visited node represents a single character extension, which, using a bidirectional FM-index, is executed in constant time. Consequently, reducing the search space results in shorter runtimes. Note that for even values of $k$, the number of nodes is averaged out over the globally co-optimal minU schemes.

Overall, the minU schemes consistently yield the smallest search space across all

***Table 4.4:*** *Average number of visited nodes (proportional to search time) for different search schemes. Search schemes above the center line have been proposed in the literature (A dash '-' indicates no results are available), while those below the center line are proposed in this work. Approximately co-optimal values are highlighted in boldface; values improved from standard minU by dynamic selection (only for even k) are highlighted in bold italics (N/A: not applicable for odd k).*

| Search scheme | Source | $k=2$ | $k=3$ | $k=4$ | $k=5$ | $k=6$ | $k=7$ |
|---|---|---|---|---|---|---|---|
| Kucherov $k+1$ | [1] | 689 | 1472 | 4500 | - | - | - |
| Kucherov $k+2$ | [1] | 858 | 1627 | 5020 | - | - | - |
| Pigeonhole principle | [10] | 847 | 3073 | 10467 | 31200 | 81908 | 191668 |
| 01*0 | [8] | 808 | 2183 | 5729 | 13861 | 31104 | 67347 |
| $\text{Man}_{\text{Best}}$ | [3] | - | - | 5030 | - | - | - |
| Kianfar | [2] | **637** | 9359 | 36967 | - | - | - |
| Greedy Kucherov $k+1$ | here | **640** | **1287** | 3643 | - | - | - |
| Greedy Pigeonhole | here | **640** | 1433 | 3493 | 6572 | 13592 | 21723 |
| Greedy 01*0 | here | 711 | 1528 | 3246 | 6281 | 11426 | 19622 |
| minU (ILP) | here | **638** | **1287** | **2943** | **6041** | **9978** | **16542** |
| minU + dynamic | here | ***567*** | N/A | ***2859*** | N/A | ***9618*** | N/A |

values of $k$, confirming that minimizing the $U$-sequence leads to better performing search schemes in practice. Notably, for $k > 2$, the simple greedy adaptations of the pigeonhole principle and the 01*0 seeds already outperform all other search schemes previously proposed in literature, demonstrating the power of this fast algorithm.

For $k = 2$, the scheme by Kianfar et al., the greedy adaptations with $k+1$ parts, and the minU schemes are equivalent. Small performance differences that can be observed are due to minor differences (e.g. mirroring) and the specific dataset that was used. For $k = 3$ and $k = 4$, the schemes by Kucherov et al. were, prior to this work, the fastest search schemes available. Our ILP-optimal minU schemes reduce the search space by respectively 12.6% and 34.6%. For $k = 3$, the greedy adaptation of Kucherov et al.'s search scheme leads to the same search scheme as the one discovered by the ILP, and hence to the same search space size. For $k \geq 5$, no prior efficient search schemes had been proposed in the literature. As a baseline, we make use of the pigeonhole principle and the 01*0-strategy, where the 01*0-strategy can be considered the prior state-of-the-art. The greedy adaptations of the 01*0 schemes reduce the search space by 54.7%, 63.3% and 70.9% for $k = 5$, 6 and 7, respectively. The minU schemes reduce the search space by 56.4%, 67.9% and 75.4% for $k = 5$, 6 and 7, respectively.

Dynamic selection of minU schemes for even $k$ reduces the search space even more (Table 4.4, last row), by an additional 11.1%, 2.9% and 3.6% for $k = 2$, 4 and 6, respectively.

In conclusion, our results demonstrate that the minU schemes, along with the greedy adaptations, outperform prior state-of-the-art approaches in terms of reducing the search space. This reduction in the number of visited nodes within the search trie translates to significantly improved runtimes.

### 4.6.3 Application to Lossless Read Mapping

We benchmark the runtime of these search schemes using our read aligner Columba 1.2. It extends the previous version Columba 1.1, detailed in the previous chapter. Columba 1.2 can handle arbitrary (valid) search schemes, including the minU schemes with dynamic selection. It features a 64-bit mode to handle larger (pan-)genomes and supports outputs in Sequence Alignment/Map (SAM) format [11].

We examine two separate datasets. The first comprises 100 000 Illumina reads, each 150 base pairs long. The second dataset consists of 100 000 subreads derived from PacBio RS II sequencing, with each subread 50 base pairs in length. The first dataset will have many reads that map with low error rates, while the second dataset has many reads that remain unmapped, even with higher error rates.

We compare Columba to Bwolo [8], GEM [5], Yara [6] and BWA [7] in all-mapping mode. Note that Bwolo does not report the Compact Idiosyncratic Gapped Alignment Report (CIGAR) string of the alignments whereas all other tools do. For the GEM aligner, not all occurrences could be reported as the tool failed when using the `all` parameter. Therefore, GEM was configured to report at most 1000 occurrences per read. For $k \geq 5$, GEM failed even with this restriction. As stated in the previous chapter, in Yara, the parameter for specifying the number of allowed errors, which was incorrectly interpreted in the previous chapter, is unique compared to other aligners. Instead of directly setting a fixed number of allowed errors, Yara lets the user define this number as a percentage of the length of the read being analyzed. Only integer percentages from 0 to 10 are allowed. The underlying algorithm then multiplies this percentage with the read length and casts it down to an integer value. For this reason, with the reads of length 150 it is impossible to get results for 2 or 5 allowed errors, and with the reads of length 50 only up to 5 errors are supported. The results for Columba 1.1 use the best-known search schemes prior to this work. These are manual adaptations based on the schemes by Kucherov et al. for $k \leq 4$ and the 01*0 seeds for $k \geq 5$.

Table 4.5 shows the runtimes and peak RAM-usage for different tools and different values of $k$ for the Illumina reads of length 150. Note that Columba is currently not optimized for memory. Both versions of Columba show superior performance in terms of runtime over other lossless aligners. Columba 1.2 outperforms its predecessor for $k = 2, 4, 5$ and 6 and matches it for $k = 3$. For $k = 2$, the reduced runtime can be mainly attributed to the dynamic selection technique as the used search schemes are highly similar. For $k = 3$ the search schemes used by Columba 1.1 and 1.2 are similar, resulting in comparable runtimes. Starting from $k = 4$, we start to see the effects of the newly designed search schemes. Columba 1.2 is 14.0% faster than its predecessor for $k = 4$. For $k = 5$ and $k = 6$ the improvements are even higher, respectively 36.4% and 53.4%, as previously no optimized search schemes were available for these values.

Columba outperforms existing lossless aligners for all values of $k \geq 2$. Among the competing tools, GEM is the fastest, but it cannot handle the large number of occurrences for some reads. Bwolo and Yara require respectively 55 and 9 minutes to align reads with up to $k = 6$ errors; Columba 1.2 performs this task in only 75 seconds and is hence approximately 7× faster than the best competitor. BWA in all-mapping

mode is outperformed by the tools specifically designed for lossless APM, and it does not finish the alignment of the 100 000 reads for $k \geq 4$ errors in 3 hours. The analysis for $k = 1$ is not included as its optimal search scheme is trivial. In Chapter 3, we found that Yara outperformed Columba for this case, likely due to the overhead of a bidirectional index used in Columba.

***Table 4.5:*** *Runtime comparison of state-of-the-art lossless alignment tools, with the exception of BWA in 'mem' mode, which is a lossy alignment algorithm on* 100 000 *Illumina reads of length 150 base pairs. For the lossless alignment tools, the footnotes mention with which parameters the tools were ran. The Mem. column indicates the peak-RAM usage. The smaller value between brackets indicates the percentage of mapped reads. DNC stands for Did Not Complete, N/A stands for Not Applicable.*

| Tool | Lang. | Mem. | $k = 2$ (90.5%) | $k = 3$ (93.1%) | $k = 4$ (94.8%) | $k = 5$ (95.9%) | $k = 6$ (96.7%) |
|---|---|---|---|---|---|---|---|
| Columba 1.2[1] | C++ | 23 GB | 6.5 ± 0.1s | 12.6 ± 0.1s | 24.0 ± 0.2s | 46.4 ± 4.2s | 74.6 ± 3.7s |
| Columba 1.1[2] | C++ | 21 GB | 7.5 ± 1.2s | 12.9 ± 0.1s | 27.9 ± 0.2s | 72.9 ± 0.7s | 160.1 ± 6.1s |
| BWA[3] | C | 3 GB | 135.9 ± 3.0s | 1473.1 ± 29.5s | DNC (> 3h) | DNC (> 3h) | DNC (> 3h) |
| Bwolo | C++ | 7 GB | 25.0 ± 1.0s | 63.9 ± 2.0s | 189.2 ± 4.9s | 631.2 ± 14.2s | 3336.2 ± 85.1s |
| GEMv3[4] | C | 13 GB | 19.8 ± 2.2s | 37.0 ± 2.7s | 83.4 ± 5.1s | crash | crash |
| Yara v0.9.11[5] | C++ | 5 GB | N/A | 21.6 ± 1.6s | 84.2 ± 5.0s | N/A | 542.3 ± 22.6s |
| BWA-MEM (lossy) | C | 5 GB | 31.4 ± 0.5s (independent of $k$)(99.9%) | | | | |

BWA-MEM (which is lossy) does not require specifying a maximum error threshold $k$ and typically reports only a single candidate alignment position per read. The runtime reported for BWA-MEM includes the time it takes to read the index structure from disk, which is not the case for Columba 1.2. BWA-MEM can handle paired-end reads, a feature not yet present in Columba 1.2. Columba 1.1 outperforms BWA-MEM in terms of speed for $k = 2$ and $k = 3$. Moreover, When $k$ is set to 4, Columba 1.2 is over 20% faster than BWA-MEM, while the runtime for Columba 1.1 is similar to that of BWA-MEM. Even for higher values of $k$, the runtime of Columba 1.2 is less than three times the runtime of BWA-MEM. Strikingly, BWA-MEM, as a lossy tool, outputs only 100 086 alignment positions, whereas Columba 1.2 outputs respectively 6 to 60 times as many alignment positions, depending on the specific choice of $k$.

The results for the PacBio subreads (length 50) for edit distance 4, 5, 6 and 7 are presented in table 4.6.

Columba 1.2 can align these short reads with high error rates in under 3 minutes for $k = 4$ and in 2 hours and 15 minutes for $k = 7$. Except for BWA in aln mode with $k = 4$, none of the other lossless tools could finish aligning these 100 000 subreads within 3 hours. Bwolo and Yara needed respectively 9 and 16 hours to finish this task with $k = 4$. For $k = 5$, no competing lossless aligner is able to finish within 3 hours. These results underscore the efficiency and effectiveness of Columba 1.2 in handling challenging alignment scenarios and highlights the superior performance of Columba 1.2, especially in scenarios with higher error rates.

[1] `-e [k] -i 5 -ss multiple /path/to/minU/schemes` for even $k$
`-e [k] -i 5 -ss custom /path/to/minU/scheme` for odd $k$

[2] `-e [k] -i 5 -ss custom /path/to/kuch_k+1_adapted/` for $k \leq 4$
`-e [k] -i 5 -ss custom /path/to/01star0/` for $k \geq 5$

[3] `aln -N -n [k] -i 0 -l 150 -k [k]`

[4] `-t 1 -e [k] -s [k] -alignment-model edit -mapping-mode complete -M 1000`

[5] `-e [k] -s [k] -y full -t 1`

***Table 4.6:*** *Runtime comparison of state-of-the-art lossless alignment tools (with the exception of BWA in 'mem' mode, which is a lossy alignment algorithm), on* 100 000 *PacBio subreads of length 50 base pairs. For the lossless alignment tools, the footnotes mention with which parameters the tools were ran. The Mem. column indicates the peak-RAM usage. The smaller values between brackets indicates the percentage of mapped reads.*

| Tool | Lang. | Mem. | $k = 4$ (5.4%) | $k = 5$ (9.2%) | $k = 6$ (13.9%) | $k = 7$ (18.6%) |
|---|---|---|---|---|---|---|
| Columba 1.2 | C++ | 21 GB | 173.69 ± 4.54s | 711.09 ± 10.49s | 2040.56 ± 19.25s | 7570.57 ± 230.69s |
| Columba 1.1 | C++ | 20 GB | 174.96 ± 1.30s | 5085.88± 44.39s | DNC (> 3h) | DNC (> 3h) |
| BWA aln | C | 3 GB | 1208.94 ± 11.28s | DNC (> 3h) | DNC (> 3h) | DNC (> 3h) |
| Bwolo | C++ | – | DNC (> 3h) | DNC (> 3h) | DNC (> 3h) | DNC (> 3h) |
| GEMv3 | C | – | crash | crash | crash | crash |
| Yara v0.9.11 | C++ | – | DNC (> 3h) | DNC (> 3h) | N/A | N/A |
| BWA-MEM (lossy) | C | 5 GB | 17.16 ± 3.04s (independent of $k$) (4.6%) | | | |

Furthermore, our comparison between Columba 1.1 and Columba 1.2 revealed significant improvements in runtime efficiency, especially for higher error thresholds ($k > 4$). Columba 1.1 struggled to handle alignments within the 3-hour timeframe for $k > 4$ errors due to the lack of dedicated search schemes for these scenarios. In contrast, Columba 1.2's introduction of novel search schemes and dynamic selection capabilities facilitated more efficient alignments, even for larger error thresholds.

Although the lossy BWA-MEM offers significantly faster performance, it reports a lower mapping rate than Columba for $k \geq 4$. This discrepancy is likely because BWA-MEM fails to identify good maximal exact matches, as its algorithm is tailored to reads longer than 70 base pairs. Notably, BWA-MEM can map only 4.6 percent of the reads, whereas with lossless alignment and 7 allowed errors, 18.6 percent of the reads could be mapped, highlighting the substantial difference in mapping efficiency between the two approaches.

## 4.7 Conclusion

We have introduced novel methods for designing efficient search schemes for lossless approximate pattern matching, allowing a larger number of errors than previously possible, up to $k = 13$. This fills a critical gap in the field, as previously dedicated search schemes existed only for $k \leq 4$. We presented two novel approaches: a greedy algorithm and an ILP formulation, implemented in Hato, a versatile open-source tool capable of crafting search schemes using a combination of these approaches.

Furthermore, we introduced the concept of dynamic selection, tailoring the choice of search scheme to the characteristics of each input read. The reduction in search space achieved through our newly designed search schemes and dynamic selection collectively amounts to 69% for $k = 6$ errors. These ideas are implemented in Columba 1.2, which significantly outperforms existing lossless read aligners.

Our results suggest a narrowing performance gap between lossless and lossy alignment tools. The inclusion of a strata-based search strategy, where $k$ is dynamically selected per read, could result in a read alignment tool with stronger guarantees for optimality. Given the performance advances in lossless approximate pattern matching, and

given the fact that most Illumina reads can be aligned with few errors, next-generation lossless mappers need not be slower than state-of-the-art lossy mappers such as BWA.

## Acknowledgments

An early version of this chapter was published as part of the 2024 Annual International Conference on Research in Computational Molecular Biology (RECOMB).

## Authorship contribution statement

**Luca Renders**: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft. **Lore Depuydt**: Software, Validation, Writing - Review & Editing. **Sven Rahmann**: Methodology, Writing - Review & Editing. **Jan Fostier**: Conceptualization, Supervision, Writing - Review & Editing.

## Authors' Disclosure

The authors declare that they have no conflict of interest.

## Funding Statement

Luca Renders and Lore Depuydt are funded by the Research Foundation – Flanders (FWO), through a PhD Fellowship SB (1SE7822N) and a PhD Fellowship FR (1117322N), respectively.

# References

[1] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[2] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert. *Optimum Search Schemes for approximate string matching using bidirectional FM-index*. arXiv preprint arXiv:1711.02035, 2017. doi:10.48550/arXiv.1711.02035.

[3] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

[4] IBM-ILOG. *CPLEX*, 2022. Accessed on Jul. 2, 2023. Available from: https://www.ibm.com/docs/en/icos/22.1.1?topic=documentation-introducing-ilog-cplex-optimization-studio-2211.

[5] S. Marco-Sola, M. Sammeth, G. R, and P. Ribeca. *The GEM mapper: fast, accurate and versatile alignment by filtration*. Nature Methods, 9(December):1185–1188, 2012. doi:10.1028/nmeth.2221.

[6] E. Siragusa. *Approximate string matching for high-throughput sequencing*. PhD thesis, Freie Universität Berlin, 2015. doi:10.17169/refubium-15562.

[7] H. Li and R. Durbin. *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

[8] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[9] V. Schneider, T. Graves-Lindsay, K. Howe, N. Bouk, H.-C. Chen, P. Kitts, T. Murphy, K. Pruitt, F. Thibaud-Nissen, D. Albracht, R. Fulton, M. Kremitzki, V. Magrini, C. Markovic, S. McGrath, K. Steinberg, K. Auger, W. Chow, J. Collins, and D. Church. *Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly*. Genome Research, 27, 2017. doi:10.1101/gr.213611.116.

[10] 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 2009 IEEE International Conference on Bioinformatics and Biomedicine, pages 31–36, 2009. doi:10.1109/BIBM.2009.42.

[11] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup. *The Sequence Alignment/Map format and SAMtools*. Bioinformatics, 25(16):2078–2079, June 2009. doi:10.1093/bioinformatics/btp352.

# 5

# Columba: Fast Approximate Pattern Matching with Optimized Search Schemes

*In the preceding chapters of this thesis, we laid the foundation for understanding and developing lossless approximate pattern matching through the design and implementation of Columba. Beginning with its early iterations, we explored the technical underpinnings and algorithmic principles that formed the core of this tool, progressively refining its functionality. These chapters provided a detailed roadmap of the advancements that brought Columba from a proof-of-concept to a robust and versatile tool.*

*This final chapter marks the culmination of these efforts by introducing Columba as a full-fledged aligner designed to address the challenges posed by modern genomic data analysis. Expanding upon the capabilities of its earlier versions, Columba 2.0 incorporates significant enhancements such as multi-threading for performance scaling, paired-end read support for higher-quality alignments, and a novel alignment mode optimized for reporting the best alignment(s). Furthermore, it includes support for a memory-efficient run-length compressed index, tailored for handling pan-genome data — a crucial feature in the age of large-scale genomic analyses.*

*To evaluate the efficacy and versatility of Columba 2.0, we test its performance across three diverse scenarios: an HRG dataset, a bacterial pan-genome, and a human haplotype pan-genome. These scenarios are carefully chosen to highlight Columba's adaptability to a range of biological data types and alignment challenges.*

*Additionally, we revisit the motivating example introduced at the start of this thesis - HLA typing. Here, we demonstrate how integrating Columba 2.0 into an existing pipeline significantly improves runtime while maintaining high accuracy, thereby underscoring its practical utility in real-world applications.*

*This chapter not only represents the apex of Columba's development but also bridges the technical contributions discussed earlier with the practical implications of lossless approximate pattern matching in genomics.*

---

**L. Renders, L. Depuydt, T. Gagie, and J. Fostier.**

**Submitted to *Bioinformatics*, Mar. 2025.**

**Abstract** Aligning sequencing reads to reference genomes is a fundamental task in bioinformatics. Aligners can be broadly categorized as lossy or lossless: lossy aligners prioritize speed by reporting only one or a few high-scoring alignments, whereas lossless aligners comprehensively output all optimal alignments, ensuring completeness, accuracy, and sensitivity. In this paper, we introduce Columba, a high-performance lossless aligner tailored for Illumina sequencing data. Columba accepts isolated or paired-end reads in FASTQ format and produces alignments in SAM format. Leveraging search schemes and bit-parallel alignment techniques, Columba achieves exceptional speed while maintaining the full precision expected of lossless tools. It is available in two variants: one based on the bidirectional FM-index and another, Columba RLC, which achieves run-length compression using a bidirectional move structure. The latter significantly reduces memory usage for large, repetitive datasets like pan-genomes. Through comprehensive benchmarking, Columba outperforms existing lossless aligners in speed, especially at higher error rates. Tests were conducted on the human genome as well as bacterial and human pan-genome datasets, demonstrating Columba's robustness and efficiency. We integrated Columba into the OptiType HLA genotyping pipeline, where it substantially reduced computational time while maintaining accuracy. These results position Columba as a versatile, state-of-the-art tool for high-sensitivity genomic analyses.

## 5.1 Introduction

This chapter introduces Columba[1] as a comprehensive, user-friendly, and high-performance lossless aligner that addresses the challenges posed by modern sequence alignment tasks. Building upon the technical contributions outlined in earlier chapters, Columba is now presented as a fully realized tool, designed for practical use in diverse genomic applications.

[1] available at:https://github.com/biointec/columba

***Table 5.1:*** *Comparison of aligners. RLC: run-length compression; configurable: tool is lossy, but can be configured to perform lossless alignment.*

| Aligner | Lossy/ lossless | Max. error rate/n° errors | RLC | Index structure |
|---|---|---|---|---|
| RazerS3 | lossless | 50% | no | no index |
| Yara | lossless | 10% | no | unidirectional |
| Bwolo | lossless | unlimited | no | unidirectional |
| BWA-aln | configurable | unlimited | no | unidirectional |
| Bowtie | configurable | 3 errors | no | unidirectional |
| GEM | configurable | unlimited | no | unidirectional |
| Bowtie2 | lossy | unlimited | no | unidirectional |
| BWA-MEM | lossy | unlimited | no | unidirectional |
| Ropebwt3 | lossy | unlimited | yes | bidirectional |
| Columba | lossless | 13 errors | no | bidirectional |
| Columba RLC | lossless | 13 errors | yes | bidirectional |

We begin by revisiting and expanding upon the categorization of aligners presented in Section 1.5. Aligners are broadly classified into lossy and lossless categories (see Table 5.1), each with distinct trade-offs between speed and sensitivity. Lossy aligners, such as Bowtie2 [1] and BWA-MEM [2], utilize a seed-and-extend approach that enables rapid identification of good alignments. These tools, however, rely on heuristics and typically report only a limited subset of alignments, offering no guarantees of completeness. For instance, a maximal exact match (MEM) seed used by these tools may contain sequencing errors or variants, causing certain valid alignments to be overlooked. Consequently, lossy aligners often struggle in applications such as pan-genomic analysis, metagenomics classification, and Human Leukocyte Antigen (HLA) typing, where comprehensive reporting of all alignment positions is crucial.

In contrast, lossless aligners, or *exact* aligners, guarantee the reporting of all co-optimal alignments under an optimality criterion, typically defined by edit distance (e.g., Levenshtein distance). Notable examples include RazerS3 [3], Yara [4], Bwolo [5], and DREAM-Yara [6], which have been widely used in tasks requiring exhaustive alignment reporting. However, tools like BWA-aln [2], Bowtie [7], and GEM [8] offer only partial support for lossless alignment, often failing to enumerate all co-optimal alignments (see Table 5.1 for details). Moreover, Ropebwt3 [9], one of the first tools aimed at pan-genomic references, reports only one alignment location, along with the number of alignments.

While lossless tools guarantee completeness, their adoption is often hindered by computational inefficiency. This performance gap between lossy and lossless aligners motivated Columba's development. Designed for exhaustive alignment with competitive runtime efficiency, Columba includes several key features:

- **Multiple alignment modes**: including *all* mode for reporting all occurrences with up to $k$ errors and *all-best* mode for identifying all co-optimal alignments with the smallest edit (or hamming) distance.

- **Run-length compressed indexing**: leveraging the b-move [10] structure to significantly reduce memory usage for pan-genomic datasets while maintaining high alignment sensitivity.

- **Paired-end alignment**: robustly supporting paired-end read input with proper pair reporting based on inferred fragment size.

- **User-friendly design**: Columba offers a user-friendly design with support for standard input and output formats (FASTQ, SAM), intuitive configuration, and multi-threaded execution for scalability. Additionally, Columba automatically builds the index for the user using standard FASTA file(s) as input, handling the concatenation of sequences and replacing non-ACGT characters. Results are reported in SAM format, directly referencing the sequence from the input file they originate from, ensuring seamless integration and traceability.

- **Support for search schemes**: Our approach supports arbitrary search schemes, including the out-of-the-box use of the best-known schemes (for up to 13 errors). In Chapter 3, we described a limitation where only 64-bit words were used in the bit-parallel banded alignment matrix, restricting support to a maximum of 10 errors. To overcome this, the program now dynamically employs 128-bit words when necessary, enabling efficient support for the edit distance metric with $k > 10$.

To evaluate Columba, we benchmark its performance against state-of-the-art aligners. The benchmarks include alignment tasks across human reference genomes, bacterial pan-genomes, and human haplotype pan-genomes, showcasing Columba's adaptability and efficiency in diverse contexts. Additionally, we study HLA typing as a case study, demonstrating Columba's practical utility in improving runtime while maintaining high alignment accuracy.

### 5.1.1 Bidirectional Move Structure

Columba offers a run-length compressed (RLC) variant. To support lossless approximate pattern matching with search schemes in run-length compressed space, Columba RLC replaces the bidirectional FM-index with the bidirectional *move structure*, or b-move. The emphasis on run-length compression in the context of BWT-based indexes stems from the observation that the BWT often contains long runs - consecutive occurrences of the same character - especially for highly repetitive reference texts like pan-genomes, making it inherently compressible.

The $r$-index was the first index to support both counting and locating exact occurrences of patterns in $O(r)$ space, where $r$ is the number of runs in the BWT [11,12]. [13] proposed the $br$-index as a bidirectional extension of the $r$-index, supporting bidirectional character extensions and enabling search schemes in run-length compressed space.

The move structure, proposed by [14], forms an alternative to the $r$-index, now supporting $O(1)$-time LF operations. More importantly, the move structure has a largely linear memory access pattern, which leads to a significantly higher performance than the $r$-index in practice. b-move is an extension of the move structure that enables bidirectional character extensions in a conceptually similar manner to the bidirectional

FM-index and the *br*-index. Like the move structure, b-move supports fast, cache-efficient character extensions in run-length compressed space [10]. To reduce space usage, bit-packing is used in the move tables at the cost of minimal performance loss. We rely on the SDSL library for sparse bit vector implementations [15].

## 5.2 Material and Experimental Setup

In a first benchmark, we align 1 million Illumina reads (avg. length 151 bp), randomly sampled from a large whole genome sequencing dataset (accession no. SRR9091899) to the human reference genome (hg38) [16]. Next, we explore HLA typing using OptiType [17]. We aligned slices of 1 012 CRAM files containing whole-exome sequencing (WES) data from the 1000 Genomes Project on the GRCh38 reference [18], utilizing the dataset described in [19], to an HLA-gene reference database [17] [2].

In a third benchmark, we align 1.6 million Illumina reads (accession no. SRR26678643, avg. length 250 bp) to a pan-genome comprising 335 *Listeria monocytogenes* genomes (see Section D.2). We also look at a pan-genome consisting of 8 154 bacterial strains, which was constructed by scraping NCBI RefSeq for all strains corresponding to the following 8 species: *Escherichia coli*, *Salmonella enterica*, *Listeria monocytogenes*, *Pseudomonas aeruginosa*, *Bacillus subtilis*, *Limosilactobacillus germentum*, *Enterococcus faecalis*, and *Staphylococcus aureus* (also see Section D.2).

In the final benchmark we align the same 1M Illumina reads as in the first benchmark to a human pan-genome. We use 2, 4, 8, 16, 32 and 64 haplotypes from the 1-year freeze of the human pan-genome reference [20].

We compare Columba to Yara, BWA, and Bowtie (both versions 1 and 2) in our benchmark analysis on the human genome. In the HLA-typing experiment, we compare the default RazerS3-based pipeline to a modified Columba-based variant. For the bacterial pan-genome, we compare Columba (RLC) and Ropebwt3. Finally, for the pan-genome consisting of human haplotypes, we compare Columba, Columba RLC, and Ropebwt3 to analyze scalability. Due to an unexpected error encountered during experimentation, GEM [8] was excluded from these comparisons.

The benchmarks on a single human genome, the bacterial pan-genome and for HLA-typing were obtained using a single thread of a 64-core Intel Xeon E5-2698 v3 CPU, operating at a base clock speed of 2.30 GHz, with 256 GiB of available RAM. The human pan-genome benchmark used a single thread of a 64-core AMD EPYC 7773X (Milan-X @ 2.2 GHz) processor, with 940 GiB of RAM available. All experiments were run 10 times; the standard deviation was always lower than 1.7%. The lossless tools use the edit distance. Command details are in Section D.1.

For the construction of the suffix array in the bidirectional FM-index, Columba employs `libsais`[3] and `libdivsufsort`[4], choosing between them based on the length of the text. In Columba RLC, there are two methods for index construction: an in-memory approach and a prefix-free parsing method [21]. While the in-memory

[2] https://raw.githubusercontent.com/FRED-2/OptiType/refs/heads/master/data/hla_reference_dna.fasta
[3] https://github.com/IlyaGrebnov/libsais
[4] https://github.com/y-256/libdivsufsort

method is faster, it requires a substantial amount of RAM, particularly for large pan-genomes. Conversely, the prefix-free parsing method, though slower, is significantly more memory-efficient.

We compare Columba to Yara, BWA and Bowtie (1 and 2) in the benchmark on the human genome. In the HLA-typing experiment we compare the default pipeline, that makes use of RazerS3 to a pipeline that uses Columba instead. For the bacterial pan-genome we also look at Columba (RLC) and Ropebwt3. Finally, for the pan-genome consisting of human haplotypes we compare only Columba, Columba RLC and Ropebwt3 to look at scaling.

### 5.2.1 Comparing the Output of Different Aligners

The aligners report results in SAM format [22], except for Ropebwt3, which reports in PAF format[5]. Ropebwt3 only reports 1 alignment per read along with the number of alignments. BWA-aln outputs alignments in the intermediate SAI format and requires BWA-SAMSE to convert its output to SAM format.

Comparing the output of various aligners is challenging, even when using the SAM format. We examine all equally best alignments per read, however not all aligners restrict their output to the best alignment; for example, BWA and Bowtie sometimes report additional alignments with a worse score. Additionally, supplementary co-optimal alignments are handled differently: Columba lists them as separate records, while BWA uses the XA tag. Yara also uses the XA tag but in a different format, though it can output separate records for supplementary alignments without CIGAR strings, which complicates SAMtools [23] compatibility. Similarly, Columba RLC never provides a CIGAR string, while Columba always reports a CIGAR string.

The interpretation of maximum error rate (or minimum identity score) can slightly vary across the different tools, resulting in tiny variations in the output. Finally, slight variations in alignment positions under an edit distance model necessitate some manual inspection, as aligners may produce subtly different results for the same region.

## 5.3 Results

### 5.3.1 Alignment to the Human Reference genome

We align 1 000 000 Illumina reads (151 bp in length) to the human reference genome (hg38) [16] using different tools. The lossless alignment tools were configured to report all co-optimal alignments in “all-best” mode. Table 5.2 shows the runtime, peak memory usage, and percentage of reads aligned for each tool.

In terms of runtime, Columba significantly outperforms existing lossless aligners. It is 1.4×, 1.9×, 8.1×, 25.8×, and 37.8× faster than Yara, its closest competitor, at error rates of 0%, 2%, 4%, 6%, and 8%, respectively. This highlights the efficiency of optimized search schemes for lossless approximate pattern matching, especially at higher error rates. The runtimes also indicate that BWA-aln was not optimized for

[5] https://github.com/lh3/miniasm/blob/master/PAF.md

***Table 5.2:*** *Runtime and peak memory usage of lossless alignment tools (Columba, Yara, BWA in aln mode, and Bowtie) at varying maximum error rate thresholds, and lossy alignment tools (BWA-MEM and Bowtie2 in normal and very sensitive (VS) modes), for aligning 1 000 000 Illumina reads (length 151 bp) from a larger WGS dataset to the human reference genome using a single thread. The alignment percentage of reads is noted alongside each runtime.*

| **Tool** | **Peak Memory** | **Maximum Error Rate** | | | | |
|---|---|---|---|---|---|---|
| | | 0% | 2% | 4% | 6% | 8% |
| **Lossless alignment tools** | | | | | | |
| **Columba** (this paper) | 11.06 GB | **24s** (66.6%) | **45s** (91.1%) | **1m 27s** (95.2%) | **4m 33s** (96.8%) | **21m 09s** (97.7%) |
| Yara | 4.94 GB | 33s (66.6%) | 1m 24s (91.1%) | 11m 42s (95.2%) | 1h 57m 10s (96.8%) | 11h 11m 42s (97.7%) |
| BWA-aln | 4.51 GB | 1m 2s (66.6%) | 41m 16s (91.1%) | 3h 38m 9s (95.0%) | 8h 41m 28s (96.5%) | 13h 55m 54s (97.2%) |
| Bowtie | 2.29 GB | 37s (66.6%) | 2m 28s (89.9%) | not supported | not supported | not supported |
| **Lossy alignment tools** | | **All Error Rates** | | | | |
| BWA-MEM | 5.19 GB | 6m 46s (99.8%) | | | | |
| Bowtie2 | 3.22 GB | 7m 53s (98.8%) | | | | |
| Bowtie2 (VS) | 3.23 GB | 17m 01s (99.0%) | | | | |

lossless alignment, consistent with its manual, which notes that the -N option may lead to significantly lower performance. Bowtie is slower than Yara and Columba at 0% and 2% error rates and does not support lossless alignment beyond 3 errors (2% for this dataset). When comparing Columba to lossy tools, we observe that up to a maximum error rate of 6% (a reasonable threshold for Illumina data), Columba is faster than the lossy alternatives. Even at an error rate of 8%, Columba's runtime remains competitive with Bowtie2 in 'very sensitive' (VS) mode. Due to its use of a bidirectional index, Columba requires slightly more than twice the RAM of the other tools. However, at 11.06 GB, it still fits comfortably on a modern laptop.

In principle, all lossless alignment tools should produce optimal - and therefore identical - results. Columba and Yara produce near-identical results, with minor differences arising from a slightly different interpretation of the error rate, different handling of 'N' characters and how overlapping alignments are handled. For example, one tool may list overlapping alignments as separate alignments, while others may flag one as redundant. In contrast, BWA-aln and Bowtie do not always report all (co-)optimal alignments, as evidenced by slightly lower alignment percentages (Table 5.2). BWA-aln does not report 1 812 560 co-optimal alignments at a 0% maximum error rate, with this number increasing to 2 538 119 at an 8% maximum error rate. Similarly, at a 2% error rate threshold, Bowtie fails to find an alignment for 15 524 reads. Additionally, for 109 reads, Columba and Yara report alignments with a smaller edit distance, and for 491 reads, they provide additionally co-optimal alignments.

Lossy alignment tools align a higher fraction of reads: at a 6% error rate threshold, Columba mapped 96.8% of the reads, while BWA-MEM found alignments for nearly all (99.8%) reads. In absolute numbers, BWA-MEM aligned 30 195 reads more than Columba, all of which had error rates > 6%, often due to spliced alignments or read clipping. On the other hand, BWA-MEM failed to align four (short) reads that were successfully aligned with Columba and Yara. For 3 190 reads, Columba reports alignments with a smaller edit distance than those reported by BWA-MEM, 1 058 of which are identified at entirely different positions in the reference genome. For many reads,

BWA-MEM reports only a single alignment, whereas Columba exhaustively reports all co-optimal alignments, resulting in 2 514 588 additional alignments that are reported by Columba. These alignments represent more than 69% of all alignments reported by Columba. Despite reporting many more alignments, Columba was faster at this task.

In Table 5.3, we align the same 1 000 000 Illumina reads (151 bp in length) as in Table 5.2, now incorporating paired-end read filtering with their corresponding mates. Among lossless aligners, Columba consistently proves to be the fastest, completing the alignment in just 19m 54s at a 6% error rate, significantly outperforming Yara (4h 32m 07s) and BWA-aln (17h 01m 55s). Moreover, Columba's runtime remains competitive with lossy alternatives. The comparison of output across aligners is complicated by the distinct methods each tool uses to define "proper pair" mappings. Columba dynamically infers the mean fragment size and orientation, reporting pairs within six standard deviations of the detected mean fragment size. When multiple such pairs are possible, it selects those with the minimal combined edit distance.

The significantly higher alignment percentage observed for BWA-aln with lower error rates can be attributed to its inclusion of alignments with clippings and its occasional allowance of errors in the mate, even when the maximum edit distance is strictly set to 0. Conversely, the significantly lower alignment percentage observed for Bowtie (both versions 1 and 2) can be attributed to its use of a fixed insert size, with default values of 250 and 500, respectively.

***Table 5.3:*** *Runtime and peak memory usage of lossless alignment tools (Columba, Yara, BWA in aln mode, and Bowtie) at varying maximum error rate thresholds, and lossy alignment tools (BWA-MEM and Bowtie2 in normal and very sensitive (VS) modes), for aligning 1 000 000 pairs of Illumina reads (length 151 bp) from a larger WGS dataset to the human reference genome using a single thread. The alignment percentage of paired reads (reads mapped in proper pair) is noted alongside each runtime.*

| **Tool** | **Maximum Error Rate** | | | | |
|---|---|---|---|---|---|
| | 0% | 2% | 4% | 6% | 8% |
| **Lossless alignment tools** | | | | | |
| **Columba** | **44s** (45.4%) | **1m 57s** (82.7%) | **5m 01s** (89.8%) | **19m 54s** (92.9%) | **1h 49m 21s** (94.7%) |
| Yara | 11m 04s (42.0%) | 7m 14s (79.1%) | 29m 36s (85.5%) | 4h 32m 07s (88.2%) | 1d 00h 55m 19s (89.8%) |
| BWA-aln | 2m 42s (73.0%) | 1h 22m 42s (89.8%) | 7h 28m 21s (94.5%) | 17h 01m 55s (94.6%) | 1d 02h 50m 36s (94.6%) |
| Bowtie | **46s** (7.4%) | 1h 11m 10s (11.2%) | not supported | not supported | not supported |
| **Lossy alignment tools** | | | **One Timing** | | |
| BWA-MEM | | | 16m 59s (99.5%) | | |
| Bowtie2 | | | 13m 55s (84.5%) | | |
| Bowtie2 (VS) | | | 25m 45s (84.7%) | | |

## 5.3.2 HLA Typing from NGS Data Using OptiType

Lossless alignment tools are crucial for various bioinformatics applications, including Human Leukocyte Antigen (HLA) typing from next-generation sequencing (NGS) data. Accurate, high-resolution HLA typing is essential for understanding immune system function and has significant applications in organ transplantation compatibility, autoimmune disease research, and cancer immunotherapy.

In a recent survey paper on HLA typing tools [19], OptiType [17] emerged as one of the most accurate tools for predicting HLA genotypes from NGS data. It focuses on Major Histocompatibility Complex (MHC) Class I alleles, including HLA-A, HLA-B, and HLA-C. For each read, OptiType requires a complete list of all possible candidate alleles that the read could have originated from. To achieve this, OptiType uses the lossless aligner RazerS3 [3, 24] to map reads to a comprehensive collection of HLA allele sequences. Next, leveraging an integer linear programming (ILP) approach,OptiType identifies the most likely set of HLA alleles that explain the sequencing data.

While OptiType delivers precise genotyping, it was also found to be computationally intensive [19], partly due to the alignment step. To address this, we use Columba as a drop-in replacement for RazerS3, which required only minimal changes to OptiType's script (see Figure D.1 in Appendix D). Using the dataset from [19], we evaluated OptiType on slices of 1 012 CRAM files of whole-exome sequencing (WES) data from the 1000 Genomes on GRCh38 project [18]. Gold standard PCR-based HLA typings for these samples were obtained from three previous studies [25–27]. In cases of discrepancies, the calls made by Gourraud et al. [27] were favored to ensure consistency with the evaluation in [19].

Figure 5.1 (left) shows the runtime distribution of the alignment phase for both Columba and RazerS3 across the 1 012 samples. In both cases, 16 CPU cores were used. Columba achieved a median runtime of 4.83 s compared to 57.96 s for RazerS3, yielding a 12× speedup. Consequently, using Columba reduced the overall runtime of the OptiType pipeline (including the ILP step) by 44% (median value). As Columba and RazerS3 produce near-identical output, OptiType's accuracy remains unaffected. We further evaluated OptiType on 462 RNA-seq samples [28]. In this analysis, the median runtime for the alignment phase decreased significantly, from 4.82 minutes with RazerS3 to just 58 seconds with Columba. Figure 5.1 (right) illustrates the runtime distributions for the alignment phases of the two tools.

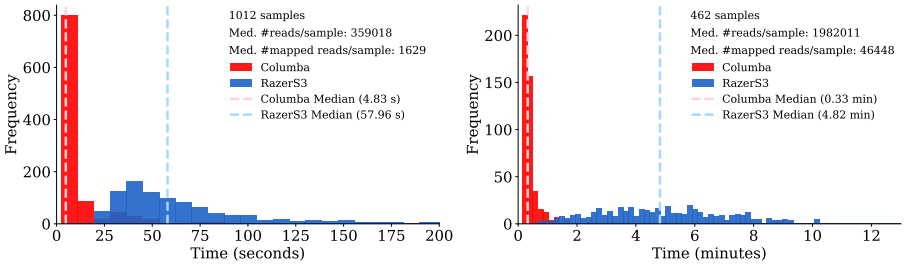


***Figure 5.1:*** *Left: Runtime distribution of OptiType's alignment phase using Columba and RazerS3 across 1 012 samples with DNA data. The x-axis is capped at 200 seconds. RazerS3's distribution continues beyond this value. Right: Runtime distribution of OptiType's alignment phase using Columba and RazerS3 across 462 samples with RNA data. The x-axis is capped at 13 minutes. RazerS3's distribution continues beyond this value.*

### 5.3.3 Alignment to Bacterial Pan-genomes

***Table 5.4:*** *Runtime and peak memory usage of lossless alignment tools (Columba, Columba RLC, Yara, BWA-aln, and Bowtie) at varying maximum error rate thresholds, and lossy alignment tools (BWA-MEM, Bowtie2 in normal and very sensitive (VS) modes, and Ropebwt3), for aligning 1.6 million Illumina reads (250 bp) to a pan-genome of 335 Listeria monocytogenes genomes using one thread. The alignment percentage of reads is noted alongside each runtime.*

| **Tool** | **Peak Memory** | **Maximum Error Rate** 0% | 1% | 2% | 3% | 4% | 5% |
|---|---|---|---|---|---|---|---|
| **Lossless alignment tools** | | | | | | | |
| **Columba** | 3.56 GB | 4m 17s (75.4%) | **5m 39s** (92.5%) | **6m 28s** (96.3%) | **6m 46s** (97.2%) | **7m 22s** (97.9%) | **7m 30s** (98.3%) |
| **Columba RLC** | 1.07 GB | 5m 56s (75.4%) | 7m 50s (92.5%) | 9m 05s (96.3%) | 9m 23s (97.2%) | 10m 18s (97.9%) | 10m 36s (98.3%) |
| Yara | 2.48 GB | 6m 23s (75.4%) | 10m 38s (92.5%) | 14m 35s (96.3%) | 15m 25s (97.2%) | 16m 52s (98.0%) | 17m 43s (98.3%) |
| BWA-aln | 1.64 GB | **2m 37s** (75.4%) | 21m 35s (92.5%) | 57m 30s (96.2%) | 1h 42m 59s (97.2%) | 3h 35m 30s (97.8%) | 5h 22m 4s (98.2%) |
| Bowtie | 0.77 GB | 12m 24s (75.4%) | 17m 22s (92.2%) | N/A | N/A | N/A | N/A |
| **Lossy tools** | | **All Error Rates** | | | | | |
| BWA-MEM | 2.20 GB | 44m 11s (99.9%) | | | | | |
| Bowtie2 | 1.09 GB | 21m 51s (99.4%) | | | | | |
| Bowtie2 (VS) | 1.08 GB | 23m 30s (99.4%) | | | | | |
| Ropebwt3 | 0.55 GB | 1h 30m 19s (99.9%) | | | | | |

We aligned 1 648 416 Illumina reads (250 bp) to a pan-genome comprising 335 *Listeria monocytogenes* genomes (987 328 727 bp in total) to evaluate runtime and peak memory usage. The lossless alignment tools were again run in *all-best* mode, a configuration suited for pan-genome analysis as it provides a comprehensive overview over all possible genomes a read could have originated from. This is particularly relevant for, e.g., bacterial species or strain identification and taxonomic sequence classification for metagenomics.

The comparison includes Columba RLC, the run-length compressed variant of Columba that employs the bidirectional move structure instead of the bidirectional FM-index, as well as Ropebwt3, a mapper specifically designed for pan-genome references. Table 5.4 shows that Columba is the fastest tool among the evaluated lossless aligners, outperforming Yara by up to 2.4×. Columba and Columba RLC produce identical outputs due to their shared alignment methodology, differing only in index structure. Yara and Columba (RLC) outputs are nearly identical, with discrepancies observed in only three specific reads caused by minor implementation differences, such as the handling of non-ACGT characters in the reference genome.

In contrast, BWA-aln does not comprehensively report co-optimal alignments, excluding more than 127 million co-optimal alignments, over 99% of those reported by Columba and Yara. This reduced output volume explains BWA-aln's faster runtime on this dataset at 0% error rate. Bowtie produces identical outputs as Columba and Yara at a 0% error rate, but does not consistently yield optimal or complete results at a 1% error rate: it aligns 249 reads with sub-optimal edit distances, fails to align 63 reads, and omits 9 281 co-optimal alignments. Lossy tools, such as BWA-MEM and Bowtie2 (VS), further sacrifice reporting co-optimal alignments, but can report alignments beyond 12 errors, which is reflected in a high alignment percentage.

Columba RLC reduces memory usage by 3.3× compared to Columba in Table 5.4. However, this memory efficiency comes at the cost of a 1.4× runtime increase compared to Columba. Even so, Columba RLC remains faster than other lossless alignment tools at error rates greater than 0%.

Since the bidirectional move structure has a reduced memory complexity of $O(r)$, its efficiency becomes increasingly pronounced as additional genomes are added to the reference text. For example, when indexing a larger pan-genome of 8 154 bacterial genomes (37.44 Gbp total, accession numbers in appendix D.2), Columba's bidirectional FM-index requires 166.7 GB of memory, whereas Columba RLC uses only 15.5 GB, achieving a 10.8× reduction.

Ropebwt3 achieves superior compression compared to Columba RLC but at the expense of significantly longer runtime. Ropebwt3 reports a single hit per read along with the count of equally optimal hits. For 97.5% of the reads (1 607 184), the alignments produced by Ropebwt3 matched the alignments reported by Columba within at most 50 bp. For 1.6% of the reads (26 360), Ropebwt3 reported alignments beyond 12 errors, which Columba failed to align. However, for 4 888 reads, Columba reported more alignments (328 228 alignments not reported by Ropebwt3), while Ropebwt3 produced a higher number of alignments for 5 739 reads (resulting in 78 312 additional alignments). As Columba and Yara are lossless tools and produce consistent outputs, we infer that the additional alignments reported by Ropebwt3 likely correspond to overlapping occurrences in the reference genome (less than 20 bp apart), which are considered spurious by Columba and Yara and/or wrongful reports.

Figure 5.2 illustrates the runtimes for Columba (RLC), Yara, Bowtie2, BWA-MEM, and Ropebwt3 when utilizing multiple threads. The lossless tools are configured with an allowed error rate of 5%. The results demonstrate that all tools benefit from multi-threading. However, at higher thread counts, the performance of the lossless tools, which output more alignments, starts to be constrained by disk-writing efficiency. Nevertheless, Columba remains one of the top performers.

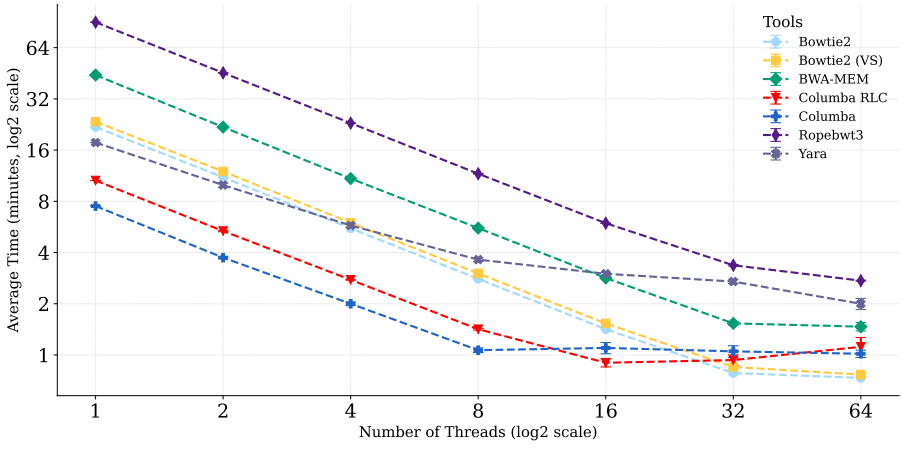


***Figure 5.2:*** *Multi-threaded timings for the benchmark where 1.6M reads are aligned to the pan-genome consisting of 335 Listeria monocytogenes species with various tools. The lossless tools (Yara and Columba) are configured with an allowed error rate of 5%. The lossless tools report significantly more occurrences than lossy aligners, leading to contention and throttling at higher thread counts due to disk write speed becoming a bottleneck.*

### 5.3.4 Alignment to Human Pan-genome Reference

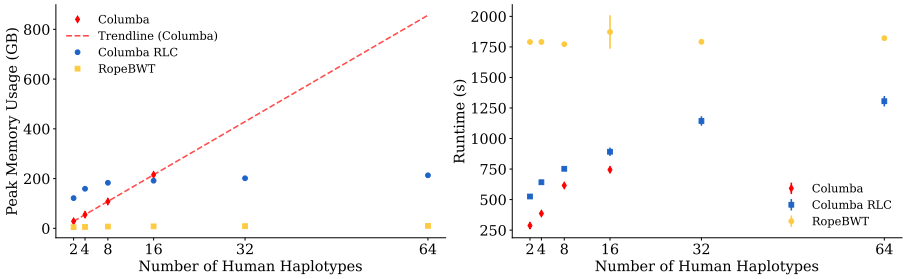


***Figure 5.3:*** *Comparison of peak memory usage (left) and runtime (right) among Columba, Columba RLC, and Ropebwt3 for aligning 1M reads to a pan-genome, with varying number of human haplotypes.*

We aligned 1M Illumina reads against pan-genomes containing up to 64 human haplotypes, sourced from the Human Pangenome Reference Consortium, which provides a comprehensive representation of genetic diversity [20]. Indexes were constructed for Columba, Columba RLC, and Ropebwt3. Columba and Columba RLC were executed in *all-best* mode with a maximum error rate of 6%, while Ropebwt3 was used with its default settings. Figure 5.3 (left) demonstrates that Columba's memory usage scales linearly with the number of haplotypes, as shown by the trendline. Therefore, we could evaluate Columba for only up to 16 human haplotypes, as constructing an index for 32 haplotypes exceeded the available RAM on the machine (∼940 GiB).

In contrast, Columba RLC and Ropebwt3 demonstrate more favorable memory requirements. Even though both tools leverage run-length compression on the BWT and share the same $O(r)$ memory complexity, Columba RLC's bidirectional move structure introduces a larger constant prefactor in its implementation. As a result, Ropebwt3 achieves a better memory footprint than Columba RLC in practice. In comparison to Columba Vanilla, Columba RLC exhibits lower memory usage only when processing more than 16 haplotypes. This suggests that Columba Vanilla is more efficient for smaller pan-genomes, aligning with previous observations by [29].

Figure 5.3 (right) illustrates the runtime for aligning 1M reads to the human pan-genome. Columba RLC has a performance penalty compared to Columba but remains significantly faster than Ropebwt3. The increase in runtime for Columba (RLC) with larger pan-genome sizes is attributed to the greater volume of output that needs to be produced, i.e., a larger number of co-optimal alignments. In contrast, Ropebwt3 reports only a single alignment and the number of co-optimal alignments.

For 956 398 reads (95.6%), both tools report the same number of alignments, with each alignment identified by Ropebwt3 also present in Columba's output. For 20 187 reads (2.0%), Ropebwt3 provides an alignment while Columba does not. A manual review of a subset of these cases revealed that these can be linked to an error rate exceeding 6%. On the other hand, for 16 reads Columba reports alignment(s), while Ropebwt3 does not report any alignments. For 10 474 reads (1.0%), Ropebwt3 did not

report (any of) the alignment(s) with the lowest edit distance, while for 6 871 reads, Columba reported additional alignments, leading to a total of 7 601 497 alignment counts not reported by Ropebwt3. Conversely, Ropebwt3 reported a higher number of alignments for 4 371 reads then Columba did. Since Ropebwt3 does not report detailed alignments but rather provides aggregate counts, it is challenging to ascertain why these alignments are not corroborated by Columba. It is plausible that Ropebwt3 either produces spurious alignment counts (e.g., alignments that are overlapping) or includes misreported results.

## 5.4 Discussion and Conclusion

We propose Columba, a fast and feature-rich lossless alignment tool. It accepts FASTQ or FASTA files as input and produces SAM records as output, with support for multi-threading to enhance performance. Unlike widely used lossy aligners such as BWA-MEM and Bowtie, Columba guarantees optimality and completeness of its output. Optimality is evaluated in terms of edit distance, a standard metric for assessing differences between reads and reference sequences. However, certain applications may benefit from alternative scoring schemes, such as affine gap penalties, which are likely more biologically relevant. In *all* mode, completeness means that all occurrences in the reference sequences with up to a predefined number of errors ($k$) are reported, with support for up to $k = 13$ errors. In *all-best* mode, completeness refers to reporting all co-optimal occurrences (i.e., occurrences with the lowest edit distance to the query read). Columba supports paired-end read mapping; in this mode, only alignment positions that conform to the expected fragment size are reported. If no such proper pair can be found, the reads are treated as single-ended reads.

Our evaluation demonstrates that Columba is significantly faster than other lossless alignment tools such as RazerS3 and Yara, especially when aligning reads with higher error rates. This performance improvement is achieved through optimized search schemes and an efficient bit-parallel implementation. However, as the search schemes require a bidirectional index, Columba's memory requirements are generally double those of tools that use a unidirectional index.

Columba supports both single-genome and pan-genome references. For large pan-genomes, Columba RLC leverages the bidirectional move structure, a run-length compressed index. This approach significantly reduces memory usage when multiple similar genomes are included in the pan-genome, albeit with a slight trade-off in speed. Other tools such as Ropebwt3 offer a different time-space tradeoff: Ropebwt3 is able to better compress pan-genomes, but appears slower. Additionally, Ropebwt3 does not report all alignments, only the alignment count.

Columba is a robust and versatile alignment tool suitable for various applications. Its integration into the OptiType pipeline for HLA genotyping demonstrates its practical utility. We believe Columba (RLC) is also relevant for tasks like metagenomics read classification, bacterial species or strain identification, and pan-genome graph alignment with visualization [30]. Additionally, Columba (RLC) can be used to benchmark the outputs of lossy alignment tools.

## Acknowledgments

This chapter is based on an article that was submitted to *Bioinformatics* in December 2024.

## Authorship contribution statement

The authors wish it to be known that, in their opinion, the first two authors should be regarded as joint first authors. **Luca Renders**: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft. **Lore Depuydt**: Software, Methodology, Validation, Writing - Review & Editing. **Travis Gagie**: Conceptualization, Writing - Review & Editing. **Jan Fostier**: Conceptualization, Supervision, Writing - Review & Editing.

## Authors' Disclosure

The authors declare that they have no conflict of interest.

## Funding Statement

Luca Renders and Lore Depuydt are funded by the Research Foundation – Flanders (FWO), through a PhD Fellowship SB (1SE7822N) and a PhD Fellowship FR (1117322N), respectively.

# References

[1] B. Langmead and S. L. Salzberg. *Fast gapped-read alignment with Bowtie 2*. Nature methods, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.

[2] H. Li and R. Durbin. *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

[3] D. Weese, M. Holtgrewe, and K. Reinert. *RazerS 3: Faster, fully sensitive read mapping*. Bioinformatics, 28(20):2592–2599, 08 2012. doi:10.1093/bioinformatics/bts505.

[4] E. Siragusa. *Approximate string matching for high-throughput sequencing*. PhD thesis, Freie Universität Berlin, 2015. doi:10.17169/refubium-15562.

[5] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[6] T. H. Dadi, E. Siragusa, V. C. Piro, A. Andrusch, E. Seiler, B. Y. Renard, and K. Reinert. *DREAM-Yara: an exact read mapper for very large databases with short update time*. Bioinformatics, 34(17):i766–i772, September 2018. doi:10.1093/bioinformatics/bty567.

[7] 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, 10:1–10, 2009. doi:10.1186/gb-2009-10-3-r25.

[8] S. Marco-Sola, M. Sammeth, G. R, and P. Ribeca. *The GEM mapper: fast, accurate and versatile alignment by filtration*. Nature Methods, 9(December):1185–1188, 2012. doi:10.1028/nmeth.2221.

[9] H. Li. *BWT construction and search at the terabase scale*. Bioinformatics, page btae717, November 2024. doi:10.1093/bioinformatics/btae717.

[10] L. Depuydt, L. Renders, S. Van de Vyver, L. Veys, T. Gagie, and J. Fostier. *b-move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index*. In S. P. Pissis and W.-K. Sung, editors, 24th International Workshop on Algorithms in Bioinformatics (WABI 2024), volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2024.10.

[11] T. Gagie, G. Navarro, and N. Prezza. *Optimal-Time Text Indexing in BWT-runs Bounded Space*. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

[12] T. Gagie, G. Navarro, and N. Prezza. *Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space*. J. ACM, 67(1), January 2020. doi:10.1145/3375890.

[13] Y. Arakawa, G. Navarro, and K. Sadakane. *Bi-Directional r-Indexes*. In 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022), volume 223 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2022.11.

[14] T. Nishimoto and Y. Tabei. *Optimal-Time Queries on BWT-Runs Compressed Indexes*. In 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference), volume 198 of *LIPIcs*, pages 101:1–101:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.101.

[15] S. Gog, T. Beller, A. Moffat, and M. Petri. *From Theory to Practice: Plug and Play with Succinct Data Structures*. In J. Gudmundsson and J. Katajainen, editors, Experimental Algorithms, pages 326–337, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07959-2_28.

[16] V. Schneider, T. Graves-Lindsay, K. Howe, N. Bouk, H.-C. Chen, P. Kitts, T. Murphy, K. Pruitt, F. Thibaud-Nissen, D. Albracht, R. Fulton, M. Kremitzki, V. Magrini, C. Markovic, S. McGrath, K. Steinberg, K. Auger, W. Chow, J. Collins, and D. Church. *Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly*. Genome Research, 27, 2017. doi:10.1101/gr.213611.116.

[17] A. Szolek, B. Schubert, C. Mohr, M. Sturm, M. Feldhahn, and O. Kohlbacher. *OptiType: precision HLA typing from next-generation sequencing data*. Bioinformatics, 30(23):3310–3316, August 2014. doi:10.1093/bioinformatics/btu548.

[18] X. Zheng-Bradley, I. Streeter, S. Fairley, D. Richardson, L. Clarke, P. Flicek, and the 1000 Genomes Project Consortium. *Alignment of 1000 Genomes Project reads to reference assembly GRCh38*. GigaScience, 6(7):gix038, May 2017. doi:10.1093/gigascience/gix038.

[19] A. Claeys, P. Merseburger, J. Staut, K. Marchal, and J. V. den Eynden. *Benchmark of tools for in silico prediction of MHC class I and class II genotypes from NGS data*. BMC Genomics, 24(1):247, 2023. doi:10.1186/s12864-023-09351-z.

[20] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel, et al. *A draft human pangenome reference*. Nature, 617(7960):312–324, 2023. doi:10.1038/s41586-023-05896-x.

[21] C. Boucher, T. Gagie, A. Kuhnle, and G. Manzini. *Prefix-Free Parsing for Building Big BWTs*. In L. Parida and E. Ukkonen, editors, 18th International Workshop on Algorithms in Bioinformatics (WABI 2018), volume 113

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2018.2.

[22] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup. *The Sequence Alignment/Map format and SAMtools*. Bioinformatics, 25(16):2078–2079, June 2009. doi:10.1093/bioinformatics/btp352.

[23] P. Danecek, J. K. Bonfield, J. Liddle, J. Marshall, V. Ohan, M. O. Pollard, A. Whitwham, T. Keane, S. A. McCarthy, R. M. Davies, and H. Li. *Twelve years of SAMtools and BCFtools*. GigaScience, 10(2), 02 2021. giab008. doi:10.1093/gigascience/giab008.

[24] 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, July 2009. doi:10.1101/gr.088823.108.

[25] S. Boegel, M. Löwer, M. Schäfer, T. Bukur, J. De Graaf, V. Boisguérin, Ö. Türeci, M. Diken, J. C. Castle, and U. Sahin. *HLA typing from RNA-Seq sequence reads*. Genome medicine, 4:1–12, 2013. doi:10.1186/gm403.

[26] Y. Huang, J. Yang, D. Ying, Y. Zhang, V. Shotelersuk, N. Hirankarn, P. C. Sham, Y. L. Lau, and W. Yang. *Accurate HLA Typing at High-Digit Resolution from NGS Data*. International Journal of Bioengineering and Life Sciences, 9(3):263–271, 2015.

[27] P.-A. Gourraud, P. Khankhanian, N. Cereb, S. Y. Yang, M. Feolo, M. Maiers, J. D. Rioux, S. Hauser, and J. Oksenberg. *HLA diversity in the 1000 genomes dataset*. PloS one, 9(7):e97282, 2014. doi:10.1371/journal.pone.0097282.

[28] T. Lappalainen, M. Sammeth, M. R. Friedländer, P. A. '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. doi:10.1038/nature12531.

[29] L. Depuydt, L. Renders, S. Van de Vyver, L. Veys, T. Gagie, and J. Fostier. *b-move: Faster Lossless Approximate Pattern Matching in a Run-Length Compressed Index*, November 2024. PREPRINT. doi:10.21203/rs.3.rs-5367343/v1.

[30] L. Depuydt, L. Renders, T. Abeel, and J. Fostier. *Pan-genome de Bruijn graph using the bidirectional FM-index*. BMC Bioinform., 24(1):400, 2023. doi:10.1186/S12859-023-05531-6.

# 6
# Conclusion and Future Work

*This final chapter provides a summary of the key contributions of this thesis, reflecting on the advancements made in the field of lossless approximate pattern matching. It revisits the core findings, the methodologies employed, and the impact of the work presented throughout the previous chapters. In addition, the chapter discusses potential future directions, offering insights into how the research can be expanded and applied in broader contexts.*

This chapter begins with a discussion of the core contributions of this thesis, focusing on the advancements in lossless sequence alignment and their practical implications. It then transitions into potential future directions, categorized into theoretical advancements, hardware implementations, and diverse applications.

## 6.1 Discussion

Over the past few decades, bioinformatics has transitioned from a specialized computational field to a central pillar of modern biological and medical research. This transformation has been catalyzed by rapid advancements in sequencing technologies and the algorithms that process their outputs. While early efforts were constrained by the computational demands of aligning even small sequences, today's challenge lies in efficiently managing and analyzing the trillions of base pairs generated daily. This thesis addresses these challenges by advancing the state of the art in lossless sequence alignment, culminating in the development of Columba — a powerful, feature-rich tool that bridges the performance gap between lossy and lossless approaches.

In this work, we have contributed in five key ways:

- We introduced a universal algorithm for the dynamic partitioning of the search pattern based on the sequence content of both the pattern and the reference during the use of a search scheme. This significantly reduced the search space (by up to 28%) and runtime (by up to 25%). Additionally, to further optimize performance, we proposed a method for minimizing redundancy in edit distance calculations, reducing wasteful computations and shrinking the search space by up to 63% and runtime by 52%.

- We developed a hybrid strategy that combines in-index matching with bit-parallel in-text verification, dynamically optimizing the search process based on the distribution of candidate occurrences. The strategy utilizes in-text verification for candidate occurrences that are rare in the genome to minimize computational overhead, while for candidate occurrences that are frequent, it stays with batch processing within the index. This approach achieved more than a twofold runtime reduction compared to pure in-index methods.

- We proposed Hato, a tool for designing optimal search schemes using two novel approaches. Previously, only search schemes designed for up to four errors were available, along with inefficient schemes for arbitrary $k$. The ILP-based method is feasible for error thresholds up to $k = 7$. It identifies optimal search schemes under a new set of constraints proposed in this work, leading to efficient practical implementations. Additionally, for higher error thresholds up to $k = 13$, we introduced a greedy heuristic algorithm that independently improves the search schemes defined for arbitrary $k$. Furthermore, we introduced dynamic search scheme selection, which complements dynamic partitioning. This approach selects the search scheme most likely to yield a smaller search space based on the specific characteristics and partitioning of each read. Collectively, these advancements reduced the search space by up to 69% for $k = 6$ errors, enabling unprecedented scalability and efficiency in lossless sequence alignment.

- These contributions culminated in the development of Columba, a comprehensive tool for lossless alignment. Along with the efficient implementation due to the technical innovations in this work, it also supports multi-threading and paired-end alignment. Columba is compatible with both single-genome and pan-genome references and incorporates advanced features such as run-length compression for handling highly repetitive reference sequences. This broad range of capabilities makes Columba adaptable to a wide variety of genomic applications, from routine Illumina read alignments to more complex pan-genome analyses, demonstrating its robustness and efficiency in sequence alignment.

- We performed extensive benchmarking and demonstrated that Columba consistently outperformed state-of-the-art lossless aligners, such as RazerS3 and Yara, particularly in high-error scenarios. Compared to heuristic-based tools like Bowtie and BWA, Columba's runtime is competitive, and in some cases, even superior, while maintaining lossless guarantees. Its successful integration

into the OptiType pipeline for HLA genotyping showcased its practical utility, achieving significant reductions in computational time while preserving accuracy.

In conclusion, these contributions show that performance levels comparable to heuristic-based aligners can be achieved by optimizing lossless alignment methods through advancements in search schemes. Additionally, we have defined what constitutes an efficient search scheme and demonstrated how new or improved schemes can be designed to further enhance the performance of lossless alignment. Through the development of novel algorithms, hybrid strategies, and dynamic search scheme selection, we have significantly reduced the search space and computational overhead, enabling scalable and efficient exhaustive alignment.

## 6.2 Future Work

The continuous evolution of sequencing technologies, genomic representations, and computational algorithms provides exciting opportunities for innovation in sequence alignment and related areas. In our work, we introduced an efficient implementation of search schemes for lossless alignment into our tool, Columba. This section will discuss future research directions that aim to tackle existing challenges and broaden the scope of search schemes in genomics and other fields. It will focus on refining these search schemes, developing a GPU-based version of Columba to leverage GPU processing power, and investigating wider applications in genomics and beyond.

### 6.2.1 Theoretical Advancements: Design of New Search Schemes

The theoretical framework for estimating the size of the search space, as defined by Kucherov et al. [1], is constrained by its reliance on assumptions of randomness. As discussed in Chapter 4, search schemes developed under this framework often fail to achieve optimal performance when applied to real-world scenarios.

To address these limitations, Chapter 4 introduced new search schemes specifically designed to improve practical performance for Next-Generation Sequencing (NGS) read-mapping. Our approach is grounded in empirical observations of what constitutes an effective search scheme in this context, enabling us to tailor the designs to the distinctive characteristics of actual sequencing data. Compared to schemes generated solely from the theoretical framework, our empirically informed strategies deliver substantially better results in practice.

Despite these advances, it remains uncertain whether the principles that guided our NGS-focused designs are equally applicable to other sequencing contexts (see for example Section 6.2.3.5). Further investigation is needed to determine if these schemes, when adapted, can maintain their effectiveness for short sequences or for long-read data produced by third-generation sequencing technologies.

In tandem with these empirical explorations, the development of a more accurate theoretical model for estimating the search space could significantly advance the field. Such a refined model, closely aligned with real-world conditions, would not only

bridge the gap between theoretical predictions and practical outcomes but also help researchers strategically design search schemes suited to the specific characteristics of various sequencing contexts. This dual approach—refining empirical insights and improving theoretical frameworks—promises to guide the next generation of robust, efficient search schemes.

### 6.2.2 Hardware Implementations: Potential GPU Implementation

Graphics Processing Units (GPUs) are specialized hardware components originally designed to accelerate image rendering by performing parallel computations across numerous cores. Initially developed for rendering images on screens, GPUs handle tasks by executing a large number of small computations simultaneously, making them highly effective for parallelizable workloads like matrix operations, scientific simulations, and machine learning. GPUs operate in a highly parallel manner, executing instructions across multiple threads organized into groups called warps. Each warp consists of a fixed number of threads (typically 32) that execute instructions simultaneously in a Single Instruction, Multiple Threads (SIMT) model [2].

This architecture can be compared to a row of robotic arms connected to a single motor, where each arm works on its own task (e.g., processing different items). The motor controls all the arms, sending a signal for them to perform a specific operation. When this signal is given, each arm either processes its assigned task or, if it has nothing meaningful to do for that step, pauses by ignoring the motor's signal. If one arm encounters a more complex or time-consuming task, it continues working as directed by the motor, while the other arms, which do not require as much time, pause and wait. This ensures all the arms stay synchronized with the motor's signals, but it also means the idle arms are unproductive during this time. This dynamic mirrors how GPUs execute instructions in a warp: threads with more work continue processing while those with less work must wait, leading to inefficiencies caused by control flow divergence.

While GPUs are highly efficient for tasks where the same operation is applied to different data points, such as image processing, their architecture presents significant challenges for sequence alignment, which involves dynamic and irregular workflows. As seen in Chapter 3, some reads require extensive searching, while others are more straightforward to align and thus demand fewer computational resources. In our context, the use of dynamic partitioning and dynamic search scheme selection techniques results in highly divergent computational paths for individual reads. These variations lead to frequent thread divergence within a warp. The delays are dynamic and unpredictable. Threads in a warp may alternately need to wait for different tasks at different times — sometimes task 1, sometimes task 2, and so on — effectively serializing the workload. This behavior can result in runtime performance approaching that of sequential execution on a single CPU thread, thereby negating the potential parallelism of the GPU.

Conversely, in-text verification procedures (see Chapter 3) for reads of the same length present a more uniform computational structure, making them better suited for GPU acceleration. Each thread could verify a read in the bit-parallel alignment matrix against the reference independently, aligning well with the SIMT model. However,

this uniformity comes at a cost: the early halting mechanism used in the CPU implementation, which allows verification to stop when it is clear no match can be found at the current site, would need to be disabled for GPU execution. The lack of early halting could lead to inefficiencies that counteract the benefits of parallelism.

The question of whether GPU acceleration can consistently outperform the optimizations in current CPU implementations remains open. Although GPUs provide significant raw computational power, the dynamic and irregular nature of sequence alignment tasks might limit their efficiency. Future research could explore strategies such as workload balancing and dynamic thread scheduling on GPUs to address this variability. Additionally, redesigning in-text verification processes to run concurrently could enhance performance. Comparative benchmarking between GPU and CPU implementations will be crucial in determining whether GPUs can deliver a practical advantage for these tasks.

### 6.2.3 Expanding Applications of Search Schemes in Genomics and Beyond

The search schemes and algorithms presented in this work not only enhance read-mapping performance in next-generation sequencing contexts but also hold promise for a wide array of related genomic and biomedical applications. While this work primarily focused on achieving faster alignment without compromising accuracy, the same principles and search schemes can be applied to broader genomic and biomedical challenges. Beyond improving traditional read mapping, these techniques open new opportunities for handling long and noisy reads, aligning sequences to complex pan-genome graphs, and tackling tasks like mappability computation, PCR primer design, and CRISPR off-target detection. In the following sections, we highlight several potential avenues of exploration, illustrating how these foundational insights can be leveraged to tackle emerging and ongoing challenges in genomics, metagenomics, and beyond.

#### 6.2.3.1 Search Schemes for Finding Inexact Seeds for Longer Reads

While this work has demonstrated the utility of search schemes for read mapping, their capacity to handle full-length, highly error-prone long reads remains limited. We have designed search schemes allowing for up to thirteen errors per read, which are sufficient for shorter sequences but proves inadequate when dealing with the extended lengths and elevated error rates of third-generation sequencing (TGS) data. Given these constraints, a direct, full-length alignment approach guided solely by search schemes is unlikely to be effective in long-read contexts.

Instead, an approach that revisits the seed-and-extend paradigm may offer a more practical path forward. Traditionally, seed-and-extend relies on Maximal Exact Matches (MEMs) to anchor the alignment. However, for long, error-prone reads, MEMs often become very short and thus fail to significantly narrow down the search space. This results in a proliferation of candidate positions in the reference genome, increasing computational complexity and slowing the alignment process.

To address these challenges, one promising direction is to incorporate inexact seeds into the seed-and-extend workflow. By identifying slightly inexact but longer seeds rapidly, using search schemes, we could achieve a more balanced trade-off: seeds that are not strictly exact, yet still informative enough to localize the read effectively. Such inexact seeds could reduce the search space before attempting a full extension and detailed alignment, ultimately enhancing the scalability and efficiency of long-read mapping tools.

#### 6.2.3.2 Sequence-to-Graph Alignment Using Search Schemes

Graph-based genome representations are increasingly preferred over linear consensus genomes due to their ability to capture genetic diversity, incorporating single nucleotide polymorphisms (SNPs) and structural variations [3, 4]. Our group has developed a novel haplotype-aware sequence alignment method within pan-genome graphs, which improves alignment accuracy by leveraging known haplotype paths [5].

Future work could focus on extending this approach to align sequences that traverse multiple haplotypes within a graph. Such advancements would allow for the identification of novel haplotypes, recombination events, and structural variations not represented in existing haplotype paths. This would provide a more comprehensive understanding of genomic diversity and evolution.

#### 6.2.3.3 Genome Mappability

Genome mappability quantifies the uniqueness of sequences within a genome, offering insights into which regions allow for unique alignment of sequencing reads. This measure is essential for improving the accuracy and reliability of key genomic applications, including variant detection, genome assembly, and transcriptomic analyses. Low-mappability regions, often characterized by repetitive sequences or structural variations, pose significant challenges to alignment and interpretation. By understanding these regions, researchers can mitigate ambiguities and improve the robustness of bioinformatics pipelines.

For instance, in variant detection, ambiguous alignments in low-mappability regions can lead to uncertain or incorrect variant calls. Similarly, in genome assembly, repetitive sequences can cause fragmentation or misassembly, complicating efforts to reconstruct accurate genome sequences. Mappability information also plays a vital role in RNA sequencing, where misaligned reads from genes with low mappability can distort expression level estimates.

In prior work, Pockrandt et al. introduced GenMap, a tool that assesses mappability using search schemes designed under the Hamming distance metric. This method identifies the uniqueness of k-mers while allowing for up to $e$ mismatches, providing a robust framework for evaluating sequence similarity [6].

However, GenMap's reliance on the Hamming distance means it does not account for insertions or deletions, limiting its applicability in regions of the genome where structural variation is prevalent. Furthermore, it makes use of older search schemes, that are outperformed by our newly designed search schemes.

Building on these advancements, it is worth exploring whether the search schemes and optimization techniques introduced in this dissertation could further enhance mappability analysis. Integrating these more efficient implementations and dynamic strategies may enable faster mappability computation. Additionally, these techniques could enable the incorporation of the edit distance metric, which considers mismatches as well as insertions and deletions, offering a more comprehensive assessment of sequence uniqueness.

#### 6.2.3.4 PCR Primer Design

Primer design is a fundamental step in PCR (see Section 1.1.2.2), requiring the identification of short DNA sequences that bind (anneal) specifically to target genomic regions. Chemically, primers must meet stringent requirements: they should have a melting temperature suitable for the PCR conditions, a balanced GC content to ensure stable binding, and minimal secondary structure formation that could interfere with annealing [7]. In addition to these biochemical constraints, ensuring primer specificity so that only the intended target is amplified remains a challenging computational task.

A key approach to enhancing primer specificity involves pinpointing highly discriminatory genomic regions — sequences that uniquely distinguish a target locus from all other genomic regions. This principle has been exemplified by tools like AmpliDiff, developed for estimating lineage abundances in viral metagenomes. AmpliDiff uses identified unique regions to design lineage-specific primers, enabling precise differentiation, for example, among SARS-CoV-2 lineages in metagenomic samples.

However, AmpliDiff's reliance on multiple sequence alignments (MSA) imposes certain limitations. Since MSA is an NP-hard problem, heuristic methods are employed, often resulting in suboptimal alignments and, consequently, suboptimal selection of discriminatory regions. Moreover, MSA-based approaches are not ideally suited for genomes with high rates of recombination or extensive repetitive sequences, conditions commonly found in complex viral or eukaryotic genomes [8].

Previous chapters have introduced and demonstrated how Columba's exhaustive pattern searching could mitigate some of these issues, offering a more direct and robust method for verifying the uniqueness of potential primer sites. By systematically searching for candidate primers without relying solely on aligned positions, Columba would handle complex genomic structures more effectively than MSA-based strategies.

In the future, Columba's capabilities could be integrated into primer design pipelines to address these challenges. For instance, Columba could be applied in conjunction with or as a validation layer after MSA-based candidate selection, ensuring that primers identified as unique by alignment-dependent methods would indeed prove consistently unique under exhaustive pattern searches. Such an approach could confirm the specificity of primers, even in highly repetitive or recombination-prone regions, thereby improving the reliability and performance of PCR-based assays for diagnostics, lineage tracking, and other applications.

#### 6.2.3.5 CRISPR-Cas9 Off-Target Detection

CRISPR-Cas9 is a transformative genome-editing tool that enables precise modifications to DNA sequences within organisms. The system comprises the Cas9 enzyme, which acts as molecular scissors to cleave DNA, and a guide RNA (gRNA), which directs Cas9 to a specific genomic location. To ensure cleavage occurs at the desired site, the target DNA sequence must lie next to a short Protospacer Adjacent Motif (PAM). This PAM requirement helps determine where Cas9 can bind and cut, thereby contributing to the overall specificity of the system. However, a critical challenge in CRISPR-Cas9 applications is the occurrence of off-target effects, where the gRNA directs Cas9 to unintended genomic locations that share partial sequence similarity with the intended target. Off-target effects can result in undesired mutations, reducing the reliability of genome editing and posing risks in therapeutic applications [9].

To mitigate these risks, tools like VARSCOT have been developed. VARSCOT addresses the off-target problem by incorporating genomic variations, such as single nucleotide polymorphisms (SNPs) and insertions or deletions (indel), into its off-target analysis. By including variant information, VARSCOT identifies off-target sites that are specific to an individual's genome, improving the accuracy of its predictions. This is particularly important in contexts where natural genomic variations can alter the landscape of potential off-target effects [10].

VARSCOT uses the search schemes designed by Kianfar et al. to identify potential off-target sites across the genome via bidirectional alignment. This approach allows VARSCOT to identify off-target sites with a higher number of mismatches between the gRNA and the genome, enhancing its sensitivity. VARSCOT also employs machine learning models to score the likelihood of off-target cleavage by considering contextual factors such as Cas9 concentration and sequence context, further refining its predictions.

While the search schemes designed with Hato (see Chapter 4) share foundational principles with those used in VARSCOT, they have been specifically designed for read-mapping NGS reads to a reference genome. The focus of future work could explore whether these newly designed search schemes are equally effective for shorter sequences like gRNAs used in CRISPR-Cas9 applications. Additionally, integrating Columba's efficient read-mapping implementation into VARSCOT, could significantly reduce its runtime without changing the pipeline.

Future work should evaluate the feasibility and effectiveness of these integrations, comparing the performance of these enhanced tools against existing standards in CRISPR-Cas9 specificity analysis. Such studies have the potential to significantly improve both the precision and applicability of CRISPR-Cas9, especially in research and therapeutic settings requiring the utmost accuracy in genome editing.

#### 6.2.3.6 Species Identification in Metagenomics

The challenge of accurately identifying species and strains in metagenomic samples is particularly significant when dealing with closely related strains or low-abundance variants. This issue was highlighted by Saltykova et al. [11] in their study of Shiga

toxin-producing *Escherichia coli* (STEC) during a foodborne outbreak. Using metagenomic shotgun sequencing, they successfully linked pathogenic strains in food samples to human isolates, demonstrating the potential of strain-level metagenomic analysis. However, they also noted significant difficulties in distinguishing closely related strains, emphasizing the need for more refined analytical methods.

Existing tools provide valuable methodologies for addressing this challenge. For example, OptiType employs comprehensive read alignment and an integer linear programming (ILP) model to optimize read assignments, achieving precise identification of HLA alleles, even those that are closely related. Similarly, StrainGE has developed a scoring model to identify the genomes most likely to match strains present in a given sample, offering a practical approach for strain-level characterization.

Building on these methods, Columba's exhaustive alignment capability could form the basis of a model that integrates comprehensive read reporting with advanced scoring and assignment strategies. Combining Columba's capabilities with OptiType's optimization techniques and StrainGE's scoring framework could produce a tool that outperforms current approaches in strain-level resolution. Such a model would offer finer granularity and more precise results, even in complex scenarios like mixed infections or strains with minimal genomic differences.

#### 6.2.3.7 Minimum Bait Cover problem

The Minimum Bait Cover Problem is a computationally challenging task central to designing synthetic DNA probes, or "baits", used in targeted sequencing. Bait design is essential because it enables the enrichment of specific genomic regions from a complex DNA pool, allowing researchers to focus sequencing efforts on areas of interest. Unlike PCR-based methods, which rely on primers for each specific target and are prone to amplification biases and failures, hybridization-based bait capture scales efficiently to a large number of targets and avoids PCR-specific artifacts. These advantages make bait design particularly critical in applications where scalability, precision, and robustness are paramount [12].

The goal of the Minimum Bait Cover Problem is to identify the smallest set of baits, each of a fixed length, that collectively cover every position in a reference genome while allowing for a specified number of mismatches. This optimization is crucial in DNA enrichment workflows, where efficient bait design enhances the precision and cost-effectiveness of sequencing efforts. Applications of bait design span diverse fields, including metagenomics, where underrepresented sequences are selectively enriched; evolutionary biology, where homologous genes across species are captured for phylogenetic analysis; medical genetics, where disease-associated genomic regions are targeted; and ancient DNA research, where degraded fragments are retrieved for analysis. The computational challenges arise from minimizing bait sets while ensuring robust coverage of target sequences, even in the presence of sequence variations [13].

Syotti, a tool developed by Alanko et al., addresses the Minimum Bait Cover Problem by leveraging the concept of $\theta$-coverage. In this framework, a bait $\theta$-covers a genomic position if it matches a substring within a given number $\theta$ of mismatches, measured by Hamming distance. This approximate matching approach allows baits

to tolerate minor sequence variations, making them effective for capturing diverse or degraded DNA [13].

Building on these advances, future work could use Columba's capabilities to improve the runtime and computational complexity of solving the Minimum Bait Cover Problem with Hamming distance. Columba's efficient algorithms for approximate matching could accelerate the identification of baits that $\theta$-cover the genome, making the process more computationally feasible for large and complex datasets. Additionally, incorporating edit distance into the bait design framework represents a promising direction. By considering both mismatches and small indels, edit distance could enable the design of baits that are more robust to structural variations, thereby enhancing their applicability in diverse and challenging genomic contexts. Additionally, the practical benefits of incorporating edit distance into the $\theta$-coverage framework should be investigated, with an emphasis on understanding its impact on bait set size, coverage efficiency, and computational scalability.

#### 6.2.3.8 Plagiarism Detection

Plagiarism detection is essential for upholding academic integrity, aiming to identify instances where text has been inappropriately copied or closely paraphrased without proper attribution. This process becomes particularly challenging when individuals employ obfuscation techniques such as paraphrasing, synonym substitution, or introducing typographical errors to disguise the original source. Extrinsic plagiarism detection involves comparing a suspicious document against a reference corpus to identify overlapping content. Typically, the text is preprocessed to normalize and segment it into tokens, which are then compared systematically to identify similarities. Tokenization plays a crucial role by dividing the text into smaller units, such as words or phrases, which serve as the fundamental elements for comparison [14].

The search schemes developed in this dissertation, designed for efficient lossless approximate pattern matching with predefined error thresholds, present a potential application in plagiarism detection. These schemes could facilitate the identification of instances where copied text has been modified to evade detection. However, the effectiveness of these techniques in the context of plagiarism detection, where textual structure and semantics are pivotal, remains to be explored.

By integrating token similarity scoring — assigning values based on semantic or orthographic resemblance — these search schemes could provide a more nuanced assessment of textual similarity, improving the system's ability to detect obfuscated plagiarism. However, the practical implementation of such an approach would requires careful consideration to avoid significant computational overhead or compromising detection sensitivity.

Future research could focus on adapting search schemes to the specific requirements of plagiarism detection workflows, including token-level similarity assessment and chunk-based analysis. Evaluating their performance on extensive reference databases is essential to determine their scalability and effectiveness. Comparative studies with existing tools are necessary to assess the potential improvements offered by this approach in detecting paraphrased or altered text.

## Final Remarks

The advancements presented in this thesis lay a robust foundation for future research in both theoretical and applied aspects of lossless sequence alignment. A key achievement of this work is the successful closure of the performance gap between lossy and lossless sequence alignment approaches. By introducing innovative algorithms and optimization techniques, the solutions developed herein not only clearly outperform existing lossless methods but also achieve runtime competitiveness with lossy approaches. This is accomplished while maintaining the exhaustive and exact nature inherent to lossless alignment, ensuring accuracy and completeness in results. From optimizing search schemes to expanding their applications in genomics and beyond, the outlined directions offer a roadmap for continued innovation. By addressing these challenges, this work has the potential to impact multiple applications in computational biology and beyond.

# References

[1] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[2] A. Habermaier and A. Knapp. *On the correctness of the SIMT execution model of GPUs*. In European Symposium on Programming, pages 316–335. Springer, 2012. doi:10.1007/978-3-642-28869-2_16.

[3] A. M. Novak, G. Hickey, E. Garrison, S. Blum, A. Connelly, A. Dilthey, J. Eizenga, M. A. S. Elmohamed, S. Guthrie, A. Kahles, S. Keenan, J. Kelleher, D. Kural, H. Li, M. F. Lin, K. Miga, N. Ouyang, G. Rakocevic, M. Smuga-Otto, A. W. Zaranek, R. Durbin, G. McVean, D. Haussler, and B. Paten. *Genome Graphs*. bioRxiv, 2017. doi:10.1101/101378.

[4] A. M. Novak, G. Hickey, E. Garrison, S. Blum, A. Connelly, A. Dilthey, J. Eizenga, M. A. S. Elmohamed, S. Guthrie, A. Kahles, S. Keenan, J. Kelleher, D. Kural, H. Li, M. F. Lin, K. Miga, N. Ouyang, G. Rakocevic, M. Smuga-Otto, A. W. Zaranek, R. Durbin, G. McVean, D. Haussler, and B. Paten. *Variation graph toolkit improves read mapping by representing genetic variation in the reference*. Nat. Biotechnology, Oct 2018. doi:10.1038/nbt.4227.

[5] L. Depuydt, L. Renders, T. Abeel, and J. Fostier. *Pan-genome de Bruijn graph using the bidirectional FM-index*. BMC bioinformatics, 24(1):400, 2023. doi:10.1186/s12859-023-05531-6.

[6] C. Pockrandt, M. Alzamel, C. S. Iliopoulos, and K. Reinert. *GenMap: ultra-fast computation of genome mappability*. Bioinformatics (Oxford, England), 36(12):3687–3692, Jun 2020. doi:10.1093/bioinformatics/btaa222.

[7] S. A. Bustin, V. Benes, J. A. Garson, J. Hellemans, J. Huggett, M. Kubista, R. Mueller, T. Nolan, M. W. Pfaffl, G. L. Shipley, J. Vandesompele, and C. T. Wittwer. *The MIQE Guidelines: Minimum Information for Publication of Quantitative Real-Time PCR Experiments*. Clinical Chemistry, 55(4):611–622, 04 2009. doi:10.1373/clinchem.2008.112797.

[8] J. v. Bemmelen, D. S. Smyth, and J. A. Baaijens. *Amplidiff: an optimized amplicon sequencing approach to estimating lineage abundances in viral metagenomes*. BMC bioinformatics, 25(1):126, 2024. doi:10.1186/s12859-024-05735-4.

[9] C. Guo, X. Ma, F. Gao, and Y. Guo. *Off-target effects in CRISPR/Cas9 gene editing*. Frontiers in bioengineering and biotechnology, 11:1143157, 2023. doi:10.3389/fbioe.2023.1143157.

[10] L. O. Wilson, S. Hetzel, C. Pockrandt, K. Reinert, and D. C. Bauer. *VARSCOT: variant-aware detection and scoring enables sensitive and personalized off-target detection for CRISPR-Cas9*. BMC biotechnology, 19:1–7, 2019. doi:10.1186/s12896-019-0535-5.

[11] A. Saltykova, F. E. Buytaers, S. Denayer, B. Verhaegen, D. Piérard, N. H. Roosens, K. Marchal, and S. C. De Keersmaecker. *Strain-level metagenomic data analysis of enriched in vitro and in silico spiked food samples: paving the way towards a culture-free foodborne outbreak investigation using STEC as a case study*. International Journal of Molecular Sciences, 21(16):5688, 2020. doi:10.3390/ijms21165688.

[12] B. Sundararaman, A. O. Vershinina, S. Hershauer, J. Kapp, S. Dunn, B. Shapiro, and R. E. Green. *A method to generate capture baits for targeted sequencing*. Nucleic Acids Research, 51(13):e69–e69, 06 2023. doi:10.1093/nar/gkad460.

[13] J. N. Alanko, I. B. Slizovskiy, D. Lokshtanov, T. Gagie, N. R. Noyes, and C. Boucher. *Syotti: scalable bait design for DNA enrichment*. Bioinformatics, 38(Supplement_1):i177–i184, 06 2022. doi:10.1093/bioinformatics/btac226.

[14] T. Foltýnek, N. Meuschke, and B. Gipp. *Academic Plagiarism Detection: A Systematic Literature Review*. ACM Comput. Surv., 52(6), October 2019. Available from: doi.org/10.1145/3345317, doi:10.1145/3345317.

# A

# Overview of Search Schemes

*In this appendix, an overview of existing search schemes is provided, encompassing both those established in the literature and those developed as part of this research.*

---

The following tables present several search schemes. Each search is represented on a new line. A search is written as $(\pi, U, L)$, see Definition 1.14. If the $U$, $L$, and $\pi$ arrays consist of single digits (for $k < 10$), they are shown as a string of these digits, otherwise, spaces are placed between numbers.

The presented search schemes are those based on the pigeon hole principle, those based on 01*0-seeds [1, 2], the search schemes presented by Kucherov et al. [3], the search schemes presented by Kianfar et al. [4], the manually designed search scheme by Pockrandt [2], the greedy adaptions of existing search schemes, found by Hato, and the minU search schemes designed by Hato (see Chapter 4 for more info on Hato).

***Table A.1:*** *The search schemes based on the pigeonhole principle and the greedily adaptions of these schemes found by using Hato for up to $k = 7$. The left column is the original scheme and the right column is the greedily adapted search scheme.*

| | Original | Greedily adapted |
|---|---|---|
| $k = 2$ | (012, 000, 022)<br>(120, 000, 022)<br>(210, 000, 022) | (012, 012, 022)<br>(102, 000, 012)<br>(210, 001, 012) |
| $k = 3$ | (0123, 0000, 0333)<br>(1023, 0000, 0333)<br>(2310, 0000, 0333)<br>(3210, 0000, 0333) | (0123, 0000, 0133)<br>(1230, 0023, 0223)<br>(2310, 0001, 0133)<br>(3210, 0112, 0133) |
| $k = 4$ | (01234, 00000, 04444)<br>(12340, 00000, 04444)<br>(23410, 00000, 04444)<br>(34210, 00000, 04444)<br>(43210, 00000, 04444) | (01234, 01234, 02344)<br>(10234, 00123, 01444)<br>(21034, 00000, 01244)<br>(32104, 00011, 01334)<br>(43210, 00002, 01344) |
| $k = 5$ | (012345, 000000, 055555)<br>(234510, 000000, 055555)<br>(123450, 000000, 055555)<br>(453210, 000000, 055555)<br>(345210, 000000, 055555)<br>(543210, 000000, 055555) | (012345, 000000, 012555)<br>(123450, 000034, 014445)<br>(234510, 000001, 013355)<br>(345210, 002345, 022555)<br>(453210, 000112, 013555)<br>(543210, 011223, 013555) |
| $k = 6$ | (0123456, 0000000, 0666666)<br>(1234560, 0000000, 0666666)<br>(2345610, 0000000, 0666666)<br>(3456210, 0000000, 0666666)<br>(4563210, 0000000, 0666666)<br>(5643210, 0000000, 0666666)<br>(6543210, 0000000, 0666666) | (0123456, 0123456, 0235666)<br>(1023456, 0012345, 0145666)<br>(2103456, 0000012, 0126666)<br>(3210456, 0001123, 0133666)<br>(4321056, 0000234, 0134466)<br>(5432106, 0000000, 0125556)<br>(6543210, 0000001, 0125666) |
| $k = 7$ | (01234567, 00000000, 07777777)<br>(12345670, 00000000, 07777777)<br>(23456710, 00000000, 07777777)<br>(34567210, 00000000, 07777777)<br>(45673210, 00000000, 07777777)<br>(56743210, 00000000, 07777777)<br>(67543210, 00000000, 07777777)<br>(76543210, 00000000, 07777777) | (01234567, 00000000, 01237777)<br>(12345670, 00000012, 01266667)<br>(23456710, 00000001, 01255577)<br>(34567210, 00003456, 01444777)<br>(45673210, 00000123, 01337777)<br>(56743210, 00234567, 02257777)<br>(67543210, 00011234, 01357777)<br>(76543210, 01122345, 01357777) |

***Table A.2:*** *The greedy adaptations of the search schemes based on the pigeonhole principle, for* $8 \leq k \leq 11$*, found by using Hato.*

| | Greedily adapted |
|---|---|
| $k = 8$ | (012345678, 012345678, 023578888)<br>(102345678, 001234567, 014578888)<br>(210345678, 000001234, 012678888)<br>(321045678, 000112345, 013378888)<br>(432105678, 000023456, 013448888)<br>(543210678, 000000012, 012555888)<br>(654321078, 000000123, 012566688)<br>(765432108, 000000000, 012377778)<br>(876543210, 000000001, 012388888) |
| $k = 9$ | (0123456789, 0000000000, 0123499999)<br>(1234567890, 0000000012, 0123888889)<br>(2345678910, 0000000001, 0123777799)<br>(3456789210, 0000001234, 0126666999)<br>(4567893210, 0000000123, 0125559999)<br>(5678943210, 0000345678, 0144499999)<br>(6789543210, 0000012345, 0133799999)<br>(7896543210, 0023456789, 0225799999)<br>(8976543210, 0001123456, 0135799999)<br>(9876543210, 0112234567, 0135799999) |
| $k = 10$ | ( 0 1 2 3 4 5 6 7 8 9 10, 0 1 2 3 4 5 6 7 8 9 10, 0 2 3 5 7 9 10 10 10 10 10)<br>( 1 0 2 3 4 5 6 7 8 9 10, 0 0 1 2 3 4 5 6 7 8 9, 0 1 4 5 7 9 10 10 10 10 10)<br>( 2 1 0 3 4 5 6 7 8 9 10, 0 0 0 0 0 1 2 3 4 5 6, 0 1 2 6 7 9 10 10 10 10 10)<br>( 3 2 1 0 4 5 6 7 8 9 10, 0 0 0 1 1 2 3 4 5 6 7, 0 1 3 3 7 9 10 10 10 10 10)<br>( 4 3 2 1 0 5 6 7 8 9 10, 0 0 0 0 2 3 4 5 6 7 8, 0 1 3 4 4 9 10 10 10 10 10)<br>( 5 4 3 2 1 0 6 7 8 9 10, 0 0 0 0 0 0 1 2 3 4, 0 1 2 5 5 10 10 10 10 10)<br>( 6 5 4 3 2 1 0 7 8 9 10, 0 0 0 0 0 0 1 2 3 4 5, 0 1 2 5 6 6 10 10 10 10)<br>( 7 6 5 4 3 2 1 0 8 9 10, 0 0 0 0 0 0 0 0 1 2, 0 1 2 3 7 7 7 10 10 10)<br>( 8 7 6 5 4 3 2 1 0 9 10, 0 0 0 0 0 0 0 0 1 2 3, 0 1 2 3 8 8 8 8 10 10)<br>( 9 8 7 6 5 4 3 2 1 0 10, 0 0 0 0 0 0 0 0 0 0, 0 1 2 3 4 9 9 9 9 9 10)<br>(10 9 8 7 6 5 4 3 2 1 0, 0 0 0 0 0 0 0 0 0 0 1, 0 1 2 3 4 10 10 10 10 10 10) |
| $k = 11$ | ( 0 1 2 3 4 5 6 7 8 9 10 11, 0 0 0 0 0 0 0 0 0 0 0, 0 1 2 3 4 5 11 11 11 11 11 11)<br>( 1 2 3 4 5 6 7 8 9 10 11 0, 0 0 0 0 0 0 0 0 0 0 1, 0 1 2 3 4 10 10 10 10 10 10 11)<br>( 2 3 4 5 6 7 8 9 10 11 1 0, 0 0 0 0 0 0 0 0 0 0 0, 0 1 2 3 4 9 9 9 9 9 11 11)<br>( 3 4 5 6 7 8 9 10 11 2 1 0, 0 0 0 0 0 0 0 0 1 2 3 4, 0 1 2 3 8 8 8 8 11 11 11)<br>( 4 5 6 7 8 9 10 11 3 2 1 0, 0 0 0 0 0 0 0 0 0 1 2 3, 0 1 2 3 7 7 7 11 11 11 11)<br>( 5 6 7 8 9 10 11 4 3 2 1 0, 0 0 0 0 0 0 1 2 3 4 5 6, 0 1 2 6 6 6 11 11 11 11 11)<br>( 6 7 8 9 10 11 5 4 3 2 1 0, 0 0 0 0 0 0 0 1 2 3 4 5, 0 1 2 5 5 11 11 11 11 11 11)<br>( 7 8 9 10 11 6 5 4 3 2 1 0, 0 0 0 0 3 4 5 6 7 8 9 10, 0 1 4 4 4 9 11 11 11 11 11 11)<br>( 8 9 10 11 7 6 5 4 3 2 1 0, 0 0 0 0 0 1 2 3 4 5 6 7, 0 1 3 3 7 9 11 11 11 11 11 11)<br>( 9 10 11 8 7 6 5 4 3 2 1 0, 0 0 2 3 4 5 6 7 8 9 10 11, 0 2 2 5 7 9 11 11 11 11 11 11)<br>(10 11 9 8 7 6 5 4 3 2 1 0, 0 0 0 1 1 2 3 4 5 6 7 8, 0 1 3 5 7 9 11 11 11 11 11 11)<br>(11 10 9 8 7 6 5 4 3 2 1 0, 0 1 1 2 2 3 4 5 6 7 8 9, 0 1 3 5 7 9 11 11 11 11 11 11) |

***Table A.3:*** *The greedy adaptations of the search schemes based on the pigeonhole principle, for* $12 \leq k \leq 13$*, found by using Hato.*

| | Greedily adapted |
|---|---|
| $k = 12$ | ( 0 1 2 3 4 5 6 7 8 9 10 11 12 , 0 1 2 3 4 5 6 7 8 9 10 11 12 , 0 2 3 5 7 9 11 12 12 12 12 12 12) |
| | ( 1 0 2 3 4 5 6 7 8 9 10 11 12 , 0 0 1 2 3 4 5 6 7 8 9 10 11 , 0 1 4 5 7 9 11 12 12 12 12 12 12) |
| | ( 2 1 0 3 4 5 6 7 8 9 10 11 12 , 0 0 0 0 1 2 3 4 5 6 7 8 , 0 1 2 6 7 9 11 12 12 12 12 12 12) |
| | ( 3 2 1 0 4 5 6 7 8 9 10 11 12 , 0 0 0 1 1 2 3 4 5 6 7 8 9 , 0 1 3 3 7 9 11 12 12 12 12 12 12) |
| | ( 4 3 2 1 0 5 6 7 8 9 10 11 12 , 0 0 0 0 2 3 4 5 6 7 8 9 10 , 0 1 3 4 4 9 11 12 12 12 12 12 12) |
| | ( 5 4 3 2 1 0 6 7 8 9 10 11 12 , 0 0 0 0 0 1 2 3 4 5 6 7 , 0 1 2 5 5 11 12 12 12 12 12 12 12) |
| | ( 6 5 4 3 2 1 0 7 8 9 10 11 12 , 0 0 0 0 0 1 2 3 4 5 6 7 , 0 1 2 5 6 6 12 12 12 12 12 12 12) |
| | ( 7 6 5 4 3 2 1 0 8 9 10 11 12 , 0 0 0 0 0 0 1 2 3 4 , 0 1 2 3 7 7 7 12 12 12 12 12 12) |
| | ( 8 7 6 5 4 3 2 1 0 9 10 11 12 , 0 0 0 0 0 0 1 2 3 4 5 , 0 1 2 3 8 8 8 8 12 12 12 12 12) |
| | ( 9 8 7 6 5 4 3 2 1 0 10 11 12 , 0 0 0 0 0 0 0 1 2 3 4 5 , 0 1 2 3 4 9 9 9 9 9 12 12 12) |
| | (10 9 8 7 6 5 4 3 2 1 0 11 12 , 0 0 0 0 0 0 0 0 1 2 3 4 , 0 1 2 3 4 10 10 10 10 10 10 12 12) |
| | (11 10 9 8 7 6 5 4 3 2 1 0 12 , 0 0 0 0 0 0 0 0 0 1 2 3 , 0 1 2 3 4 5 11 11 11 11 11 11 12) |
| | (12 11 10 9 8 7 6 5 4 3 2 1 0 , 0 0 0 0 0 0 0 0 0 0 1 2 , 0 1 2 3 4 5 12 12 12 12 12 12 12) |
| $k = 13$ | ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 , 0 0 0 0 0 0 0 0 0 0 0 0 , 0 1 2 3 4 5 6 13 13 13 13 13 13) |
| | ( 1 2 3 4 5 6 7 8 9 10 11 12 13 0 , 0 0 0 0 0 0 0 0 0 0 0 1 2 , 0 1 2 3 4 5 12 12 12 12 12 12 13) |
| | ( 2 3 4 5 6 7 8 9 10 11 12 13 1 0 , 0 0 0 0 0 0 0 0 0 0 0 1 , 0 1 2 3 4 5 11 11 11 11 11 13 13) |
| | ( 3 4 5 6 7 8 9 10 11 12 13 2 1 0 , 0 0 0 0 0 0 0 0 0 0 1 2 3 4 , 0 1 2 3 4 10 10 10 10 10 13 13) |
| | ( 4 5 6 7 8 9 10 11 12 13 3 2 1 0 , 0 0 0 0 0 0 0 0 0 1 2 3 , 0 1 2 3 4 9 9 9 9 9 13 13 13) |
| | ( 5 6 7 8 9 10 11 12 13 4 3 2 1 0 , 0 0 0 0 0 0 0 0 1 2 3 4 5 6 , 0 1 2 3 8 8 8 8 8 13 13 13 13) |
| | ( 6 7 8 9 10 11 12 13 5 4 3 2 1 0 , 0 0 0 0 0 0 0 1 2 3 4 5 , 0 1 2 3 7 7 7 7 13 13 13 13 13) |
| | ( 7 8 9 10 11 12 13 6 5 4 3 2 1 0 , 0 0 0 0 0 1 2 3 4 5 6 7 8 , 0 1 2 6 6 6 13 13 13 13 13 13 13) |
| | ( 8 9 10 11 12 13 7 6 5 4 3 2 1 0 , 0 0 0 0 0 0 1 2 3 4 5 6 7 , 0 1 2 5 5 11 13 13 13 13 13 13 13) |
| | ( 9 10 11 12 13 8 7 6 5 4 3 2 1 0 , 0 0 0 0 3 4 5 6 7 8 9 10 11 12 , 0 1 4 4 4 9 11 13 13 13 13 13 13) |
| | (10 11 12 13 9 8 7 6 5 4 3 2 1 0 , 0 0 0 0 1 2 3 4 5 6 7 8 9 , 0 1 3 3 7 9 11 13 13 13 13 13 13) |
| | (11 12 13 10 9 8 7 6 5 4 3 2 1 0 , 0 0 2 3 4 5 6 7 8 9 10 11 12 13 , 0 2 2 5 7 9 11 13 13 13 13 13 13) |
| | (12 13 11 10 9 8 7 6 5 4 3 2 1 0 , 0 0 0 1 1 2 3 4 5 6 7 8 9 10 , 0 1 3 5 7 9 11 13 13 13 13 13 13) |
| | (13 12 11 10 9 8 7 6 5 4 3 2 1 0 , 0 1 1 2 2 3 4 5 6 7 8 9 10 11 , 0 1 3 5 7 9 11 13 13 13 13 13 13) |

***Table A.4:*** *The search schemes based on 01*0 seeds [1, 2] and the greedily adaptions of these schemes found by using Hato for up to $k = 7$. The left column is the original scheme and the right column is the greedily adapted search scheme.*

| | Original | Greedily adapted |
|---|---|---|
| $k = 2$ | (0123, 0000, 0122)<br>(1230, 0000, 0122)<br>(2310, 0000, 0022) | (0123, 0002, 0122)<br>(1230, 0011, 0112)<br>(2310, 0000, 0022) |
| $k = 3$ | (01234, 00000, 01333)<br>(12340, 00000, 01333)<br>(23410, 00000, 01333)<br>(34210, 00000, 00333) | (01234, 00003, 01233)<br>(12340, 00022, 01223)<br>(23410, 00111, 01133)<br>(34210, 00000, 00333) |
| $k = 4$ | (012345, 000000, 014444)<br>(123450, 000000, 014444)<br>(234510, 000000, 014444)<br>(345210, 000000, 014444)<br>(453210, 000000, 004444) | (012345, 000004, 012444)<br>(123450, 000033, 012334)<br>(234510, 000222, 012244)<br>(345210, 001111, 011444)<br>(453210, 000000, 003444) |
| $k = 5$ | (0123456, 0000000, 0155555)<br>(1234560, 0000000, 0155555)<br>(2345610, 0000000, 0155555)<br>(3456210, 0000000, 0155555)<br>(4563210, 0000000, 0155555)<br>(5643210, 0000000, 0055555) | (0123456, 0000004, 0123555)<br>(1234560, 0000045, 0124445)<br>(2345610, 0000333, 0123355)<br>(3456210, 0002222, 0122555)<br>(4563210, 0011111, 0115555)<br>(5643210, 0000000, 0035555) |
| $k = 6$ | (01234567, 00000000, 01666666)<br>(12345670, 00000000, 01666666)<br>(23456710, 00000000, 01666666)<br>(34567210, 00000000, 01666666)<br>(45673210, 00000000, 01666666)<br>(56743210, 00000000, 01666666)<br>(67543210, 00000000, 00666666) | (01234567, 00000005, 01236666)<br>(12345670, 00000044, 01235556)<br>(23456710, 00000456, 01244466)<br>(34567210, 00003333, 01233666)<br>(45673210, 00022222, 01226666)<br>(56743210, 00111111, 01156666)<br>(67543210, 00000000, 00356666) |
| $k = 7$ | (012345678, 000000000, 017777777)<br>(123456780, 000000000, 017777777)<br>(234567810, 000000000, 017777777)<br>(345678210, 000000000, 017777777)<br>(456783210, 000000000, 017777777)<br>(567843210, 000000000, 017777777)<br>(678543210, 000000000, 017777777)<br>(786543210, 000000000, 007777777) | (012345678, 000000004, 012347777)<br>(123456780, 000000056, 012366667)<br>(234567810, 000000445, 012355577)<br>(345678210, 000004567, 012444777)<br>(456783210, 000033333, 012337777)<br>(567843210, 000222222, 012277777)<br>(678543210, 001111111, 011577777)<br>(786543210, 000000000, 003577777) |

***Table A.5:*** *The search schemes by Kucherov et al. [3]*

| | $k = 2$ | $k = 3$ | $k = 4$ |
|---|---|---|---|
| $p = k + 1$ | (012, 000, 022)<br>(210, 000, 012)<br>(102, 001, 012) | (0123, 0000, 0133)<br>(1023, 0011, 0133)<br>(2310, 0000, 0133)<br>(3210, 0011, 0133) | (01234, 00000, 02244)<br>(43210, 00000, 01344)<br>(10234, 00133, 01334)<br>(01234, 00133, 01334)<br>(32410, 00011, 01244)<br>(21034, 00013, 01244)<br>(10234, 00124, 01244)<br>(01234, 00034, 00444) |
| $p = k + 2$ | (0123, 0000, 0112)<br>(3210, 0000, 0122)<br>(1230, 0001, 0012)<br>(0123, 0002, 0022) | (01234, 00000, 01233)<br>(12340, 00000, 01223)<br>(23410, 00001, 01133)<br>(34210, 00012, 00333) | (012345, 000000, 012344)<br>(123450, 000000, 012344)<br>(543210, 000001, 012244)<br>(345210, 000012, 011344)<br>(234510, 000023, 011244)<br>(453210, 000133, 003344)<br>(012345, 000333, 003344)<br>(012345, 000044, 002444)<br>(231045, 000124, 002244)<br>(453210, 000044, 001444) |

***Table A.6:*** *The greedy adaptations of the search schemes by Kucherov et al. with $p = k + 1$ found by Hato.*

| $k = 2$ | $k = 3$ | $k = 4$ |
|---|---|---|
| (012, 012, 022)<br>(102, 001, 012)<br>(210, 000, 012) | (0123, 0000, 0133)<br>(1023, 0111, 0133)<br>(2310, 0002, 0133)<br>(3210, 0113, 0133) | (01234, 00002, 02244)<br>(01234, 00334, 01334)<br>(01234, 00344, 00444)<br>(10234, 01334, 01334)<br>(10234, 01224, 01244)<br>(21034, 00113, 01244)<br>(32410, 00111, 01244)<br>(43210, 00000, 01344) |

***Table A.7:*** *The search schemes by Kianfar et al. for $p = k + 1$ [4], and the manual adaption ("$Man_{Best}$") of the scheme for $k = 4$ [2].*

| $k = 2$ | $k = 3$ | $k = 4$ | $\text{Man}_{\text{Best}}$ |
|---|---|---|---|
| (012, 002, 012)<br>(210, 000, 022)<br>(120, 011, 012) | (0123, 0003, 0233)<br>(1230, 0000, 1233)<br>(2310, 0022, 0033) | (01234, 00004, 03344)<br>(12340, 00000, 22334)<br>(43210, 00033, 00444) | (012345, 000004, 033344)<br>(123450, 000000, 022334)<br>(213450, 011111, 022334)<br>(321450, 012222, 012334)<br>(543210, 000033, 004444) |

***Table A.8:*** *The minU schemes for $k+1$ parts found by Hato. For $k=4$ and $k=6$, two co-optimal variants are given. For all k, symmetric variants can be created by reverse-mapping the $\pi$-strings.*

| | | |
|---|---|---|
| $k=2$ | (012, 011, 022)<br>(102, 000, 012)<br>(210, 002, 012) | |
| $k=3$ | (0123, 0000, 0133)<br>(1023, 0111, 0133)<br>(2310, 0002, 0133)<br>(3210, 0113, 0133) | |
| $k=4$ | (01234, 00222, 02244)<br>(12034, 00000, 01244)<br>(21034, 01111, 01244)<br>(34210, 00003, 01444)<br>(43210, 01114, 01444) | (01234, 01114, 01444)<br>(10234, 00003, 01444)<br>(23410, 01111, 02244)<br>(32410, 00000, 01244)<br>(43210, 00222, 01244) |
| $k=5$ | (012345, 000222, 013555)<br>(102345, 011333, 013555)<br>(231045, 000000, 013355)<br>(321045, 011111, 013355)<br>(453210, 000004, 013555)<br>(543210, 011115, 013555) | |
| $k=6$ | (0123456, 0022226, 0226666)<br>(1203456, 0111115, 0126666)<br>(2103456, 0000004, 0126666)<br>(3456210, 0000000, 0133666)<br>(4356210, 0111111, 0133666)<br>(5643210, 0002222, 0133666)<br>(6543210, 0113333, 0133666) | (0123456, 0111115, 0126666)<br>(1023456, 0000004, 0126666)<br>(2103456, 0022226, 0226666)<br>(3456210, 0002222, 0133666)<br>(4356210, 0113333, 0133666)<br>(5643210, 0000000, 0133666)<br>(6543210, 0111111, 0133666) |
| $k=7$ | (01234567, 00000000, 01337777)<br>(10234567, 01111111, 01337777)<br>(23104567, 00022222, 01337777)<br>(32104567, 01133333, 01337777)<br>(45673210, 00000004, 01337777)<br>(54673210, 01111115, 01337777)<br>(67543210, 00022226, 01337777)<br>(76543210, 01133337, 01337777) | |

# References

[1] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[2] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

[3] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[4] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert. *Optimum Search Schemes for approximate string matching using bidirectional FM-index*. arXiv preprint arXiv:1711.02035, 2017. doi:10.48550/arXiv.1711.02035.

# B

# Supplementary Benchmarks for Chapter 2

Tables B.1- B.5 repeat the benchmark results from Table 2.1 for different search schemes: the search schemes based on the pigeonhole principle (Table B.1); the search schemes by Kucherov et al. [1] with $k$+2 parts (Table B.2); the search schemes based on the 01*0 principle [2] (Table B.3); the search schemes by Kianfar et al. [3] (Table B.4); and the search scheme by Pockrandt et al [4] (Table B.5).

In all cases, dynamic partitioning significantly reduces the search space and the runtime. The only exception is the search scheme by Kianfar et al. for $k = 4$ errors. In that case, dynamic partitioning reduces the search space by only 3.2%. This particular search scheme contains a search that allows 2 errors in the first part to be matched (see Table A.7). For the edit distance metric, this shows to be an inefficient design and dynamic partitioning can only slightly improve upon this. Pockrandt proposed a replacement search scheme for $k = 4$ that appears to work well in practice. For that search scheme, dynamic partitioning reduces the search space by 44.6% (see Table B.5).

Table B.6 shows the benchmark result for the experiment on the reads subsampled from Pacific Biosciences data.

***Table B.1:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes based on the pigeonhole principle.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Uniform | 6.77 ± 0.15 s | | 29 212 674 | | 45 697 288 | |
| Optimal static | 6.77 ± 0.15 s | (−0.0%) | 29 212 674 | (−0.0%) | 45 697 288 | (−0.0%) |
| Dynamic | 7.13 ± 0.04 s | (+5.3%) | 25 829 974 | (−11.6%) | 39 391 011 | (−13.8%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Uniform | 31.85 ± 1.04 s | | 148 267 054 | | 554 152 825 | |
| Optimal static | 31.85 ± 1.04 s | (−0.0%) | 148 267 054 | (−0.0%) | 554 152 825 | (−0.0%) |
| Dynamic | 24.13 ± 0.15 s | (−24.2%) | 104 201 691 | (−29.7%) | 382 688 979 | (−30.9%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Uniform | 197.40 ± 4.32 s | | 955 196 790 | | 4 716 346 658 | |
| Optimal static | 197.40 ± 4.32 s | (−0.0%) | 955 196 790 | (−0.0%) | 4 716 346 658 | (−0.0%) |
| Dynamic | 144.88 ± 0.99 s | (−26.6%) | 672 861 861 | (−29.6%) | 3 497 759 300 | (−25.8%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 970.70 ± 9.70 s | | 4 602 144 092 | | 25 700 861 146 | |
| Optimal static | 970.70 ± 9.70 s | (−0.0%) | 4 602 144 092 | (−0.0%) | 25 700 861 146 | (−0.0%) |
| Dynamic | 722.21 ± 5.72 s | (−25.6%) | 3 272 044 124 | (−28.9%) | 19 799 584 018 | (−23.0%) |

***Table B.2:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes by Kucherov et al. with $k + 2$ parts.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Uniform | 7.99 ± 0.18 s | | 40 359 575 | | 58 563 927 | |
| Optimal static | 6.81 ± 0.06 s | (−14.7%) | 30 800 050 | (−23.7%) | 45 306 870 | (−22.6%) |
| Dynamic | 7.23 ± 0.09 s | (−9.5%) | 26 147 840 | (−35.2%) | 48 835 288 | (−16.6%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Uniform | 24.99 ± 1.60 s | | 112 817 883 | | 339 010 141 | |
| Optimal static | 21.87 ± 0.16 s | (−12.5%) | 107 672 232 | (−4.6%) | 300 126 287 | (−11.5%) |
| Dynamic | 20.93 ± 0.12 s | (−16.2%) | 95 021 936 | (−15.8%) | 281 913 373 | (−16.8%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Uniform | 61.12 ± 0.25 s | | 313 864 359 | | 1 038 322 109 | |
| Optimal static | 60.82 ± 0.43 s | (−0.0%) | 301 001 885 | (−4.1%) | 1 073 595 957 | (+3.4%) |
| Dynamic | 58.61 ± 0.35 s | (−4.1%) | 272 202 234 | (−13.2%) | 1 075 404 827 | (+3.6%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 224.34 ± 4.92 s | | 1 160 723 050 | | 4 542 523 185 | |
| Optimal static | 217.33 ± 2.06 s | (−3.1%) | 1 131 536 993 | (−2.5%) | 4 462 555 385 | (−1.8%) |
| Dynamic | 195.52 ± 1.57 s | (−12.8%) | 1 007 833 149 | (−13.2%) | 4 155 838 510 | (−8.5%) |

***Table B.3:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes based on the 01*0 principle.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Uniform | 7.99 ± 0.21 s | | 40 359 575 | | 58 563 927 | |
| Optimal static | 6.77 ± 0.06 s | (−15.2%) | 30 212 674 | (−25.1%) | 44 987 995 | (+23.2%) |
| Dynamic | 7.22 ± 0.09 s | (−9.6%) | 26 147 840 | (−35.2%) | 48 835 288 | (−16.6%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Uniform | 24.98 ± 0.87 s | | 126 456 147 | | 345 597 744 | |
| Optimal static | 22.61 ± 0.17 s | (−9.3%) | 108 815 630 | (−13.9%) | 319 666 890 | (−7.5%) |
| Dynamic | 18.50 ± 0.10 s | (−25.9%) | 78 708 712 | (−37.8%) | 235 553 233 | (−31.8%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Uniform | 100.02 ± 2.53 s | | 503 926 039 | | 1 954 547 459 | |
| Optimal static | 76.47 ± 0.88 s | (−23.5%) | 382 794 362 | (−24.0%) | 1 518 261 339 | (−22.3%) |
| Dynamic | 68.67 ± 0.38 s | (−31.3%) | 320 725 458 | (−36.4%) | 1 318 567 825 | (−32.5%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 400.60 ± 5.36 s | | 1 951 040 416 | | 9 347 810 879 | |
| Optimal static | 319.76 ± 2.68 s | (−20.2%) | 1 536 196 802 | (−21.3%) | 7 601 863 039 | (−18.7%) |
| Dynamic | 241.17 ± 2.07 s | (−39.8%) | 1 133 408 772 | (−41.9%) | 5 585 934 949 | (−40.2%) |

***Table B.4:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes by Kianfar et al. with $k + 1$ parts.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 959 844, reads mapped 93.6% | | | | | |
| Uniform | 6.97 ± 0.70 s | | 29 212 674 | | 45 697 288 | |
| Optimal static | 6.97 ± 0.70 s | (+0.0%) | 29 212 674 | (+0.0%) | 45 697 288 | (+0.0%) |
| Dynamic | 7.11 ± 0.06 s | (+2.0%) | 25 829 974 | (−11.6%) | 39 391 011 | (−13.8%) |
| | $k = 2$, non-redundant matches = 2 329 746, reads mapped 96.0% | | | | | |
| Uniform | 21.99 ± 1.40 s | | 96 901 001 | | 302 567 180 | |
| Optimal static | 19.20 ± 0.12 s | (−12.7%) | 89 893 258 | (−7.2%) | 261 663 132 | (−13.5%) |
| Dynamic | 17.05 ± 0.11 s | (−22.5%) | 71 232 445 | (−26.5%) | 209 771 958 | (−30.7%) |
| | $k = 3$, non-redundant matches = 4 606 995, reads mapped 97.0% | | | | | |
| Uniform | 191.57 ± 4.76 s | | 1 393 999 288 | | 4 568 071 993 | |
| Optimal static | 156.94 ± 0.77 s | (−18.1%) | 1 151 410 198 | (−17.4%) | 4 223 678 000 | (−7.5%) |
| Dynamic | 152.54 ± 1.19 s | (−20.5%) | 1 132 899 436 | (−18.7%) | 3 679 649 324 | (−19.4%) |
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 2019.33 ± 18.50 s | | 17 701 658 871 | | 83 203 012 638 | |
| Optimal static | 2003.37 ± 17.94 s | (−0.8%) | 17 485 236 184 | (−1.2%) | 86 955 173 979 | (+4.5%) |
| Dynamic | 1994.92 ± 19.69 s | (−1.2%) | 17 142 532 216 | (−3.2%) | 80 898 661 856 | (−2.8%) |

***Table B.5:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Illumina reads in the human reference genome using the $Man_{best}$ search scheme by Pockrandt. This search scheme supports only the case of $k = 4$ errors.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 4$, non-redundant matches = 7 997 221, reads mapped 97.7% | | | | | |
| Uniform | 333.81 ± 2.45 s | | 1 666 160 771 | | 6 728 144 825 | |
| Optimal static | 264.21 ± 1.29 s | (−20.9%) | 1 298 876 393 | (−22.0%) | 5 337 354 757 | (−20.1%) |
| Dynamic | 192.56 ± 1.35 s | (−42.3%) | 922 526 634 | (−44.6%) | 3 917 702 967 | (−41.8%) |

***Table B.6:*** *Comparison of different partitioning strategies when identifying all occurrences of both strands of 100 000 Pacific Biosciences subpatterns ($|P| = 50$) for different values of the maximum allowed edit distance k using the search schemes by Kucherov et al. with $k + 1$ parts.*

| partitioning strategy | wall clock time ± SD | | no. of nodes visited (search space) | | no. of matrix elements computed | |
|---|---|---|---|---|---|---|
| | $k = 1$, non-redundant matches = 151 090, patterns mapped 0.5% | | | | | |
| Uniform | 1.17 ± 0.01 s | | 7 837 976 | | 4 027 509 | |
| Optimal static | 1.16 ± 0.02 s | (−0.9%) | 7 837 976 | (+0.0%) | 4 027 509 | (+0.0%) |
| Dynamic | 1.39 ± 0.00 s | (+18.8%) | 7 112 524 | (−9.3%) | 3 011 346 | (−25.2%) |
| | $k = 2$, non-redundant matches = 306 628, patterns mapped 1.2% | | | | | |
| Uniform | 6.80 ± 1.17 s | | 40 951 027 | | 131 159 309 | |
| Optimal static | 6.20 ± 1.02 s | (−8.8%) | 36 761 149 | (−10.2%) | 101 877 008 | (−22.3%) |
| Dynamic | 5.88 ± 0.88 s | (−13.5%) | 31 889 669 | (−22.1%) | 85 493 365 | (−34.8%) |
| | $k = 3$, non-redundant matches = 742 451, patterns mapped 2.7% | | | | | |
| Uniform | 40.71 ± 1.43 s | | 254 473 999 | | 1 093 410 620 | |
| Optimal static | 40.62 ± 1.23 s | (−0.2%) | 254 473 999 | (+0.0%) | 1 093 410 620 | (+0.0%) |
| Dynamic | 38.23 ± 0.70 s | (−6.1%) | 225 340 512 | (−11.4%) | 902 694 330 | (−17.4%) |
| | $k = 4$, non-redundant matches = 1 916 277, patterns mapped 5.4% | | | | | |
| Uniform | 307.7 ± 3.14 s | | 1 819 713 150 | | 8 877 954 820 | |
| Optimal static | 246.86 ± 2.80 s | (−19.8%) | 1 447 744 722 | (−20.4%) | 6 893 467 627 | (−16.6%) |
| Dynamic | 246.38 ± 2.68 s | (−20.0%) | 1 425 456 865 | (−21.7%) | 6 921 902 062 | (−22.0%) |

# References

[1] G. Kucherov, K. Salikhov, and D. Tsur. *Approximate String Matching Using a Bidirectional Index*. In A. S. Kulikov, S. O. Kuznetsov, and P. Pevzner, editors, Combinatorial Pattern Matching, pages 222–231, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07566-2_23.

[2] C. Vroland, M. Salson, S. Bini, and H. Touzet. *Approximate search of short patterns with high error rates using the 01*0 lossless seeds*. Journal of Discrete Algorithms, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

[3] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert. *Optimum Search Schemes for approximate string matching using bidirectional FM-index*. arXiv preprint arXiv:1711.02035, 2017. doi:10.48550/arXiv.1711.02035.

[4] C. M. Pockrandt. *Approximate String Matching: Improving Data Structures and Algorithms*. PhD thesis, Freie Universität Berlin, 2019. doi:10.17169/refubium-2185.

# C

# Search Space Comparison for a Single Read: Dynamic vs. Static Partitioning

*In this appendix, we focus on the search space of a single read alignment, which demonstrates a significant reduction when applying the dynamic partitioning technique introduced in Chapter 2.*

Let us consider a read on which dynamic partitioning yields a large reduction in search space over the use of (optimal) static partitioning. The read is aligned to the human reference genome with up to $k = 4$ errors (substitutions and/or indels) using the search scheme by Kucherov et al. with $k + 1 = 5$ parts. The alignment of this read using static partitioning explores 541 875 nodes of the search trie, whereas the same task explores only 305 101 nodes using dynamic partitioning, a difference of 236 774 nodes.

The upper half of Figure C.1 shows the static partitioning of this pattern, the length of each part and the number of exact occurrences of each part in the reference genome. Immediately, the low number of 101 occurrences for $\mathcal{P}_3$, as compared to the other parts, stands out. On average, a 19-mer ($\mathcal{P}_3$) would have (many) more exact occurrences than a 27-mer ($\mathcal{P}_0$) or a 20-mer ($\mathcal{P}_1$ and $\mathcal{P}_5$). However, given the complex repeat structure of the human reference genome, a large variability in the number of occurrences can be observed.

The lower half of Figure C.1 illustrates the same read, this time partitioned using dynamic partitioning. With dynamic partitioning, the length of $\mathcal{P}_3$ decreases by 10

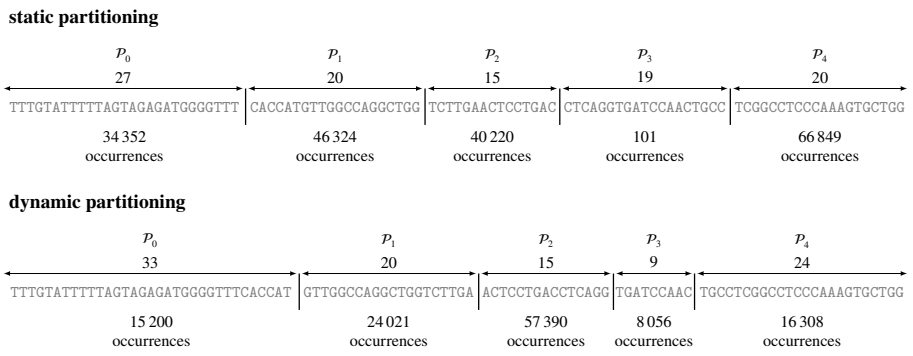


***Figure C.1:*** *Static and dynamic partitioning of a search pattern for $k = 4$ errors using the Kucherov $k$+1 search scheme. The number of exact occurrences, in the human reference genome, of each part are indicated below the read. The length of each part is indicated above the read.*

nucleotides, yet it still has the fewest occurrences (8 056) among all parts. Conversely, $\mathcal{P}_0$ increases in length from 27 to 33 nucleotides.

In the Kucherov $k + 1$ search scheme, 3 out of 8 searches begin with $\mathcal{P}_0$ (see Table A.5). This results in a higher weight being assigned to $\mathcal{P}_0$ during dynamic partitioning, explaining its increased length. This correlation between the number of searches starting with a part and its weight is crucial: parts that initiate more searches generate more nodes during the search process, amplifying their overall importance.

Interestingly, $\mathcal{P}_1$ and $\mathcal{P}_2$ (through sheer coincidence) retain their lengths. However, the number of exact occurrences of these parts of equal size differs greatly as compared with static partitioning, highlighting the non-uniform distribution of the human genome. Only $\mathcal{P}_2$ and $\mathcal{P}_3$ see an increase in the number of exact occurrences with dynamic partitioning compared to static partitioning. However, since only one search begins with these parts, the expected impact of this increase in size on the number of explored nodes is minimal.

Notably, $\mathcal{P}_0$ and $\mathcal{P}_1$ together make up more than half of the pattern's total length, despite there being five parts. This is because 5 out of 8 searches start with these two parts, resulting in greater weight being allocated to them during partitioning.

Table C.1 summarizes the number of explored nodes during the approximate pattern matching phase for each search, comparing dynamic and static partitioning, along with the differences between the two strategies.

Searches that begin with an exact match of $\mathcal{P}_0$ ($\mathcal{S}_0$, $\mathcal{S}_3$, and $\mathcal{S}_7$) explore significantly fewer nodes with dynamic partitioning compared to static partitioning. This reduction is primarily due to the increased size of $\mathcal{P}_0$, which results in less than half the number of exact occurrences for this part after dynamic partitioning. The larger relative decrease in nodes visited for $\mathcal{S}_7$ stems from this search starting with an exact match of $\mathcal{P}_0$ and $\mathcal{P}_1$ combined. This combination occurs 4 114 times with static partitioning but only 1 126 times with dynamic partitioning—a decrease of nearly 75%.

Similarly, searches starting with an exact match of $\mathcal{P}_1$ ($\mathcal{S}_2$ and $\mathcal{S}_6$) also show a reduction in nodes explored. Search $\mathcal{S}_5$, which begins with an exact match of $\mathcal{P}_2$,

***Table C.1:*** *The number of explored nodes in the search trie during the approximate matching phase after static and dynamic partitioning, and the difference in explored nodes between these two partitioning strategies.*

| | Number of explored Nodes | | | |
|---|---|---|---|---|
| Search | Static | Dynamic | Difference | |
| $\mathcal{S}_0$ = (01234, 00000, 02244) | 101 656 | 52 200 | -49 156 | (−48.4%) |
| $\mathcal{S}_1$ = (43210, 00000, 01344) | 19 953 | 37 529 | 17 576 | (+88.1%) |
| $\mathcal{S}_2$ = (10234, 00133, 01334) | 108 528 | 42 739 | -65 789 | (−60.6%) |
| $\mathcal{S}_3$ = (01234, 00133, 01334) | 88 911 | 32 999 | -55 912 | (−62.9%) |
| $\mathcal{S}_4$ = (32410, 00011, 01244) | 1 671 | 6 835 | 5 164 | (+309.0%) |
| $\mathcal{S}_5$ = (21034, 00013, 01244) | 115 485 | 86 153 | -29 332 | (−25.4%) |
| $\mathcal{S}_6$ = (10234, 00124, 01244) | 73 719 | 37 535 | -36 184 | (−49.1%) |
| $\mathcal{S}_7$ = (01234, 00034, 00444) | 31 952 | 8 811 | -23 141 | (−72.4%) |
| Total | 541 875 | 305 101 | -236 774 | (−43.7%) |

experiences a decrease in explored nodes despite $\mathcal{P}_2$ having more exact occurrences with dynamic partitioning. This decrease is likely due to the longer combined lengths of $\mathcal{P}_0$ and $\mathcal{P}_1$, which the search extends to first, reducing the search space.

In contrast, the two remaining searches ($\mathcal{S}_1$ and $\mathcal{S}_4$) show an increase in explored nodes with dynamic partitioning. Both are influenced by $\mathcal{P}_3$, which experiences the largest change in length between the partitioning strategies. Search $\mathcal{S}_4$, starting with an exact match of $\mathcal{P}_3$, explores three times as many nodes with dynamic partitioning due to the 80-fold increase in occurrences of $\mathcal{P}_3$. However, since $\mathcal{P}_3$ still has the fewest occurrences among all parts, the overall impact of $\mathcal{S}_4$ on the total search space is limited.

Search $\mathcal{S}_1$, which begins with an exact match of $\mathcal{P}_4$, sees an increase in explored nodes because $\mathcal{P}_3$, the first extension target, is smaller after dynamic partitioning. Notably, $\mathcal{P}_4$ is longer and has 75% fewer exact occurrences with dynamic partitioning. These two searches ($\mathcal{S}_1$ and $\mathcal{S}_4$) exhibit the smallest absolute differences in explored nodes between the two partitioning strategies. Moreover, the combined increase in nodes for these searches is smaller than the absolute decrease in any other search.

This analysis highlights how dynamic partitioning can substantially reduce the search space by leveraging the non-uniform distribution of the reference genome. This effect is particularly pronounced for reads with many approximate occurrences, demonstrating the efficiency of dynamic partitioning in such cases.

# D

# Supplementary Material for Chapter 5

## D.1 Commands Used for the Different Tools

***Table D.1:*** *Commands (except I/O) used in the alignment benchmarks. $E$ stands for the maximal error rate and $L$ stands for the average length of the reads (150 bp for the experiment on the HRG and 250 for the experiment on the listeria monocytogenes pan-genome.*

| **Tool** | **Command** |
|---|---|
| Yara | `./yara_mapper -y full -t 1 -e [E] -sa record` |
| RazerS3 | `./razers3 -i [100 - E] -m 99999 -dr 0` |
| BWA-aln | `./bwa aln -N -n [` $1 - \sum_{i=0}^{\left\lfloor \frac{L \cdot E}{100} \right\rfloor - 1} \frac{e^{-\lambda} \cdot \lambda^i}{i!}$ `]` with, $e = 0.02$, or `./bwa aln -N -n 0` if $E = 0$, followed by `./bwa samse` |
| Bowtie | `./bowtie -a -best -strata -v [E·L]` |
| Columba | `./columba -I [100-E]` |
| BWA-MEM | `./bwa mem -L 10000` |
| Bowtie2 | `./bowtie2` |
| Bowtie2 (Very Sensitive) | `./bowtie2 -very-sensitive` |
| Ropebwt3 | `./ropebwt3 sw -t1` |

## D.2 Used Genomes for the Bacterial Pan-Genome Benchmarks

A list of the used genomes, with their Reference Sequence ID and Name is available at: https://doi.org/10.5281/zenodo.14277604.

## D.3 Drop-in Nature of Columba in OptiType Script

See figure D.1 on the next page.

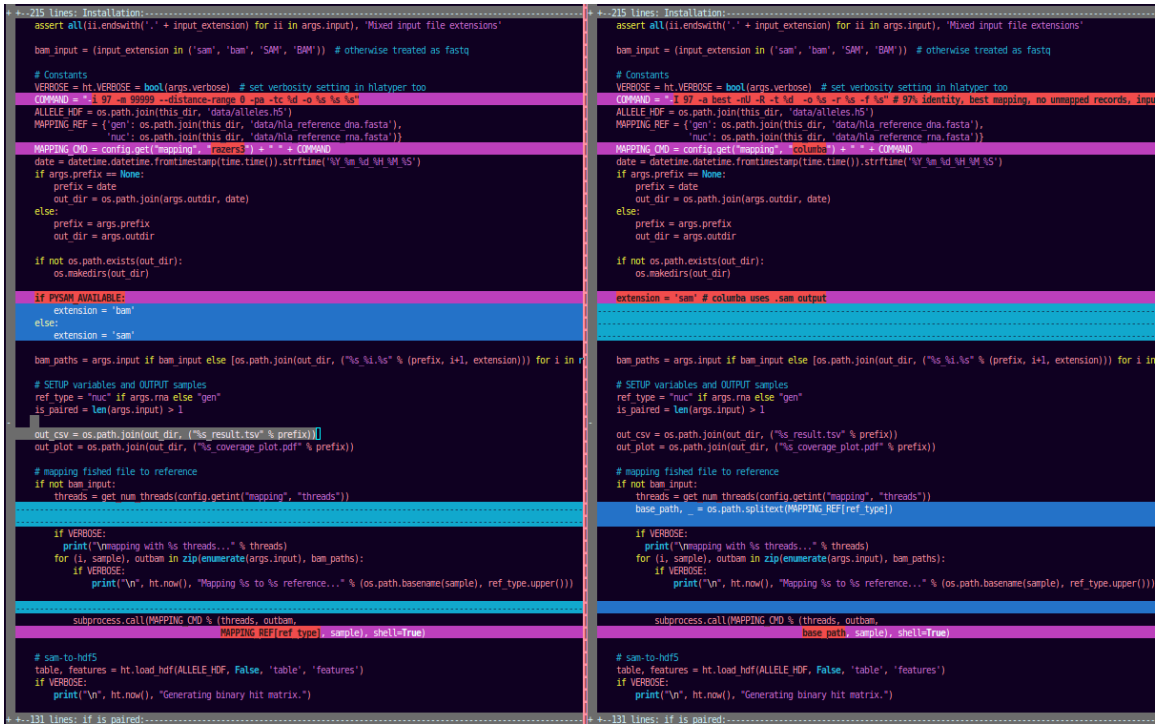

***Figure D.1:*** *Illustration of Columba's drop-in compatibility in the OptiType script. The figure shows a side-by-side comparison of the two OptiType script versions illustrating the drop-in nature of replacing the RazerS3 mapping tool with columba for HLA typing. The left panel shows the original script utilizing RazerS3 with specific command-line parameters for mapping, while the right panel demonstrates the modified version. This visual highlights the minimal modifications required to adapt existing workflows for Columba, showcasing its compatibility and flexibility in the pipeline.*