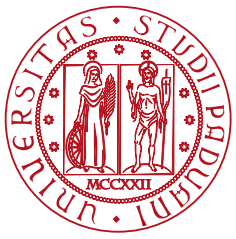


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Improving Spaced k-mer Extraction and Hash Encoding for Bioinformatics Applications

Candidate

Leonardo Gemin

Student ID 2023860

Supervisor

Prof. Matteo Comin

University of Padova

Academic Year 2023/2024 – July 11

Abstract

This thesis focuses on improving the extraction and hash encoding of spaced *k*-mers for bioinformatics applications. It explores the concept of spaced seeds, which improve similarity detection by allowing nonconsecutive matches within *k*-mers, albeit at the expense of increased computational complexity.

The main goal of this research is to develop advanced software capable of rapid forward and reverse complement hashing for spaced *k*-mer in nucleotide sequences. This includes optimizing the hashing process to better handle large genomic datasets and minimize processing time and computational resources. The work includes the introduction of the DuoHash tool, an improved version of Multiple Iterative Spaced Seed Hashing (MISSH), and we compare its performance with ntHash2. Results demonstrate how DuoHash performs on different datasets, showing its time efficiency and integrability with tools such as JellyFish. Finally, practical implications and suggestions for future research directions are discussed.

Sommario

Questa tesi si concentra sul miglioramento dell'estrazione e della codifica hash di *k*-mer spaziati per applicazioni bioinformatiche. Esplora il concetto di semi spaziati, che migliorano il rilevamento della somiglianza consentendo corrispondenze non consecutive all'interno dei *k*-mer, anche se a spese di una maggiore complessità computazionale.

Lo scopo principale di questa ricerca è sviluppare un software avanzato in grado di eseguire rapidamente l'hashing e l'hashing del complemento inverso per i *k*-mer spaziati nelle sequenze nucleotidiche. Ciò include l'ottimizzazione del processo di hashing per gestire meglio grandi insiemi di dati genomici e minimizzare il tempo di elaborazione e le risorse computazionali. Il lavoro include l'introduzione dello strumento DuoHash, una versione migliorata di MISSH e ne confrontiamo le prestazioni con ntHash2. I risultati dimostrano come DuoHash si comporta su diversi set di dati, mostrando la sua efficienza in termini di tempo e l'integrabilità con strumenti come JellyFish. Infine, vengono discusse le implicazioni pratiche e i suggerimenti per le future direzioni di ricerca.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ddNTPs Dideoxynucleotides

DNA Deoxyribonucleic acid

FNV Fowler-Noll-Vo

FSH Fast Spaced Seed Hashing

ISSH Iterative Spaced Seed Hashing

MISSH Multiple Iterative Spaced Seed Hashing

NGS Next-Generation Sequencing

PacBio Pacific Biosciences

RNA Ribonucleic acid

SBS Sequencing-By-Synthesis

SIMD Single Instruction Multiple Data

SMRT Single Molecule Real-Time

1

Introduction

Bioinformatics has changed the way we understand and analyze biological sequences, thereby opening up new vistas in scientific research and practical applications. The sequence classification which is a problem highly essential in this area has numerous applications that range from phylogenetic reconstruction to protein classification, mapping metagenomic reads to oligonucleotide design.

However, alignment as a widespread technique of sequence classification has certain limitations in terms of handling large data sets produced by modern sequencing technologies. Alignment-free approaches were introduced for that purpose and they are primarily based on splitting sequences into consecutive k-mer subsequences and indexing them with appropriate data structures. This allowed much faster processing, but also decreased sensitivity because exact matches for every position of a k-mer were required.

To overcome this limitation, there have been variations of the exact matches of the k-mers such as allowing longer matches with errors or non-consecutive matches within the k-mer itself. In this regard was a significant breakthrough when "spaced seeds," fixed-length patterns that allow wildcards at specific positions were introduced. Seeds like these have been shown to be able to significantly improve the ability to detect relevant similarities between different sequences thus enabling more efficient algorithms to be applied in various areas of bioinformatics.

Nevertheless, spaced seeds tend to result in substantial delays during execution time compared to k-mer-based solutions due to extra complexities involved in in-

dexing/hashing sequences. Consequently, efforts have been directed at improving hashing methods for spaced seeds so as to enhance performance without affecting sensitivity.

1.1 Purpose of the thesis

The main objective of this thesis is to develop advanced software that efficiently computes forward hashing and reverse complement hashing for spaced *k*-mer in nucleotide sequences. This software is designed to improve the handling and analysis of the large amounts of genetic data generated by modern DNA sequencing techniques. Another crucial goal is to optimise the speed and computational efficiency of the hashing process. Considering the increasing size of genomic datasets, it is crucial to have tools that can perform fast analyses without compromising accuracy. The software developed will have to be able to handle these volumes of data efficiently, minimising the processing time and computational resources required. In addition, the software will have to be tested on various datasets to evaluate its performance in comparison to existing tools. This includes a detailed analysis of execution times and a comparison with other hashing techniques currently in use.

1.2 Organization of the work

The structure of the thesis is organised into the following chapters:

Chapter 2: Spaced k-mer. In this chapter, DNA concepts, sequencing methods and assembly techniques are introduced. We introduce *k*-mer and spaced *k*-mer, their applications in bioinformatics, including advantages and disadvantages.

Chapter 3: Hashing of spaced seeds. In this chapter, various hashing methods for spaced *k*-mer are described, including the tools Fast Spaced Seed Hashing (FSH), Iterative Spaced Seed Hashing (ISSH) and MISSH, ntHash and ntHash2.

Chapter 4: A new version of our tool. This chapter introduces the DuoHash tool, the new version of MISSH, and discusses the new features introduced.

Chapter 5: Results. In this chapter, the temporal performance of the new tool is analysed across various datasets and experimental configurations, compar-

ing the performance of DuoHash with ntHash2. The performance of DuoHash and its integration with JellyFish compared to third-party tools such as MaskJelly are also analysed.

Chapter 6: Conclusions. This chapter summarises the results obtained, discussing the practical implications and proposing future directions for research.

Appendices. Additional tables complete with times and speed-ups, and graphs, comparing DuoHash with ntHash2 and integrating DuoHash with JellyFish are provided.

This organisation aims to guide the reader through a thorough understanding of the problem, the proposed solutions and their experimental evaluations, culminating in a summary of conclusions and potential future research directions.

2

Spaced k-mer

2.1 DNA

The essence of life lies in the intricate dance of DNA, or deoxyribonucleic acid, a molecule located in the nucleus of every cell. This molecule is, in fact, a macromolecule and is easily identified by its characteristic double-helix shape consisting of two nucleotide chains. Each nucleotide that makes up the chain consists of a sugar-phosphate molecule and a nitrogenous base. We recognise four nitrogenous bases, Adenine, Cytosine, Guanine and Thymine, which bind two by two through hydrogen bonds forming specific pairs, as illustrated in Figure 2.1: Adenine and Thymine (A-T), Cytosine and Guanine (C-G). The order in which the nitrogenous bases follow one another along the nucleotide chain orchestrates the symphony of existence.

2.1.1 DNA sequencing

DNA sequencing is the process of determining the nucleotide sequence of a DNA fragment, and is fundamental to genetic research, molecular biology, medicine and other disciplines. Numerous technologies have been developed over the years to make this process faster, more accurate and more accessible. Costs have also fallen dramatically (Figure 2.2 and Figure 2.3): from the Human Genome Project, which required billions of dollars, we have moved on to technologies that allow the sequencing of an entire human genome for less than $1,000. The cost, of course, varies depending on the technology used, the coverage required and the complexity of the project [38]. NGS platforms such as Illumina and Ion Torrent offer inexpensive op-

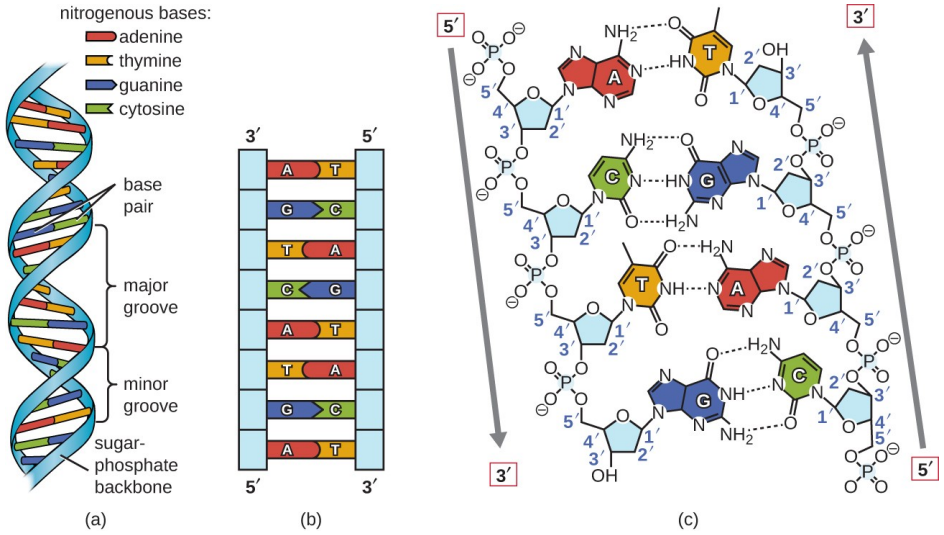


Figure 2.1: The fundamental structure of the DNA double helix consists of two strands, each composed of chains of nucleotides. Within this framework, every nucleotide forms a bond with its complementary counterpart on the opposing strand [28].

tions for large-scale projects, while technologies such as PacBio, while more expensive, provide unique details for specific needs. The main sequencing techniques are described below.

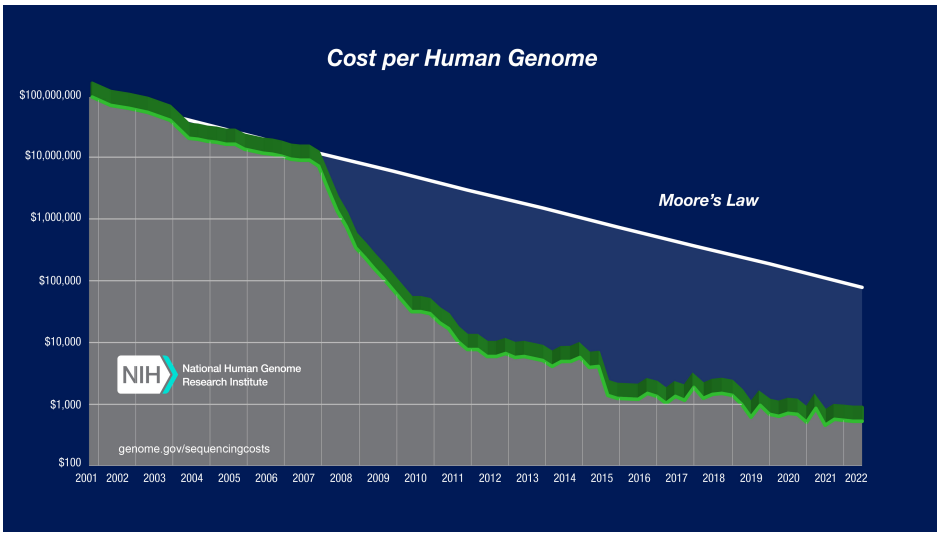


Figure 2.2: Sequencing cost per genome data - 2022: the cost per genome has become much lower than Moore's Law [38].

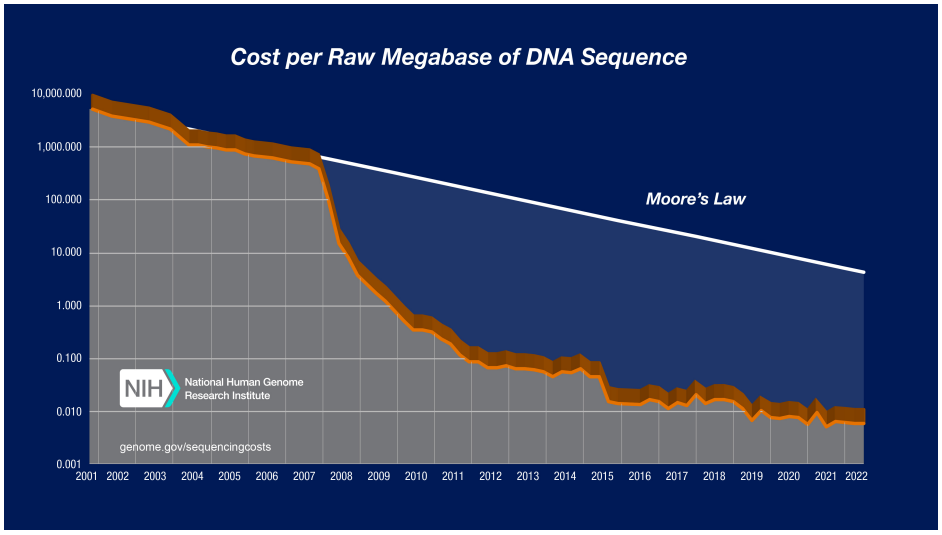


Figure 2.3: Sequencing cost per megabase - 2022: the cost per megabase has become much lower than Moore's Law [38].

Sanger Method

The Sanger method [1], developed by Frederick Sanger in the 1970s, was the first successful approach to DNA sequencing. It utilises chain termination, where Dideoxynucleotides (ddNTPs) interrupt DNA synthesis, allowing the sequence to be read according to the length of the fragments obtained. Although accurate, the Sanger method is relatively slow and expensive, suitable mainly for shorter DNA sequences. This method was used for the Human Genome Project, which sequenced the entire human genome at a cost of approximately $2.7 billion.

Next-Generation Sequencing

In recent decades, Next-Generation Sequencing (NGS) technologies [11, 40] have revolutionised the field, enabling massive, parallel sequencing of billions of DNA fragments. Among the NGS platforms, Illumina, Ion Torrent and PacBio stand out.

Illumina is one of the leaders in the NGS market. It uses Sequencing-By-Synthesis (SBS) technology, where fluorescently labelled nucleotides are incorporated into DNA, and fluorescent images reveal the sequence. This method offers high accuracy and read depth, making it ideal for large-scale genomics projects. The cost to sequence a complete human genome with Illumina can range from a few hundred to a few thousand dollars, depending on the coverage and specifications of the project.

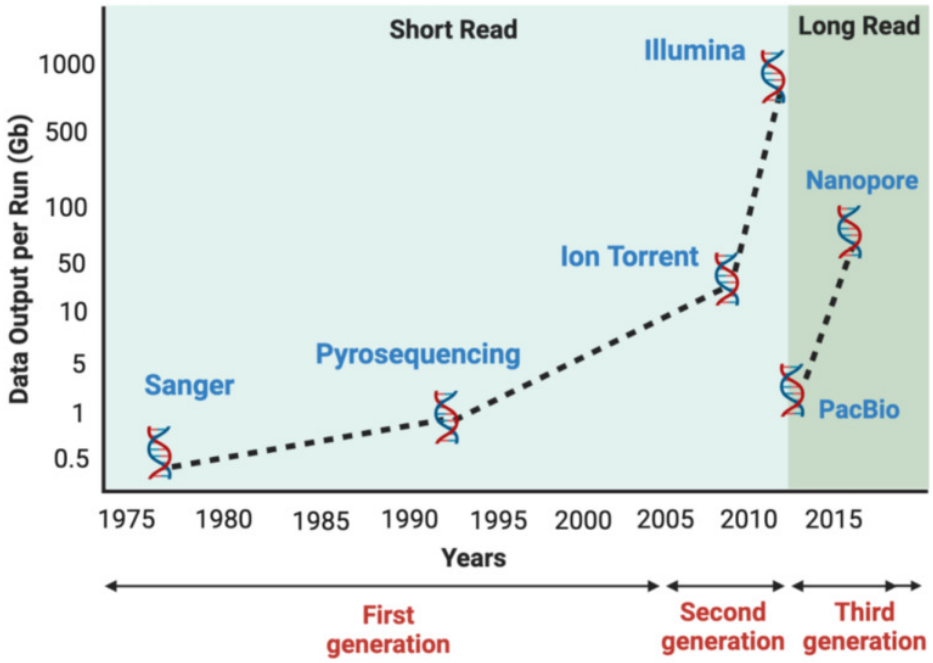


Figure 2.4: Evolution of sequencing technologies [40].

The **Ion Torrent** technology, developed by Life Technologies, is based on the detection of hydrogen ion release during nucleotide incorporation. This fluorescence-free approach allows rapid and low-cost sequencing. Although accuracy may be lower than Illumina, Ion Torrent is advantageous for applications requiring speed and low cost.

Pacific Biosciences (PacBio) has introduced Single Molecule Real-Time (SMRT) Sequencing technology. It offers the ability to read long DNA sequences, with reads that can exceed 10,000 bases. This is particularly useful for the assembly of genomes, but also for the analysis of complex repetitive regions. Despite its higher costs compared to other NGS technologies, PacBio is useful for studies requiring long and detailed reads.

2.1.2 Assembly Techniques

DNA sequencing produces fragments, called *reads*, that must be assembled to reconstruct the original sequence. These are the two main assembly strategies:

- **de novo assembly** [21] is used when no reference sequence is available. Frag-

ments are assembled based only on the overlaps between them. This technique is essential for sequencing new genomes.

- In **reference assembly**, mainly used for genetic variability studies, fragments are aligned to an existing reference sequence. This facilitates the assembly process and improves accuracy.

2.2 k-mer

k-mer are sequences of nucleotides of length *k*, resulting from the decomposition of a longer genomic sequence.

Example

Considering the DNA sequence

CTTGTCGTTGACT,

its 6-mer are:

CTTGTC TTGTCG TGTCGT GTCGTT
TCGTTG CGTTGA GTTGAC TTGACT

The number of *k*-mer in a sequence is governed by the following equation

$$\#(k\text{-mer}) = \ell - k + 1,$$

where ℓ is the length of the sequence.

In bioinformatics, *k*-mer have become one of the most powerful versatile tools that are applied in various genomic studies. Also, they are instrumental in genome assembly, metagenomics, gene expression analysis, pattern and motif recognition as well as phylogenetic analysis thus proving their importance and value in scientific research. These applications confirm how fundamental manipulation and analysis of these patterns are to our understanding of biological processes and genetic dynamics. The choice of value of *k* is important because it affects the sensitivity and specificity of the analyses: very small values may lead to a lot of redundancy and

low specificity making it hard to differentiate very similar sequences; on the other hand large values improve specificity but at the same time reduce coverage which increases computation complexity. In practice, the optimal k value depends on what is being analyzed and its nature. For example, in de novo assembly of genomes, k values between 21 and 31 are commonly used since they optimally combine high sequence coverage with specificities for many purposes.

2.2.1 Applications in Bioinformatics

Genome assembly is one of the most relevant applications. In this context, genome reconstruction from short fragments using k-mer, a process known as *de novo* assembly [21]. During this process, k-mer facilitate the overlapping and connection of sequence fragments, forming structures called *contigs* and *scaffolds*.

In metagenomics [24], k-mer are used to identify and quantify the presence of different microbial species in environmental samples. This is possible by comparing the k-mer derived from the samples with known sequence databases, thus allowing the taxonomic composition of the sample to be determined. The analysis of k-mer in metagenomics facilitates study complex microbial communities contributing to understanding biodiversity and ecological dynamics within diverse environments.

Another significant application for which k-mer are useful is gene expression analysis [26]: RNA-seq techniques create k-mer from transcripts that can be mapped on reference genomes to identify and count expressed genes. These enable precise measurements of gene expression levels required for studies on gene function, cellular responses as well as disease.

k-mer is also used to detect repetitive patterns and motifs in DNA and RNA sequences [3]. Identifying these recurring patterns, as well as functional motifs, is fundamental to understanding gene regulation and protein functions. Notably, motifs are particular sequences that are vital for regulatory protein binding sites. Researchers can also find and study such hidden regulatory elements within genomic sequences using k-mer.

Lastly, k-mer helps in building phylogenetic trees based on the similarity of genomic sequences during phylogenetic analysis [30]. This method is quite fast, especially when dealing with large-scale evolutionary studies using huge genomic datasets. Comparing k-mer across different species can help to reconstruct evolutionary rela-

tionships and give insights into the past and diversity of life on Earth.

2.2.2 Benefits and Disadvantages

The use of *k*-mer in bioinformatics offers numerous advantages, making them valuable tools in multiple genomic analyses. First of all, *k*-mer allow great computational efficiency. Indeed, the decomposition of long sequences into blocks of a fixed size facilitates the indexing and searching of sub-sequences, accelerating complex processes such as sequence comparison and assembly [16, 14]. This feature is particularly advantageous in the era of big data, where speed of processing is crucial. Furthermore, *k*-mer processing can be easily parallelised. This means that computational tasks can be divided among several processors or cores, significantly improving the performance of bioinformatics software on modern architectures, such as supercomputers and GPUs. Another important advantage is the reduction in complexity that *k*-mer can offer: by representing complex genomic sequences in terms of *k*-mer, subsequent calculations can be simplified, making it easier to compare and assemble sequences [22]. This approach helps to manage and interpret complex genomic data, making intricate analyses more accessible.

However, the use of *k*-mer also has some disadvantages. One of the main ones concerns the choice of the *k*-value, which can be critical for the success of the analysis. A sub-optimal *k*-value may in fact compromise results by increasing the number of false positives or negatives. Small values of *k* can lead to greater redundancy and lower specificity, while large values can reduce sequence coverage and increase computational complexity. Therefore, the selection of the value of *k* requires a careful balance between specificity and coverage, adapted to the specific application. The storage of *k*-mer represents another disadvantage, as it can require a considerable amount of memory, especially when working with large genomes or many samples. Efficient memory management is therefore essential to avoid performance problems and to ensure that computational resources are optimally utilised [17, 33]. Finally, *k*-mer can be sensitive to noise in input sequences [20]. Read errors or mutations can generate unique *k*-mer that do not correctly represent the original sequence, negatively affecting the accuracy of analyses. This sensitivity requires the implementation of filtering and error correction strategies to ensure that the data used are as accurate and representative as possible.

2.3 Spaced k-mer

Spaced seeds — and, consequently, spaced k-mer — represent fundamental tools in bioinformatics, particularly in the context of sequence alignment and the search for similarities between DNA, RNA and proteins [25, 31, 32]. Unlike traditional k-mer, spaced seeds allow for the introduction of gaps (ignored positions) within the nucleotide or amino acid sequence, thus allowing for a more flexible comparison of sequences. In formal terms, a spaced seed Q is a string on the alphabet $\{0,1\}$, where the `1` correspond to the matching positions:

$$Q = \{x \mid x \in \{0,1\}^*,\ \#_1(x) = w\}\,,$$

where k is the *length*, or *span*, and w is the *weight* of the spaced seed. Since any position `0` placed before the first `1` and any position `0` placed after the last `1` does not change the spaced seed, we only consider spaced seeds starting and ending with the character `1`, defined by the following regular expression:

$$Q = \{x \mid x = 1 + 1 \cdot (0+1)^* \cdot 1,\ \#_1(x) = w\}\,. \tag{2.1}$$

One can also represent the spaced seeds by their *shape* Q, which is the set of positions of the `1` in the spaced seed [4]. In this case the weight of Q is defined as $k = |Q|$, while the length, or span, is equal to $s(Q) = \max(Q) + 1$.

Spaced k-mer, also called Q-gram, are fragments of a nucleotide sequence x that respect the pattern dictated by a spaced seed Q. Given a string x, the spaced k-mer $x[i+Q]$ is a string defined as follows:

$$x[i+Q] = \{x_{i+k} \mid k \in Q\}\,,$$

where $i \in \{0,1,\ldots,|x| - s(Q)\}$.

Example

Let us consider the spaced seed `111010101`, defined as $Q = \{0,1,2,4,6,8\}$, and the sequence seen in the previous section,

`CTTGTCGTTGACT`.

Then the Q-gram at position 0 of x is defined as

x	C	T	T	G	T	C	G	T	T	G	[...]
Q	1	1	1	0	1	0	1	0	1		
$x[0+Q]$	C	T	T		T		G		T		

The other Q-grams, in total there are $|x| - s(Q) + 1 = 5$ Q-grams, are:

$$x[1+Q] = \texttt{TTGCTG} \qquad x[2+Q] = \texttt{TGTGTA}$$
$$x[3+Q] = \texttt{GTCTGC} \qquad x[4+Q] = \texttt{TCGTAT}$$

2.3.1 Applications in Bioinformatics and Benefits

One of the greatest advantages is that spaced seeds can enhance sequence alignment sensitivity without much loss in specificity, which can be handy when a sequence may have local genetic variation like small mutations or indels. Furthermore, spaced seeds also detect homology regions with discontinuities or variations that would elude k-mer contiguous approaches [9]. However, due to the handling of more complex gaps such as: those that appear in between and optimizing spaced seeds design; this algorithm is generally computationally intensive than its counterpart for work spacing seed contiguously on k-mer. In fact, it is very difficult to compute correspondences and solve them while designing and optimizing spaced seeds because there might be many gaps between them [19]. But these problems are outweighed by significant increase in sensitivity and accuracy resulting from the use of spaced seed.

Spaced seeds are commonly used across various bioinformatics disciplines. For instance, they are employed for enhancing sequence alignment algorithms such as BLAST and its variants where they have been shown to boost both sensitivity as well as speed during search operations [2, 5]. The introduction of spaced seeds into BLAST improved the detection of homologous regions in biological sequences thus reducing false positives and improving overall efficiency of alignment process [12].

Among other things, algorithms like PatternHunter utilize spaced seed to make comparison between genomic sequences from different species more efficient by minimizing false positives compared to contiguous methods based on k-mer [5].

This approach is particularly useful when comparing complex genomes that often contain local variations masking global similarities. Moreover alternative methods cannot detect remote homologies unlike these based on spaced seed [7].

In order to create more robust De Bruijn graphs for de novo genome assembly, spaced k-mer can be used which can help identify correct contigs despite presence of sequencing errors [15]. Spaced seeds allow for more continuous and complete genomic assemblies with fewer artifacts due to systematic and random reading errors in sequences [18].

Moreover, within metagenomics, the use of spaced seed improves microbial community analyses, making it easier to differentiate close species better and identify new genomic variants [8]. This is especially important when dealing with complex systems such as soil or human gut, where high microbial diversity requires fine species resolution necessary to model their ecological and functional dynamics [10].

In summary, spaced seeds are a major breakthrough in bioinformatics that achieves a tradeoff between sensitivity and specificity in various applications. Although they may need more computational resources to achieve this goal, the accuracy improvement in sequence analysis is enough reason to use them.

3

Hashing of spaced seeds

3.1 FSH: Fast Spaced Seed Hashing

The FSH [34] is an algorithm developed to increase the efficiency of hash calculation for spaced seeds in bioinformatics applications. This approach exploits the similarities between spaced seed hash values computed at neighbouring positions within an input sequence through a dynamic programming technique aimed at reducing the number of symbols read and encoded in the hash computation process. This results in a higher processing speed than traditional methods.

Symbol encoding, a process of transforming data into a different format using a specific encoding scheme, is essential for the numerical representation of DNA or protein sequences. The encoding used by FSH is based on the function

$$encode(\mathtt{ch}) : \mathcal{A} \rightarrow \{0,1\}^{\log_2 |\mathcal{A}|}$$

and the four nitrogenous bases are mapped as follows:

$$encode(\mathtt{A}) = 00 \qquad encode(\mathtt{C}) = 01$$
$$encode(\mathtt{G}) = 10 \qquad encode(\mathtt{T}) = 11$$

This encoding is necessary for the subsequent application of hashing functions, as it transforms the sequence data into a numerical format suitable for computational processing.

In the paper of FSH, the authors considered the Rabin-Karp rolling hash [13], defined as follows:

$$h\left(x[i+Q]\right) = \bigvee_{k \in Q} \left(encode(x_{i+k}) << m(k) \cdot \log_2 |\mathcal{A}|\right),$$

where $m(k) = |\{i \in Q \mid i < k\}|$ is the number of shifts that must be applied to the k-th symbol.

Example

In connection with the example from the previous chapter, the steps for calculating the encoding of the Q-gram $x[0+Q]$ are given:

x	C	T	T	G	T	C	G	T	T	G	[...]
Q	1	1	1	0	1	0	1	0	1		
m	0	1	2	3	3	4	4	5	5		
$x[0+Q]$	C	T	T		T		G		T		
$encode(x[0+Q])$	01	11	11		11		10		11		

And thus for the calculation of the hashing value referred to it:

$$\begin{aligned} h\left(x[0+Q]\right) &= (01 << 0) \vee (11 << 2) \vee (10 << 4) \vee \\ &\quad \vee (01 << 6) \vee (10 << 8) \vee (01 << 10) \\ &= 111011111101 \end{aligned}$$

Similarly, the hashing values for the remaining Q-grams are:

$$\begin{aligned} x[1+Q] &= \texttt{TTGCTG} & h(x[1+Q]) &= 101101101111 \\ x[2+Q] &= \texttt{TGTGTA} & h(x[2+Q]) &= 001110111011 \\ x[3+Q] &= \texttt{GTCTGC} & h(x[3+Q]) &= 011011011110 \\ x[4+Q] &= \texttt{TCGTAT} & h(x[4+Q]) &= 110011100111 \end{aligned}$$

It is now possible to define the set of hashing values of a string x given a spaced

seed Q:

$$\mathcal{H}(x, Q) = \{h(x[0 + Q]),\ h(x[1 + Q]),\ \ldots,\ h(x[n - 1 + Q])\}\,,$$

where $n = |x| - s(Q) + 1$ is the number of all Q-grams of x.

According to the authors of FSH, the aim is to minimise the number of times a symbol needs to be read and encoded in order to calculate $\mathcal{H}(x, Q)$.

The idea is to reuse part of the previous hashes to speed up the calculation of the new value. A new definition can be introduced:

$$\mathcal{C}_j = \{k \in Q \mid k + j \in Q \wedge m(k) = m(k + j) - m(j)\}.$$

$\mathcal{C}_j$ is the set of positions that can be retrieved from the previously calculated $h(x[i - j + Q])$ when $h(x[i + Q])$ is being calculated.

Example

Having already calculated $h(x[0+Q])$, the time has come to calculate $h(x[1+Q])$. In this example, the calculation of $\mathcal{C}_1$ is shown.

k		0	1	2	3	4	5	6	7	8
Q		1	1	1	0	1	0	1	0	1
$m(k)$		0	1	2	3	3	4	4	5	5
$Q \ll 1$	1	1	1	0	1	0	1	0	1	
$m(k+1)$	0	1	2	3	3	4	4	5	5	
Q		1	1	1	0	1	0	1	0	1
$Q \ll 1$	1	1	1	0	1	0	1	0	1	
$(k \in Q) \wedge (k+1 \in Q)$		T	T	F	F	F	F	F	F	
$m(k)$		0	1	2	3	3	4	4	5	5
$m(k+1) - m(1)$	-1	0	1	2	2	3	3	4	4	
$m(k) = m(k+1) - m(1)$		T	T	T	F	T	F	T	F	
$\mathcal{C}_1$		0	1							

The symbols at positions $\mathcal{C}_1 = \{0, 1\}$ of the hash $h(x[1 + Q])$ have already been encoded in the hash $h(x[0 + Q])$ and it is possible to reuse them without having

to compute them *ex-novo* each time. To complete the computation of the hash $h(x[1+Q])$ it is necessary to compute the remaining $|Q \setminus \mathcal{C}_1| = 4$ symbols, which must be read from x at positions $i+k$, where $i = 1$ and $k \in Q \setminus \mathcal{C}_1 = \{2, 4, 6, 8\}$.

x	C	T	T	G	T	C	G	T	T	G	[...]
$x[0+Q]$	C	T	T		T		G		T		
$\mathcal{C}_1$		0	1								
$Q \setminus \mathcal{C}_1$				2		4		6		8	
$x[1+Q]$		T	T	G		C		T		G	

For completeness, all values of $\mathcal{C}_j$ are given:

$$\begin{aligned}\mathcal{C} &= \{\mathcal{C}_1,\ \mathcal{C}_2,\ \ldots,\ \mathcal{C}_8\} \\ &= \{\{0,1\},\ \{0\},\ \emptyset,\ \{0\},\ \emptyset,\ \{0\},\ \emptyset,\ \{0\}\}\end{aligned}$$

To optimise the reuse of part of the previous hashes, it is necessary to minimise the number of times a symbol must be read and encoded. It is sufficient, therefore, to find the value j that maximises $|\mathcal{C}_j|$, and this can be solved via the function

$$ArgBH(k) = \underset{j \in \{1,2,\ldots,k\}}{\arg\max} |\mathcal{C}_j|.$$

Having already computed the previous j hashes, the best hashing value can be found at position $j - ArgBH(j)$. This will produce a saving in terms of symbols that do not have to be read and encoded again equal to $|\mathcal{C}_{ArgBH(j)}|$. For the extraction of useful symbols from the previous hashes, a mask, $Mask(j)$, is defined to filter the relevant positions. Following the observations made so far, it is possible to compute all hashing values $\mathcal{H}(x, Q)$ using the dynamic programming Algorithm 3.1.

A second algorithm, Algorithm 3.2, is also provided for use when working with more than one spaced seed. The use of more than one spaced seed increases the sensitivity [23] and, therefore, merits a dedicated approach capable of increasing the speed-up of the algorithm.

Let $\vec{Q} = \{Q_0, Q_1, \ldots, Q_{n-1}\}$ be the set of n spaced seeds, all of the same length $s(Q)$. It is possible to compute, for each spaced seed Q_j its vector $m_j(k)$ as described above. In order to compare a given spaced seed Q_j with all other spaced seeds it is

Algorithm 3.1: FSH: Fast Spaced Seed Hashing

```
1:  h_0 ← compute h(x[0 + Q]);
2:  for i ← 1 to |x| − s(Q) do
3:      if i < s(Q) then
4:          p ← ArgBH(i);
5:          h_i ← h_i or ((h_{i−p} and Mask(p)) >> (m(p) · log_2 |A|));
6:          forall k ∈ Q \ C_p do
7:              insert encode(x_{i+k}) at position m(k) · log_2 |A| of h_i;
8:      else
9:          p ← ArgBH(s(Q) − 1);
10:         h_i ← h_i or ((h_{i−p} and Mask(p)) >> (m(p) · log_2 |A|));
11:         forall k ∈ Q \ C_p do
12:             insert encode(x_{i+k}) at position m(k) · log_2 |A| of h_i;
```

necessary to redefine the set $\mathcal{C}_j$ as follows:

$$\mathcal{C}_j^{y,\,z} = \{k \in Q_y \mid k + j \in Q_z \wedge m_y(k) = m_z(k + j) - m_z(j)\}.$$

In this new definition $\mathcal{C}_j^{y,\,z}$ evaluates the number of symbols in common between the seed Q_y and the j-th shift of the seed Q_z. Similarly, it is necessary to redefine the function $ArgBH(k)$ as follows:

$$ArgBSH(y,\ k) = \underset{z\in\{0,1,\ldots,n-1\},\ j\in\{1,2,\ldots,k\}}{\arg\max} |\mathcal{C}_j^{y,\,z}|.$$

$ArgBSH(y,\ k)$ returns, for the seed Q_y, the pair of indices (z, p) representing the best seed Q_z and the best hash p. The updated algorithm can be found at page 20.

Demonstrating significant performance enhancements, FSH accelerates spaced seed hashing by $1.6\times$ compared to conventional methods across diverse seed configurations. Particularly notable is its $4\times$ to $5\times$ speedup in scenarios of high seed autocorrelation. This efficiency amplifies with longer read lengths typical of modern sequencing technologies or intricate spaced seed designs. Moreover, this work paves the path for further exploration into accelerating spaced seed hashing through innovative indexing techniques and assessing its broader utility in various bioinformatics applications.

Algorithm 3.2: Fast Multiple Spaced Seed Hashing

```
1:  for j ← 0 to n − 1 do
2:      h_{0,j} ← compute h(x[0 + Q_j]);
3:  for i ← 1 to |x| − s(Q) do
4:      for j ← 0 to n − 1 do
5:          if i < s(Q) then
6:              (z, p) ← ArgBSH(j, i);
7:              h_{i,j} ← h_{i,j} or ((h_{i−p,z} and Mask(p)) >> (m_z(p) · log_2 |A|));
8:              forall k ∈ Q_j \ C_p^{j,z} do
9:                  insert encode(x_{i+k}) at position m_j(k) · log_2 |A| of h_{i,j};
10:         else
11:             (z, p) ← ArgBSH(j, s(Q) − 1);
12:             h_{i,j} ← h_{i,j} or ((h_{i−p,z} and Mask(p)) >> (m_z(p) · log_2 |A|));
13:             forall k ∈ Q_j \ C_p^{j,z} do
14:                 insert encode(x_{i+k}) at position m_j(k) · log_2 |A| of h_{i,j};
```

3.2 ISSH: Iterative Spaced Seed Hashing

The ISSH algorithm [35] was born as a response to the request for greater optimisation of the FSH software. The latter, in fact, reuses part of a previous hash value to re-read and re-encode as few symbols as possible. With the new algorithm, ISSH, the authors attempted to further reduce the number of symbols to be re-read to a single symbol. This was done by using not only the previous hash that maximises the number of symbols to be reused, but by combining the use of several previous hashes that together cover almost all the symbols that make up the new Q-gram.

It is necessary to introduce a new definition of $\mathcal{C}_{g,j}$ that defines the positions of Q that, after j shift, continue to be in Q with the property that the positions k and $k+j$ both belong to Q and are separated by $j-g-1$ (not necessarily consecutive) characters `1`:

$$\mathcal{C}_{g,j} = \{k \in Q \mid k+j \in Q \wedge m(k) = m(k+j) - m(j) + m(g)\}.$$

The set $\mathcal{C}_{0,j}$ corresponds to the definition of $\mathcal{C}_j$ given by Girotto et al. in "FSH: fast

spaced seed hashing exploiting adjacent hashes" [34]. This is because in the algorithm FSH we always take the position 0 of $h(x[j+Q])$ as the starting point.

Example

Let x and Q be the same of the previous example. In this example, the calculation of $\mathcal{C}_{0,2}$ is shown for the calculation of $h(x[2+Q])$, $h(x[0+Q])$ having already been calculated previously.

k			0	1	2	3	4	5	6	7	8
Q			1	1	1	0	1	0	1	0	1
$m(k)$			0	1	2	3	3	4	4	5	5
$Q \ll 2$	1	1	1	0	1	0	1	0	1		
$m(k+2)$	0	1	2	3	3	4	4	5	5		
Q			1	1	1	0	1	0	1	0	1
$Q \ll 2$	1	1	1	0	1	0	1	0	1		
$(k \in Q) \wedge (k+2 \in Q)$			T	F	T	F	T	F	T		
$m(k)$			0	1	2	3	3	4	4	5	5
$m(k+2) - m(2) + m(0)$	-2	-1	0	1	1	2	2	3	3		
$m(k) = m(k+2) - m(2) + m(0)$			T	T	F	F	F	F	F		
$\mathcal{C}_{0,2}$			0								

Thus the only position recoverable from $h(x[0+Q])$ is $\mathcal{C}_{0,2} = \{0\}$.
If, on the other hand, the first position of $h(x[0+Q])$ was skipped and the hash was considered from its second position, $\mathcal{C}_{1,2}$ would be obtained:

k			0	1	2	3	4	5	6	7	8
Q			1	1	1	0	1	0	1	0	1
$m(k)$			0	1	2	3	3	4	4	5	5
$Q \ll 2$	1	1	1	0	1	0	1	0	1		
$m(k+2)$	0	1	2	3	3	4	4	5	5		
Q			1	1	1	0	1	0	1	0	1
$Q \ll 2$	1	1	1	0	1	0	1	0	1		
$(k \in Q) \wedge (k+2 \in Q)$			T	F	T	F	T	F	T		
$m(k)$			0	1	2	3	3	4	4	5	5
$m(k+2) - m(2) + m(1)$	-1	0	1	2	2	3	3	4	4		
$m(k) = m(k+2) - m(2) + m(1)$			F	F	T	T	T	T	T		
$\mathcal{C}_{1,\,2}$					2		4		6		

In the last case, the number of symbols reusable by the hash $h(x[0 + Q])$ to produce the hash $h(x[2 + Q])$ increases, being $\mathcal{C}_{1,\,2} = \{2, 4, 6\}$.

The new algorithm, Algorithm 3.3 proposed by Petrucci et al. aims to improve the efficiency of hash computation by using an iterative technique that, instead of relying only on the best previous hash, considers all previous hashes to create a combination that covers all the symbols needed for $h(x[i + Q])$, except the last one.

For this, the function $BestPrev(k,\ Q')$ is defined which returns a pair $(g,\ j)$ that identifies the best previous hash, $h(x[i-j+Q])$, from which $|\mathcal{C}_{g,\,j} \cap Q'|$ symbols, after removing its first g symbols:

$$BestPrev(k,\ Q') = \underset{g \in \{0,1,\ldots,k-1\},\, j \in \{1,2,\ldots,k\}}{\arg\max} |\mathcal{C}_{g,\,j} \cap Q'|.$$

For the extraction of useful symbols from the previous hashes, a mask, $Mask(g,\ j)$, is defined to filter the relevant positions.

Demonstrating a noteworthy average acceleration of computation, ISSH yields a speedup ranging between $3.5\times$ to $7\times$ compared to traditional hash value calculations. This enhancement varies based on the density of spaced seeds and the length of reads under consideration. Across all conducted experiments, ISSH consistently

Algorithm 3.3: ISSH: Iterative Spaced Seed Hashing

```
1:  h_0 ← compute h(x[0 + Q]);
2:  for i ← 1 to |x| − s(Q) do
3:      if i < s(Q) then
4:          Q' ← Q;
5:          while |Q'| ≠ 1 do
6:              (g, j) ← BestPrev(i, Q');
7:              if |C_{g,j} ∩ Q'| = 0 then
8:                  Exit while;
9:              else
10:                 h_i ← h_i or ((h_{i−j} and Mask(g, j)) >> (j · log_2 |A|));
11:                 Q' ← Q' \ C_{g,j};
12:         forall k ∈ Q' do
13:             insert encode(x_{i+k}) at position m(k) · log_2 |A| of h_i;
14:     else
15:         Q' ← Q;
16:         while |Q'| ≠ 1 do
17:             (g, j) ← BestPrev(s(Q) − 1, Q');
18:             h_i ← h_i or ((h_{i−j} and Mask(g, j)) >> (j · log_2 |A|));
19:             Q' ← Q' \ C_{g,j};
20:         insert encode(x_{i+s(Q)−1}) at last position of h_i;
```

surpasses the performance of existing algorithms, promising significant efficiency gains in computational tasks reliant on spaced seed hashing methodologies.

3.3 MISSH: Multiple Iterative Spaced Seed Hashing

A further improvement to the FSH algorithm was achieved with MISSH, developed by Mian et al. in "Efficient Hashing of Multiple Spaced Seeds with Application" [39]. As in the case of the Algorithm 3.2 concerning the handling of multiple spaced seeds of FSH, special attention was paid to the development of an algorithm using the same idea as MISSH, but for a set of spaced seeds.

In the article, the authors propose three different approaches. In the first version, called **MISSH Multi**, several different spaced seeds are considered simultaneously, although the hashing of the sequence of DNA is done independently for each spaced

seed. In fact, only the previous hashing values referring to the spaced seed Q_j are exploited for the calculation of $h(x[i + Q_j])$. The substantial difference with the algorithm ISSH lies in the fact that the hashing matrix is constructed by columns, as illustrated in Figure 3.1. The convenience lies in the fact that the last character of each Q-gram, which is always read for the first time, belongs to all hashing values due to the definition 2.1 of spaced seed given in page 12. The reading and encoding of the character is done only once and is valid for the entire spaced seed set.

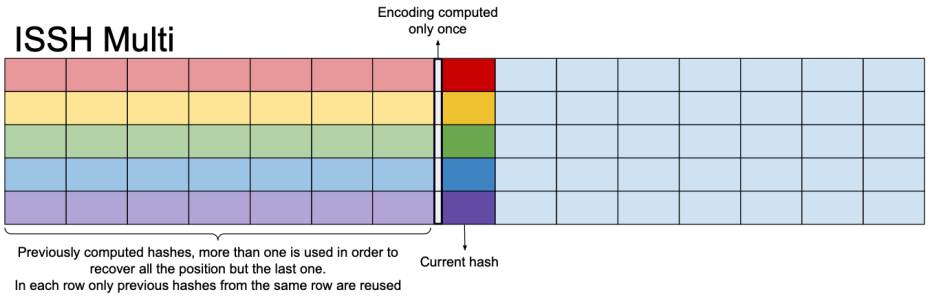


Figure 3.1: A schematic representation of the ISSH Multi approach. The rows of the matrix represent the different spaced seeds, while the columns represent the position of the sequence where to compute the hash. [39].

The second approach, **MISSH Multi Column**, builds on the previous approach and improves on it by introducing a new degree of freedom: in this case, it is allowed to retrieve positions not only from previous hashing values related to the same spaced seed, but also from spaced seeds different from the one currently under consideration. A schematic description can be found in Figure 3.2.

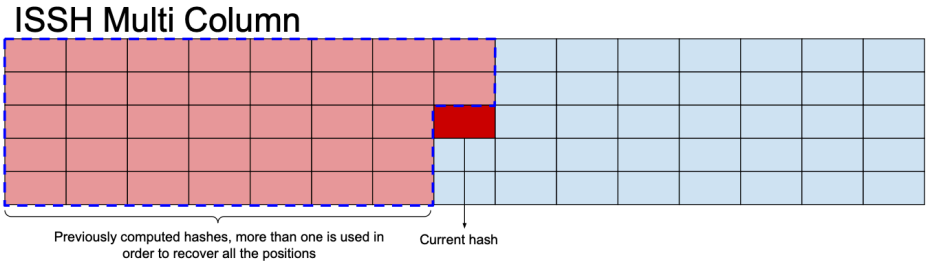


Figure 3.2: A schematic representation of the ISSH Multi Column approach. The rows of the matrix represent the different spaced seeds, while the columns represent the position of the sequence where to compute the hash. [39].

To compute a generic hash $h_{i,\,j}$, where i is the index of the Q-gram to be hashed and j is the index of the spaced seed, the algorithm searches for the hash $h_{n,\,m}$ that

allows to recover most positions among all previously computed hashes. The condition that $h_{n,\,m}$ needs to satisfy in order to be used by $h_{i,\,j}$ is

$$(n < i \textbf{ or } (n = i \textbf{ and } m < j)) \textbf{ and } n \geq 0. \tag{3.1}$$

In order to extract symbols from $h_{n,\,m}$ to be reused in the new hash $h_{i,\,j}$ is defined a mask $Mask(j,\ n,\ m,\ l)$ that filters the appropriates positions.

Algorithm 3.4: ISSH Multi Column

```
for i ← 0 to |x| − s(Q) do
    forall Q_j ∈ Q⃗ do
        h_{i,j} ← 0;
        while missing positions can be recovered from available hashes do
            (n, m, l) ← such that condition 3.1 holds and h_{n,m} with l shifts allows
              to recover the highest number of missing positions.;
            h_{i,j} ← h_{i,j} or ((h_{n,m} >> (l · log_2 |A|)) and Mask(j, n, m, l));
        if there are still missing positions then
            add missing encodings to h_{i,j};
```

This algorithm has a considerable advantage because it significantly reduces the number of encoding operations required during the transitional phase. Firstly, during this phase, the number of encoding operations is much lower: even the hashing of the first Q-gram of the second spaced seed already has the possibility of recovering positions from the first hash, which was not possible previously. Furthermore, the encryption function is only used once for each character in the sequence, even during the transition phase. This approach allows for a considerable improvement in calculation time compared to the ISSH Multi algorithm. The possibility of recovering positions as early as the first hash of the second spaced seed means that the algorithm can start obtaining useful results earlier, improving overall efficiency. Finally, the algorithm optimises the use of computational resources, reducing the workload and allowing faster and less time-consuming processing.

Finally, the third method, called **ISSH Multi Row**, follows the previous scheme by reversing the order in which it fills the hashing matrix. By filling it by rows, it is also possible to retrieve positions from hashes that were previously calculated with

a different spaced seed.

To compute a generic hash $h_{i,j}$, the algorithm searches for the hash $h_{n,m}$ that allows to recover most positions among all previously computed hashes. The condition that $h_{n,m}$ needs to satisfy in order to be used by $h_{i,j}$ is

$$(m < j \textbf{ or } (m = j \textbf{ and } n < i)) \textbf{ and } 0 \leq n \leq |x| - s(Q). \tag{3.2}$$

The schematic description can be found in Figure 3.3.

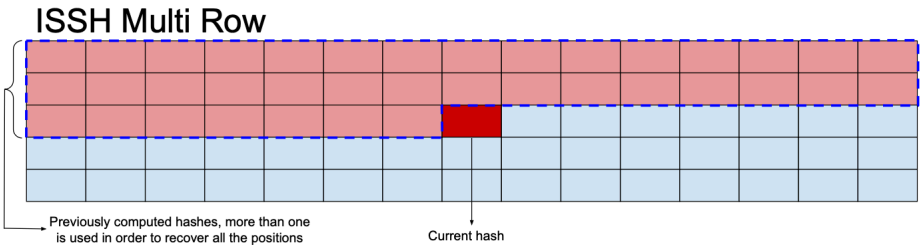


Figure 3.3: A schematic representation of the ISSH Multi Row approach. The rows of the matrix represent the different spaced seeds, while the columns represent the position of the sequence where to compute the hash. [39].

This other algorithm has distinct advantages arising from its structure and the handling of transitional phases. In particular, the introduction of successive hashes, along with the previous ones, results in a significant modification of the transient phase. These hashes are generated by the overlaps of the spaced seeds positioned further to the right of the current position. This approach results in two transitional phases, one at the beginning and one at the end of the DNA sequence. This is due to the fact that the "right-hand" hashes calculated during pre-processing will eventually be unavailable, just as the "left-hand" hashes were initially unavailable. Furthermore, this method is expected to perform better on longer sequences. The presence of two transitional phases and the handling of the "right" hashes make it more efficient in handling longer sequences. Consequently, it is able to provide better and more reliable results than other methods, especially when sequence length is a critical factor to consider.

The performance analysis of various MISSH algorithms reveals notable speedups across diverse seed configurations and read lengths. Notably, the novel approaches outperform previous methods, with ISSH Multi Column demonstrating the high-

est overall speedup, slightly edged by ISSH Multi Row in datasets with lengthier reads. Performance enhancements are particularly pronounced with longer reads due to transient time contributions, especially evident for ISSH Multi Row and ISSH Multi. MISSH variants exhibit remarkable speedups exceeding 17×, contrasting prior methods. Moreover, considering multiple spaced seeds concurrently significantly decreases computation time across all methods, showcasing substantial improvements even with smaller seed groups.

3.4 ntHash & ntHash2: Recursive (Spaced Seed) Hashing for Nucleotide Seqences

The method of ntHash [29] is based on a recursive function, known as a rolling hash function, which calculates the hash value of the current k-mer h_i from the hash value of the previous k-mer h_{i-1} via a recursive formula. To initialise the hash calculation, the first k-mer is calculated as follows:

$$h_0 = \bigoplus_{j=0}^{k-1} \mathrm{rol}^{k-1-j}\, h(x[j]).$$

In this formula, $\mathrm{rol}^j(\cdot)$ is a left-cyclic rotation, $\oplus$ is the exclusive OR (XOR) operator, and $h(\cdot)$ is a seed table where the nucleotide characters, $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, are assigned to different 64-bit random integers.

The hash value of each successive k-mer is calculated recursively:

$$\begin{aligned} h_i &= f(h_{i-1}, x[i+k-1], x[i-1]) \\ &= \mathrm{rol}^1\, h_{i-1} \oplus \mathrm{rol}^0\, h(x[i+k-1]) \oplus \mathrm{rol}^k\, h(x[i-1]) \end{aligned}$$

The time complexity of ntHash is $O(k+|x|)$, in contrast to the $O(k\cdot|x|)$ complexity of conventional hash functions.

To calculate the hash values of the inverse complementary sequences of a k-mer, ntHash uses a seed table that includes the integers corresponding to the complementary bases, allowing efficient calculation without actually inverting the input

sequence:

$$h_i^{'} = \begin{cases} \bigoplus_{j=0}^{k-1} \mathrm{rol}^j h((x[j] + d) & \text{if } i = 0 \\ \mathrm{ror}^1 h_{i-1}^{'} \oplus \mathrm{rol}^{k-1} h(x[i+k-1] + d) \oplus \mathrm{ror}^1 h(x[i-1] + d) & \text{otherwise} \end{cases}$$

where d is the offset of the complementary bases in the seed table. This property makes ntHash extremely useful in applications requiring the handling of complementary sequences.

With the introduction of ntHash2 [37], the algorithm was further improved to handle spaced seeds. One of the main improvements in ntHash2 is the introduction of the rotation function, $\mathrm{srol}^j(\cdot)$, which splits a 64-bit word into subwords w_0, $w_1, \ldots, w_{n-1}$-bit long ($\sum w_i = 64$ and $\gcd(w_i, w_j) = 1\ \forall i,\ j$), rotates the subwords separately and combines the results. The periodicity of the split rotation is equal to $\mathrm{lcm}(w_0, w_1, \ldots, w_{n-1})$, making it more suitable for longer k-mer lengths. Furthermore, to improve the uniform distribution of hashes, ntHash2 defines the canonical hash value of each seed as the sum of the forward hash and the reverse hash, replacing the old version that used the minimum between the two values.

The main innovation of ntHash2 is the method for hashing spaced seeds. First of all, the definition of a block is given, which is a sub-sequence of Q consisting of consecutive `1` characters, bounded by the `0` character or the edges of the spaced seed. To compute the hash value for the first $s(Q)$ characters, we iterate over block intervals and include the characters using $\mathrm{srol}^j(\cdot)$ and XOR operations with time complexity $O(|Q|)$. Subsequent hashes are generated by removing and including characters based on block indices, with complexity $O(|B|)$. For faster calculation, ntHash2 redefines blocks as traits of `0` if the number of expected XOR operations is less by excluding `0` from the hash value.

The function that calculates the hashing value is redefined according to this formula:

$$h_i = \begin{cases} \bigoplus_{j=0}^{k-1} \mathrm{rol}^{k-1-j} h(x[j]) & \text{if } i = 0 \\ \mathrm{srol}^1 h_{i-1} \oplus \mathrm{srol}^0 h(x[i+k-1]) \oplus \mathrm{srol}^k h(x[i-1]) & \text{otherwise} \end{cases}$$

For the hashing value of the reverse complement, however, the updated formula is as follows:

$$h_i^{'} = \begin{cases} \bigoplus_{j=0}^{k-1} \text{srol}^j\, h((x[j]+d) & \text{if } i = 0 \\ \text{sror}^1\, h_{i-1}^{'} \oplus \text{srol}^{k-1}\, h(x[i+k-1]+d) \oplus \text{sror}^1\, h(x[i-1]+d) & \text{otherwise} \end{cases}$$

3.5 MISSH vs ntHash2

MISSH and ntHash2 are both innovative tools designed for efficient hashing of nucleotide sequences, each offering unique approaches and advantages in their implementations.

MISSH, introduced as an enhancement to the FSH and ISSH algorithms, incorporates new strategies for handling multiple spaced seeds simultaneously. The ISSH Multi approach processes several spaced seeds simultaneously, optimising hashing efficiency by constructing a hashing matrix organised by columns. This design facilitates the reuse of previously calculated hashes, significantly reducing the computational workload and improving the overall processing speed. Furthermore, ISSH Multi Column further refines this concept by introducing a mechanism to retrieve positions from hashes associated with differently spaced seeds, improving the flexibility and adaptability of the algorithm. Performance analysis demonstrates substantial speed-ups in various configurations of seeds and read lengths, proving the effectiveness of MISSH in accelerating hash computation operations.

In contrast, ntHash2 builds upon the recursive hashing function of ntHash, introducing enhancements tailored for spaced seed hashing. ntHash2 revolutionizes the hashing process by incorporating a split rotation function and redefining the hashing formula to accommodate spaced seeds efficiently. By utilizing block-based hashing and optimizing hash value calculation, ntHash2 achieves remarkable performance improvements, outperforming its predecessor ntHash and other competing algorithms like CityHash and ISSH [29]. The algorithm's versatility and scalability make it suitable for diverse applications such as genome assembly and k-mer counting, offering significant advantages in speed and accuracy.

Algorithm 3.5: ntHash2: Spaced Seed Hashing Procedure

Function `parse_seed(`Q`)` **:**
  $blocks, monomers, start, is\_block \leftarrow \{\}, \{\}, 0,$ **true**;
  for $i \leftarrow 0$ **to** $s(Q) - 1$ **do**
    if $i \in Q$ **and** $is\_block =$ **false then**
      $is\_block \leftarrow true$;
      $start \leftarrow i$;
    else if $i \notin Q$ **and** $is\_block =$ **true then**
      if $i - start > 1$ **then**
        $blocks \leftarrow blocks \cup \{\langle start,\ i\rangle\}$;
      else
        $monomers \leftarrow monomers \cup \{i\}$;
      $is\_block \leftarrow false$;
  return *blocks, monomers*;
Function `base_hash` $(x, s(Q), blocks, monomers)$ **:**
  for $\langle p,\ q\rangle \in blocks$ **do**
    for $i \leftarrow p$ **to** q **do**
      $h \leftarrow h \oplus \text{srol}^{s(Q)-i-1}\, h(x[i])$;
      $h' \leftarrow h' \oplus \text{srol}^{i}\, h(x[i]+d)$; `/* d is the offset for the complementary bases */`
  $h_b,\ h'_b \leftarrow h,\ h'$;
  for $i \in monomers$ **do**
    $h \leftarrow h \oplus \text{srol}^{s(Q)-i-1}\, h(x[i])$;
    $h' \leftarrow h' \oplus \text{srol}^{i}\, h(x[i]+d)$;
  return $hash\_results \leftarrow \langle h,\ h',\ h_b,\ h'_b\rangle$;
Function `slide_hash` $(x, s(Q), blocks, monomers, hash\_results)$ **:**
  $h_b \leftarrow \text{srol}^{1}\, h_b$;
  for $\langle p,\ q\rangle \in blocks$ **do**
    $h_b \leftarrow h_b \oplus \text{srol}^{s(Q)-p}\, h(x[p]) \oplus \text{srol}^{s(Q)-q}\, h(x[q])$;
    $h'_b \leftarrow h'_b \oplus \text{srol}^{p}\, h(x[p]+d) \oplus \text{srol}^{q}\, h(x[q]+d)$;
  $h'_b \leftarrow \text{sror}^{1}\, h'_b$;
  $h,\ h' \leftarrow h_b,\ h'_b$;
  for $i \in monomers$ **do**
    $h \leftarrow h \oplus \text{srol}^{s(Q)-i-1}\, h(x[i])$;
    $h' \leftarrow h' \oplus \text{srol}^{i}\, h(x[i]+d)$;

4

A new version of our tool

In the implementation of modifications to the MISSH tool, several optimisations were introduced to improve its efficiency. One of the most significant changes concerns the function for encoding individual characters of the input nucleotide sequence. Originally, this function used a series of multiple conditional instructions to encode each character, as described in Algorithm 4.1. This logic has been replaced with a bitwise manipulation function

```
encode = ((ch >> 1) & 0b11)
```

suggested by the authors of JellyFish [17]. This type of function is known to be significantly more efficient in terms of execution time, as bitwise operations are inherently faster than multiple conditional instructions.

After the implementation of this change, some preliminary tests were conducted to evaluate the efficiency of the new encoding function. The results showed a significant improvement over the previous version. In Table 4.1, which shows the speed-ups obtained, it can be seen that the new bitwise encoding function has reduced processing times, resulting in considerable efficiency gains.

However, the introduction of the bitwise encoding function highlighted a critical issue: unlike the old function, the new one does not recognise the "N characters" in the nucleotide sequence, which represent unknown bases. The old function predisposed to the generation of warnings and errors when it encountered these characters, providing a level of control over the correctness of the data. During testing, it was observed that the symbol correctness check is already performed when the tool loads

Algorithm 4.1: Original encoding function

```
1:  Function char_to_int(ch):
2:      if ch = A then
3:          return 0;
4:      else if ch = C then
5:          return 1;
6:      else if ch = G then
7:          return 2;
8:      else if ch = T then
9:          return 3;
10:     return 4 ;                          /* ch is a N character */
```

method	original encoding	JellyFish like encoding	speed-up
naive	94.3	11.2	8.42
FSH	48.0	22.4	2.14
ISSH	27.0	12.9	2.09

Table 4.1: Comparison of processing times (expressed in milliseconds) between the original function and the bitwise function. A sequence of 1010 characters and the spaced seed $Q = 10111011$ was used for testing. The data is an average of 10 runs.

the FASTA file containing the nucleotide sequence into memory. At this stage, any incorrect symbols are handled and filtered out. Consequently, further post-loading correctness checks are redundant and time-consuming.

Another important change concerns the functions that calculate hashing. All functions have been optimised so that forward hashing and reverse hashing are calculated simultaneously. This approach not only improves the overall efficiency of the hashing process, but also ensures that the results are consistent with those obtained using the FSH, ISSH and MISSH tools, which employ the same hashing function. By implementing these changes, a significant improvement in the performance of the MISSH tool was achieved, while maintaining the reliability and correctness of the results.

In the second phase of the modifications made to the MISSH tool to improve its efficiency, it became apparent that the hash function used needed to be revised.

Previously, MISSH implemented a specific variant of the Rabin-Karp hash function. However, it became apparent that a more general hash function conforming to the canons defined in the literature was needed to ensure greater flexibility and adherence to standards.

Array of contributions

The implementation of a new hash function required a careful evaluation of the different options available. One of the initial proposals was to handle the contributions due to each character of the nucleotide sequence by storing them in a dedicated array. This array of contributions would have allowed the hashing value $h(x[i+Q])$ for each Q-gram to be calculated by summing the individual contributions of each character. In theory, this solution would have allowed any hash function to be implemented in a modular and flexible manner, combining the various contributions to form the overall hash value of the Q-gram $x[i+Q]$. However, this solution was quickly discarded after a practical evaluation. Despite the theoretical elegance of the method, the implementation showed no significant speed-up advantage over the use of the ntHash2 tool. Tests showed that the handling of contribution arrays entailed a computational overhead that nullified the potential benefits of the modular approach.

Chunk of `one_to_keep`

In the continuous search for improvements to make the MISSH tool more efficient, a new function called "chunk of `one_to_keep`" has been introduced. The `one_to_keep` variables are vectors created during the preprocessing of MISSH, which keep in memory the positions of previous hashes that are useful for calculating the current hash.

> **Example**
>
> Let $Q = \{0,1,2,3,5,6,7,9,10,11,14,16,19,20,21,23,24,25,27,28,29,30\}$ be the shape of spaced seed `111101110111001010011101110111`. The vectors $\texttt{one\_to\_keep}_i$ calculated by the preprocessing of MISSH are as fol-

lows:

$$\texttt{one\_to\_keep}_0 = \{1, 2, 3, 5, 6, 8, 9, 13, 14, 16, 17, 19, 20, 21\}$$
$$\texttt{one\_to\_keep}_1 = \{4, 7, 11, 15, 18\}$$
$$\texttt{one\_to\_keep}_2 = \{10, 12\}$$

It is important to notice that the intersection of the (disjoint) sets $\texttt{one\_to\_keep}_i$ and the set $\{0\}$ constitutes the set of positions of the characters `1` indicated by the spaced seed form Q.

The production of a new string using the vectors `one_to_keep` is done in the following way:

1. the n-th previous string is taken;
2. a shift to the left of n positions is performed;
3. the first character of each run of consecutive characters is deleted;
4. the last character of each run of consecutive characters is added.

Example

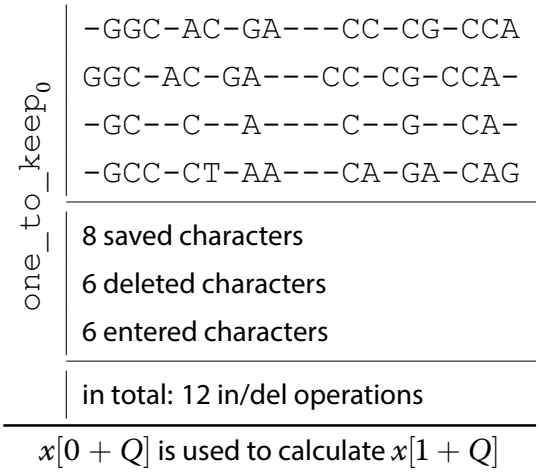
Let Q be the spaced seed of the previous example and x the nucleotide sequence $x = \texttt{AGGCCCACTGGAAGTTGTAGCCACCGAGCCAG[...]}$. The Q-gram $x[0 + Q]$ will be

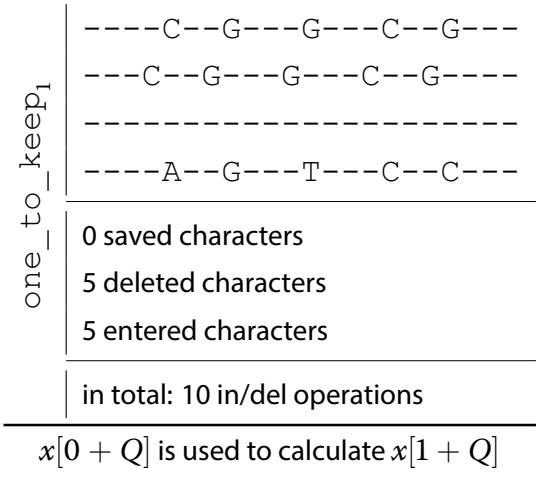
x	AGGCCCACTGGAAGTTGTAGCCACCGAGCCAG[...]
Q	11110111011100101001110111011111
$x[0 + Q]$	AGGC CAC GGA T G GCC CCG GCCA

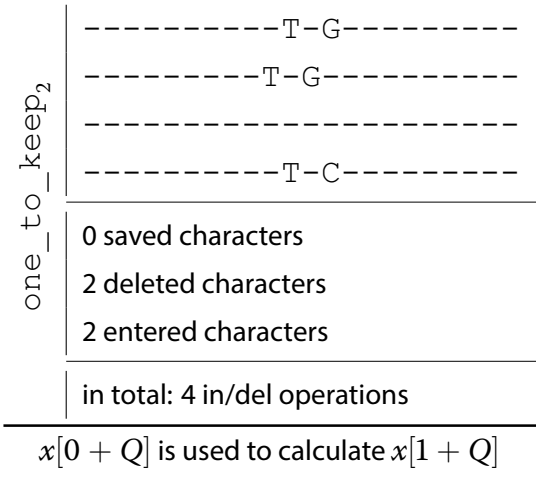
The three associated $\texttt{one\_to\_keep}_i$ vectors are:

$x[0 + Q]$	AGGCCACGGATGGCCCCGGCCA
$\texttt{one\_to\_keep}_0$	GGC AC GA CC CG CCA
$\texttt{one\_to\_keep}_1$	C G G C G
$\texttt{one\_to\_keep}_2$	T G
	A

Wanting to calculate the next Q-gram, $x[1+Q]$, given the previously calculated Q-gram, the steps to calculate are as follows:







Thus, the Q-gram $x[1+Q]$ = GGCCACTGAATTCCACGACCAGis formed, consistent with the calculation according to the naive method:

x	AGGCCCACTGGAAGTTGTAGCCACCGAGCCAG[...]
Q	11110111011100101001110111011111
$x[1+Q]$	GGCC ACT GAA T T CCA CGA CCAG

This method is only efficient if the cardinality of each run of consecutive characters is greater than unity. If, on the other hand, you take the chunk of n[1] previous positions and perform a left rotational shift of m[2] positions, you can save a few more characters.

Example

Let x and Q be the nucleotide sequence and spaced seed of the previous example.

one_to_keep$_0$

```
    -GGC -AC -GA  - -  -CC -CG -CCA
-GGC -AC -GA  - -  -CC -CG -CCA
    --AC -GA ---  - -  -CG -CC ----
    -CAC -GA -TT  - -  -CG -CC -CCG
```

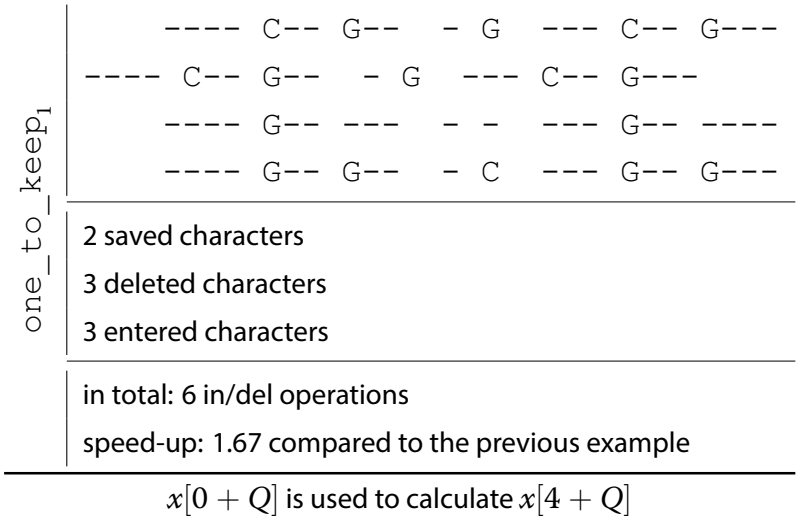
8 saved characters
6 deleted characters
6 entered characters

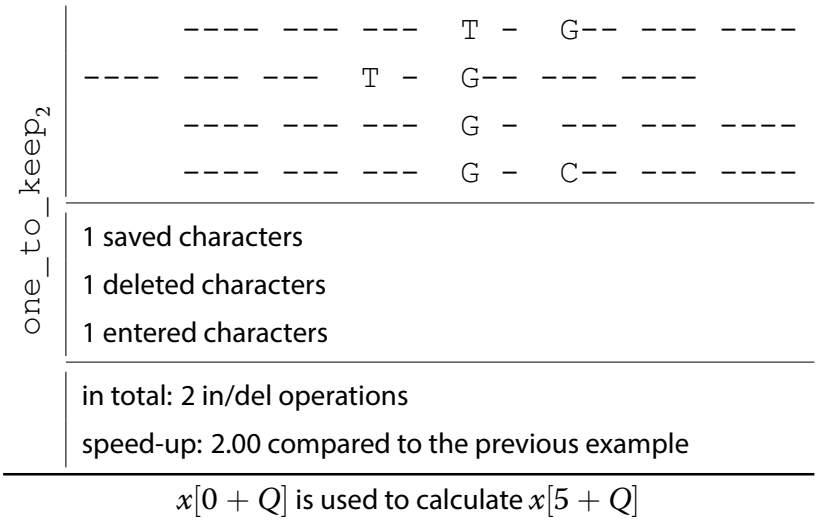
in total: 12 in/del operations
speed-up: 1.00 compared to the previous example

$x[0+Q]$ is used to calculate $x[4+Q]$

[1] $n = Q[chunk[i][1].start] + chunk[i][1].length - Q[chunk[i][0].start] - chunk[i][0].length, \forall i \in chunk.size()$.

[2] $m = chunk[i][1].start + chunk[i][1].length - chunk[i][0].start - chunk[i][0].length$





Before proceeding further with the implementation, it was necessary to evaluate the efficiency benefits. For the example case, it was calculated that for each hash, 20 in/del operations would be performed instead of 26. This should lead to an increase in efficiency of approximately 1.30 times. However, comparing this solution with ntHash2, the speed-up factor only increases from 0.54 to 0.70, remaining insufficient to exceed the performance of ntHash2.

The question arises: "Is this method actually better than the naive method?" Analysing the number of operations,

- for `one_to_keep`$_0$, 14 XOR operations are required for an *ex-novo* construc-

tion, whereas reusing a previous hash requires 12 plus 1 rotational shift[3].

- `one_to_keep`$_1$ takes 5 XOR operations to build it from scratch, while it takes 6 XOR operations and 1 rotational shift to use a previous hash.
- for `one_to_keep`$_2$, there is no difference in the number of XOR operations, but building it from scratch would save the shift.

It follows that building the three chunks from scratch would use 21 XOR operations, whereas reusing the previous hashes would require 20 XOR operations and 3 rotational shift operations (equivalent to 29 bitwise operations). Therefore, rebuilding from scratch might actually be more efficient.

`one_to_keep` as spaced seed

The next improvement in the handling of the MISSH tool involved the idea of treating `one_to_keep` vectors as spaced seeds. However, this proposal also proved to be ineffective. Treating the vector `one_to_keep` as spaced-seed produces Q-grams that do not correspond to the desired result.

Example

Let us consider a practical example:

x	AGGCCCACTGGAAGTTGTAGCCACCGAGCCAG[...]
Q	11110111011100101001110111011111
$x[0+Q]$	AGGC CAC GGA T G GCC CCG GCCA

The contributions given by the three vectors `one_to_keep`$_i$, coded as spaced seeds, are:

$i=0$	11101101100011011 0111
$i=1$	100100010001001
$i=2$	101

Using the `one_to_keep` vector as a spaced-seed results in Q-grams not related to the sequences originally sought. As can be seen in the table below, none of the given strings are descriptive of a `one_to_keep`$_i$ vector:

[3] In terms of bitwise operations, a rotational shift (right or left) corresponds to 2 simple shift operations and 1 OR operation.

$i = 0$	$i = 1$	$i = 2$
AGGCCCTAGTGAGC	ACCAT	AG
GGCCATGGTGTGCC	GCTAT	GC
GCCACGGTTTACCA	GCGGG	GC
CCCCTGATGAGCAC	CAGTT	CC
[...]	[...]	[...]

The way of adapting the vector `one_to_keep` to take into account the characters `0` due to the original spaced-seed is also not feasible,

$$\begin{array}{l||l} i = 0 & 1110011001100000000110011001111 \\ i = 1 & \quad 1000100000010000001000 1 \\ i = 2 & \quad\quad\quad\quad 100001 \end{array}$$

as the resulting strings need to be further processed and divided into several substrings in order to be used correctly.

$i = 0$	$i = 1$	$i = 2$
AGGCAGGGCCCGCC	ACAAA	AC
GGCACGACCCGCCA *	GCAGC	GA
GCCCTAACAGACAG	GAGCC	GT
CCCTGAGACAGAGC	CCTCG	CT
CCAGGGTCCGCGCC	CTTAA	CG
CACGATTCGCCCCG	CGGCG *	CG
ACTAATGGACACGG	AGTCC	AA
CTGAGGTAGAGGGT	CAAGC	CA
TGGGTTAGCGCGTC	TAGAA	TG *
[...]	[...]	[...]

The same problem of dividing the hashing value of a string is neither simple nor efficient. Consider the strings

- `GGC-AC-GA---CC-CG-CCA`, string correctly written, that is the contribution given by the first vector `one_to_keep`

- and GGCACGACCCGCCA, taken from the previous table and resulting from the use of the one_to_keep vector as spaced-seed having also considered the characters 0 of the original spaced-seed

Their hashing values according to the hash function also implemented by ntHash are 11100101000000111111010111011 and 11110111001011100011, for the first and second string respectively. There is an obvious need to manipulate the second value to make it like the first, but this task requires re-reading each character to delete its current contribution and to return the correctly repositioned contribution. This procedure is not computationally efficient and makes it impractical to handle one_to_keep vectors as spaced seeds.

4.1 DuoHash: the new version of MISSH

The final improvement of the MISSH tool took shape with the introduction of a new strategy to calculate the hashing of nucleotide sequences, significantly reducing the calculation time and increasing the overall efficiency of the process. The original hash function from which the optimisation started is the rolling hash function, the pseudo-code of which is shown in Algorithm 4.2.

Algorithm 4.2: Rolling Hash function

```
1: values ← {0x3C8BFBB395C60474, 0x3193C18562A02B4C,
     0x20323ED082572324, 0x295549F54BE24456};
2: Function getHashes (x, Q, i) :
3:     forward ← 0;
4:     reverse ← 0;
5:     foreach k ∈ Q do
6:         index ← char_to_int(x[i + k]);    // defined at Page 32
7:         forward ← forward ⊕ rol^(|Q|−1−k) values[index];
8:         reverse ← reverse ⊕ rol^k values[|values| − 1 − index];
9:     return ⟨forward, reverse⟩;
```

The DuoHash approach exploits initial encoding[4] and a structure called Hash that contains the variables for forward and reverse hashing. The basic idea behind

[4] In the DuoHash tool, what was called "encoding" in previous versions was called "hashing".

the new algorithm is the use of tables with pre-calculated hashes to speed up the hash calculation process. Instead of calculating the hash for each nucleotide base at runtime, look-up tables are used that contain pre-calculated values for all possible combinations of four nitrogen bases. This approach drastically reduces the number of operations required.

Example

To better explain the concept, let us consider an example of look-up tables.

sequence	encoding	hashing
AAAA	00000000	0x53EC3F8647623EED
CAAA	00000001	0x3B2DEE31FC53472D
GAAA	00000010	0xB622149EFBEB046D
TAAA	00000011	0xFD19ADB0B6403FFD
ACAA	00000100	0x678CD75D9AFA820D
...	...	...
TTTT	11111111	0x9400B260ACBDFF13

Given the nucleotide sequence $x = \texttt{AGGCCCACTGGAAGTTGTAGCCACCG}$ and the spaced seed $\texttt{1111011101110011101101111}$ defined as $Q = \{0, 1, 2, 3, 5, 6, 7, 9, 10, 11, 14, 15, 16, 18, 19, 20, 22, 23, 24, 25\}$, the Q-gram $x[0+Q]$ is calculated as follows:

x	AGGCCCACTGGAAGTTGTAGCCACCG
Q	1111011101110011101101111
$x[0+Q]$	AGGC CAC GGA TTG AGC ACCG

The resulting Q-gram is $x[0+Q] = \texttt{AGGCCACGGATTGAGCACCG}$. Its encoding, in accordance with MISSH and earlier versions, is

$$h(x[0+Q]) = \texttt{10010100011000101111001010010001011010 00}$$

A total of $k = 5$ groups of 8 bits (corresponding to the encoding of 4 bases) are counted. Each of these groups is used as an index to access the pre-calculated hash tables:

i	encoding	hashing
0	01101000	0x15609AFAC162C235
1	10010001	0x3DA45F3F050E3E0D
2	11110010	0x8841C2559987C40B
3	01100010	0x8249A46E23AF65F5
4	10010100	0x6105665363A7FB2D

The hashing value is then calculated using the following formula:

$$\text{hashing} = \bigoplus_{i=0}^{k-1} \text{rol}^{k-i-1}\, \text{look-up}[i]$$

In the example case, hashing takes the value

$$\begin{aligned}\text{hashing} &= \texttt{0x9AFAC162C2351560} \oplus \texttt{0x45F3F050E3E0D3DA} \oplus \\ &\quad \oplus \texttt{0x41C2559987C40B88} \oplus \texttt{0x249A46E23AF65F58} \oplus \\ &\quad \oplus \texttt{0x6105665363A7FB2D} \\ &= \texttt{0xDB54441AFF406947}\end{aligned}$$

Figure 4.1 may help in understanding the process.

To better handle forward and reverse hashing, and cases where the last group of 4 nitrogen bases is not completely filled, there are 8 look-up tables:

- `e4_to_fHash` and `e4_to_rHash` for handling complete groups of all 4 bases;
- `e3_to_fHash` and `e3_to_rHash` for the management of groups composed of 3 bases;
- `e2_to_fHash` and `e2_to_rHash` for the management of groups consisting of 2 bases;
- `e1_to_fHash` and `e1_to_rHash` for the management of groups with only 1 base.

Each table used to manage groups composed of k nitrogen bases contains 4^k

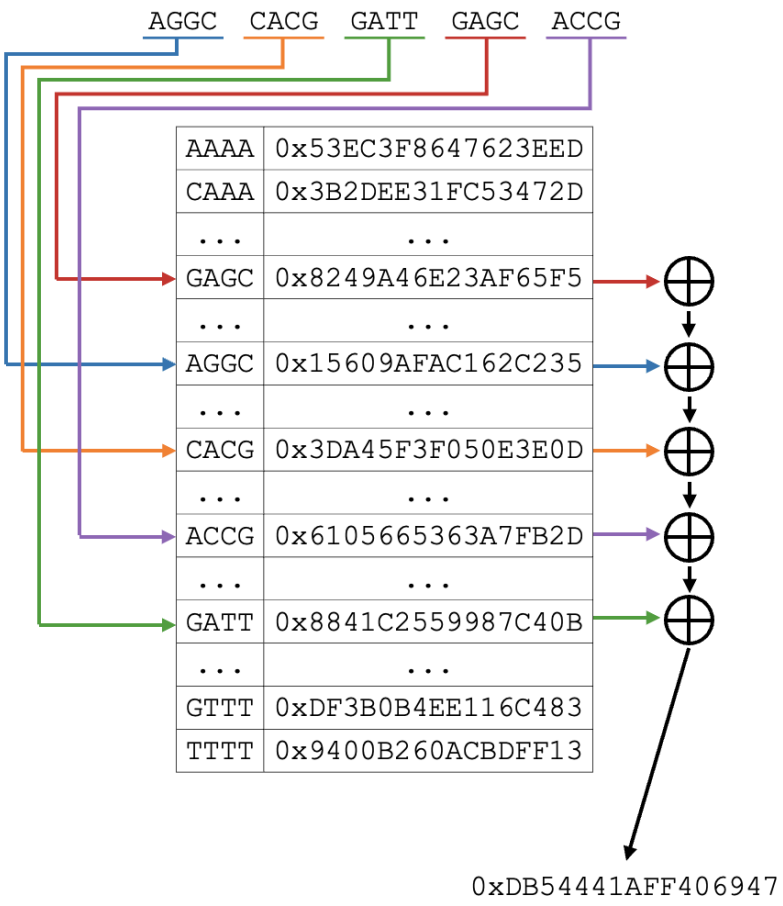


Figure 4.1: Schematic and visual procedure for using look-up tables to calculate the hashing value of the Q-gram AGGCCACGGATTGAGCACCG.

values. For each of these values, the corresponding shifts are also pre-calculated, avoiding shift operations at runtime. Taking into account that spaced seeds with a maximum weight of 32 are allowed, and that the values in the look-up tables are in groups of 4 nitrogenous bases, the possible shifts are 8. Considering that each table contains 4^k values, that for each value the corresponding 8 shifts are also pre-calculated, and that each value is a 64-bit integer (8 Bytes), each table requires memory space equal to 4^{k+3} Bytes. In total, the look-up tables occupy a total space of 21.25 kBytes[5]. With modern memory capacities, the space required to handle the 8 look-up tables is acceptable. The Algorithm 4.3 describes how the look-up tables

[5]1 Byte = 8 bits; 1 kByte = 1024 Bytes.

are pre-calculated.

Algorithm 4.3: DuoHash: look-up tables

```
1:  values ← {0x3C8BFBB395C60474, 0x3193C18562A02B4C,
      0x20323ED082572324, 0x295549F54BE24456};
2:  Function lookupTable(k):
3:      for i ← 0 to 4^k − 1 do
4:          ek_to_fHash[i][0] ← 0;
5:          ek_to_rHash[i][0] ← 0;
            /* For loop to prepare primary hashing values   */
6:          for j ← 0 to k − 1 do
7:              index ← (i >> 2j) ∧ 0b11;
8:              ek_to_fHash[i][0] ← ek_to_fHash[i][0] ⊕ rol^(k−j−1) values[index];
9:              ek_to_rHash[i][0] ←
                  ek_to_rHash[i][0] ⊕ rol^j values[|values| − 1 − index];
            /* For loop to populate the shifts table        */
10:         for j ← 0 to 8 − 1 do
11:             ek_to_fHash[i][j] ← rol^(4j) ek_to_fHash[i][0];
12:             ek_to_rHash[i][j] ← rol^(4j) ek_to_rHash[i][0];
13:     return ⟨ek_to_fHash, ek_to_rHash⟩;
```

The starting point of the new algorithm is the production of only the encoding of the nucleotide sequence, i.e. what was called the hashing value in previous versions of the software. This encoding is stored within a structure called `Hash`, which also contains two variables set for forward and reverse hashing. The latter are calculated only later, using the specially optimised `getHashes` function. The function, as illustrated in the Algorithm 4.4, receives two parameters: the structure `Hash` and the value $s(Q)$. The variable `encoding` is temporarily broken down into bytes, each of which represents the encoding of 4 nitrogen bases. Using the byte value as an index, the function accesses a series of tables of pre-calculated values, as described in the previous section.

This solution was developed to address some of the limitations of the previous techniques used in MISSH, which, although effective, had room for improvement in terms of computational efficiency. The new strategy is based on the idea of avoiding the repetitive calculation of the same values and instead exploiting a pre-computed

Algorithm 4.4: DuoHash: getHashes function

```
1:  Function getHashes(Hash, s(Q)):
2:      bytes ← s(Q)/4;                    // Hash = ⟨encoding, forward, reverse⟩
3:      for i ← 0 to bytes do
4:          curr_byte ← i-th byte of encoding;
5:          forward ← forward ⊕ e4_to_fHash[curr_byte][bytes − i − 1];
6:          reverse ← reverse ⊕ e4_to_rHash[curr_byte][i];
7:      if s(Q) mod 4 ≠ 0 then
8:          curr_byte ← bytes-th byte of encoding;
9:          forward ← rol^(s(Q) mod 4) forward;
10:         if s(Q) mod 4 = 3 then
11:             forward ← forward ⊕ e3_to_fHash[curr_byte][0];
12:             reverse ← reverse ⊕ e3_to_rHash[curr_byte][bytes];
13:         else if s(Q) mod 4 = 2 then
14:             forward ← forward ⊕ e2_to_fHash[curr_byte][0];
15:             reverse ← reverse ⊕ e2_to_rHash[curr_byte][bytes];
16:         else if s(Q) mod 4 = 1 then
17:             forward ← forward ⊕ e1_to_fHash[curr_byte][0];
18:             reverse ← reverse ⊕ e1_to_rHash[curr_byte][bytes];
```

look-up table, which drastically reduces the number of operations required to obtain the desired hashes. The `getHashes` function was implemented to make the most of this optimisation, ensuring that the necessary values are always readily available without having to recalculate them each time. The approach taken also takes into account the need to handle variable-length sequences efficiently. Indeed, handling $s(Q) \bmod 4$ makes it possible to deal with cases where the length of the sequence is not an exact multiple of 4 nitrogenous bases.

A further significant advantage of this implementation is the ease with which the hashing function can be modified. Thanks to the modular structure, it is possible to update the `getHashes` function to switch from a rolling hash function to any other hashing function, without having to modify other parts of the code. This makes the tool extremely flexible and easily adaptable to new requirements or hashing algorithms, improving its longevity and usefulness. The possibility of easily changing the hashing function is made possible by the fact that the encoding, initially calculated

by MISSH, provides a robust and flexible basis on which different hashing strategies can be applied. The integration of this new hashing function with the other components of MISSH required careful consideration of the overall architecture of the tool. The decision to initially produce only the forward encoding and to postpone the calculation of the forward and reverse hashes to a later stage was taken in order to maximise efficiency and reduce the initial computational load. This approach allows all the necessary encodings to be accumulated before proceeding with the calculation of the hashes, thus optimising the use of resources and improving the overall speed of the process. Furthermore, the choice of using a `Hash` structure to contain both the encoding and the forward and reverse hashes made the code more modular and easier to maintain. This structure makes it possible to isolate the calculation of the hashes from other operations, facilitating debugging and eventual updating of the code. The `getHashes` function has been designed to be highly efficient, minimising the number of operations required and making maximum use of the calculation capabilities of modern CPUs.

The innovative approach adopted in this new tool DuoHash represents a significant step forward compared to previously used techniques. The combination of an optimised data structure, the use of pre-computed look-up tables and the efficient management of remainders results in a significant improvement in performance, making the tool more competitive and suitable for handling large amounts of data with greater speed and accuracy. The benefits of this approach will be further explored in the following chapters, where the results of performance tests and comparisons with other tools will be presented, demonstrating the effectiveness and superiority of the new strategy adopted.

4.2 DuoHash: new features

The latest update of the MISSH tool, called DuoHash, introduces a number of new features that significantly improve the flexibility and efficiency of the hashing process of nucleotide sequences. The two main new features are the possibility to easily modify the hash function and the implementation of the `getSpacedKmer` function, which converts encodings into nucleotide sequences and saves them in a FASTA file, creating a dataset that can be used by third-party tools such as JellyFish. These improvements make DuoHash an extremely versatile and powerful tool

for analysing genomic sequences.

One of the strengths of DuoHash is the possibility to easily change the hash function without compromising the efficiency of the tool. The new structure of DuoHash is designed in such a way that the hashing function can be replaced by mainly acting on the `getHashes` function. This makes it possible to quickly adapt the tool to different calculation requirements and hashing algorithms, while still maintaining the same base of encoding values calculated by MISSH. The current implementation of DuoHash uses a rolling hash function, which exploits pre-computation of values and bitwise operations to guarantee efficient calculation. However, due to the modularity of the system, other hash functions can be implemented with only a few modifications to the code. For example, one could replace the rolling hash function with a hash function based on the algorithm of Fowler-Noll-Vo (FNV) [6], known for its simplicity and efficiency. The Algorithm 4.5 describes a possible implementation of the hash function FNV-1A.

Algorithm 4.5: DuoHash: getHashes function with FNV-1A hash function.

```
1: FNV_offset_basis ← 0xCBF29CE484222325;
2: FNV_prime ← 0x00000100000001B3;
3: Function getHashes (Hash, s(Q)) :
4:     k ← ⌈s(Q)/4⌉;                    // Hash = ⟨encoding, forward, reverse⟩
5:     forward ← FNV_offset_basis;
6:     reverse ← FNV_offset_basis;
7:     for i ← 0 to k − 1 do
8:         forward ← (forward ⊕ i-th byte of encoding) × FNV_prime;
9:         reverse ← (reverse ⊕ (k − 1 − i)-th byte of encoding) × FNV_prime;
```

Creation of FASTA file

In addition to the flexibility of the `getHashes` function, DuoHash introduces the `getSpacedKmer` function, shown in the Algorithm 4.6. This function converts the encodings into nucleotide sequences and saves them in a FASTA file, making it possible to use the data with third-party tools such as JellyFish [17], a programme used for fast k-mer counting in DNA sequences.

This function works by breaking down the encoding variable into 1-byte chunks and converting them into nucleotide characters using look-up tables (`e4_to_char`,

Algorithm 4.6: DuoHash: getSpacedKmer function

```
1:  Function getSpacedKmer(Hash, s(Q)):
2:      k ← ⌊s(Q)/4⌋;                          // Hash = ⟨encoding, spacedKmer⟩
3:      for i ← 0 to k do
4:          curr_encoding_byte ← i-th byte of encoding;
5:          curr_spacedKmer_word is the i-th word of 32-bits of spacedKmer;
6:          curr_spacedKmer_word ← e4_to_char[curr_encoding_byte];
7:      if s(Q) mod 4 ≠ 0 then
8:          curr_encoding_byte ← k-th byte of encoding;
9:          curr_spacedKmer_word is the k-th word of 32-bits of spacedKmer;
10:         if s(Q) mod 4 = 3 then
11:             curr_spacedKmer_word ← e3_to_char[curr_encoding_byte];
12:         else if s(Q) mod 4 = 2 then
13:             curr_spacedKmer_word ← e2_to_char[curr_encoding_byte];
14:         else if s(Q) mod 4 = 1 then
15:             curr_spacedKmer_word ← e1_to_char[curr_encoding_byte];
16:     spacedKmer[k] = '\0';
```

`e3_to_char`, etc.). The result is a nucleotide sequence `spacedKmer` that is then saved in a FASTA file.

5 Results

5.1 Tools and Experimental Setup

In this section, the tools and experimental setup used for the validation and testing of the developed software are described. The analysis focuses on the different datasets, seedset, and computational platform employed. Each component is critical in assessing the performance, efficiency, and scalability of the software under various conditions and constraints. This comprehensive setup ensures that the software is rigorously tested and its capabilities thoroughly evaluated.

5.1.1 Dataset

For the validation and testing of the developed software, two distinct groups of artificial datasets were used, designed in such a way as to vary one of the two main parameters: the length and the number of reads. The structure of each group, summarised in Table 5.1, is described in detail here:

- the first group of datasets, denoted by the letter "L", is characterised by reads of a constant length of 80bp. The variability between the datasets in this group is given solely by the number of reads, which varies from a minimum of 500,000 to a maximum of 5,000,000. This variation makes it possible to assess the performance of the software in relation to the volume of data processed, while keeping the length of the reads constant.
- the second group of datasets, denoted by the letter "R", keeps the number

of reads constant at 500,000. The length of the reads in this group varies from 250 to 5,000bp. This makes it possible to examine the impact of sequence length on software performance, while keeping the number of reads unchanged.

The heterogeneity of the datasets was designed to test the software under different conditions and simulate real usage scenarios. In particular, the L-group datasets allow us to observe how the software scales as the volume of data increases, while the R-group datasets allow us to understand how software performance is affected by the length of reads. These analyses are crucial in assessing the efficiency, speed and scalability of the software, ensuring that it can adequately handle different types of genomic data.

Dataset	Number of reads	Reads length
L500000	500,000	80
L1000000	1,000,000	80
L1500000	1,500,000	80
L2000000	2,000,000	80
L5000000	5,000,000	80
R80	500,000	80
R200	500,000	250
R350	500,000	350
R500	500,000	500
R1000	500,000	1,000
R1500	500,000	1,500
R2000	500,000	2,000
R5000	500,000	5,000

Table 5.1: Number of reads and average lengths for each of the dataset used in the experiments.

5.1.2 Seedset

In the initial design phase of the experimental setup, the use of the same set of spaced seeds (seedset) used in previous versions of the software was considered. However, to enable an accurate comparison with the ntHash2 tool, it was necessary to modify the spaced seeds so that they were symmetrical. For ntHash2, in fact, the symmetry

of the spaced seeds is fundamental for the calculation of the reverse hashing[1]. It is important to emphasise that the new version of MISSH does not require spaced seed symmetry, which is a considerable advantage in terms of flexibility: accepting symmetric seeds and retaining the ability to support asymmetric variants ensures that the tool remains robust and able to meet the different needs and preferences of users.

Each spaced seed set is initially composed of three sets of three spaced seeds, designed to meet specific criteria:

- maximisation of the probability of success [31],
- minimisation of overlap complexity [27],
- maximisation of sensitivity [27].

Example

Below is an example of the original seedset W22L31, which groups spaced seeds of weight 22 and length 31.

Spaced seeds maximizing the hit probability	
Q1	1111011101110010111001011011111
Q2	1111101011100101101110011011111
Q3	1111101001110101101100111011111
Spaced seeds minimizing the overlap complexity	
Q4	1111010111010011001110111110111
Q5	1110111011101111010010110011111
Q6	1111101001011100111110101101111
Spaced seeds maximizing the sensitivity	
Q7	1111011110011010111110101011011
Q8	1110101011101100110100111111111
Q9	1111110101101011100111011001111

A total of six seedsets of different weights and lengths were used, as shown in Table 5.2.

[1]It is not necessary, instead, for the calculation of the forward hash only.

Seedset	Brief description
W10L15	this seedset contains spaced seeds of weight 10 and length 15
W14L31	this seedset contains seeds of weight 14 and length 31
W18L31	seedset with weight 18 and length 31
W22L31	includes spaced seed of weight 22 and length 31
W26L31	with weight 26 and length 31
W32L45	seedset with weight 32 and length 45

Table 5.2: Seedset used in the experiments.

The heterogeneity of the seedsets makes it possible to assess the efficiency of the tool in various situations. In particular, the four seedsets with a length of 31 allow the tool's efficiency to be analysed as weight increases. This diversity of seedsets is fundamental to understanding how the tool performs under different conditions and constraints, offering a complete overview of its capabilities.

The use of seedsets with varying weights and lengths makes it possible to examine the impact of these parameters on tool performance. Seedsets with higher weights tend to have higher sensitivity, while those with longer lengths can improve specificity. This balance between sensitivity and specificity is crucial to optimise the use of the tool in practical applications. The choice of a diverse set of spaced seeds allowed for a thorough and versatile analysis of the tool, ensuring that its performance is adequately tested and validated in a wide range of possible situations.

The flexibility introduced by the acceptance of symmetric and asymmetric spaced seeds represents a significant step forward in the tool's evolution, making it suitable for a variety of application contexts.

5.1.3 Machine

In the experimental setup, the computational tasks were performed on a personal MacBook Pro, late 2020 model, equipped with the revolutionary Apple M1 processor. This processor, based on the `arm64` architecture, offers remarkable speed and efficiency thanks to its octa-core CPU configuration, which includes four high-performance cores and four high-efficiency cores, enabling seamless multitasking and energy optimisation. In addition, the M1 chip utilises a unified memory architecture, integrating RAM directly into the processor package for improved performance and power efficiency. With 16GB of unified memory at its disposal, the

MacBook Pro M1 offers unparalleled responsiveness and fluidity, making it an ideal platform for computational analysis.

The `arm64` architecture of the M1 chip provided a smooth transition from traditional `x86` processors during code compilation. One of the main considerations was the need to explicitly use version 13 of the `g++` compiler. This adjustment ensured the compatibility and optimal performance of the algorithms on the Apple M1 chip, emphasising its versatility and effectiveness in handling diverse computational tasks.

5.2 Analysis of the time performances

In this chapter, we present a detailed analysis of the time performance of the new tool DuoHash compared to the existing tool ntHash2. The various tests conducted were aimed at evaluating the efficiency and speed of DuoHash in comparison to ntHash2. The results are presented in terms of speed-up, which is calculated using the formula:

$$\text{speed-up} = \frac{\text{reference time}}{\text{time to be evaluated}}$$

where "reference time" refers to the time taken by ntHash2 and "time to be evaluated" refers to the time taken by DuoHash.

To ensure the accuracy of the results, each configuration was tested 10 times, and the average of these values was taken. This repetition helps to mitigate any anomalies or inconsistencies that may arise during individual test runs. For greater precision, the times were measured in microseconds and subsequently converted to milliseconds.

Additionally, to objectively evaluate the two tools, the test scripts were optimized and compiled using the `-O3` optimization option of the `GNU` compiler. This ensures that both tools are operating at their highest potential performance during the tests.

The detailed results of these performance evaluations are presented in this chapter and further elaborated upon in the appendix.

5.2.1 General analysis

The speed-up achieved by DuoHash tends to remain constant, or decrease slightly, in the various methods presented when the seedset is set, regardless of the dataset

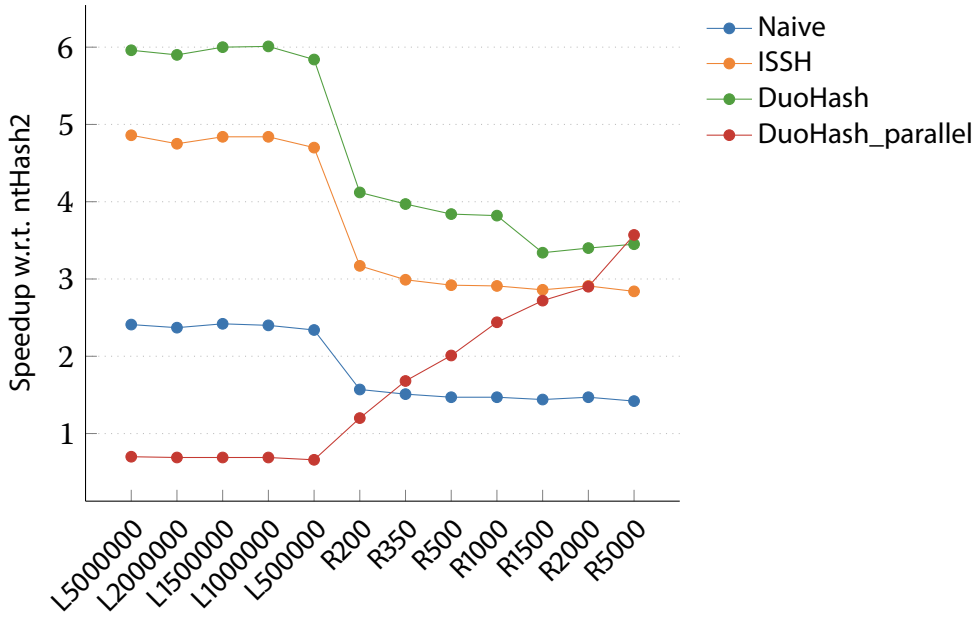


Figure 5.1: Speed-up graph for seedset W26L31

being processed. This trend is particularly evident for datasets in the "L" group, where the speed-up is highest. For the "L" group datasets - characterised by reads of varying number, but constant length - the method demonstrates a constant and significant speed-up in all scenarios tested. This indicates that the software scales efficiently with the data volume when the length of the reads remains unchanged.

For datasets in the "R" group that maintain a constant number of reads, but vary in length - the speed-up shows greater variability. In this group, the speed-up generally tends to decrease slightly as the length of the reads increases. Despite this, the overall performance of the instrument remains robust, demonstrating its ability to effectively handle different lengths of reads.

Analysing the results, it clearly emerges that the `DuoHash` method prevails over all other methods in almost all situations, with a speed-up between $3.34\times$ and $11.23\times$. This dominance is evident in both the "L" and "R" group datasets, with the exception of one specific case: in the "R5000" dataset, the `DuoHash_parallel` method outperforms `DuoHash` in terms of performance, arriving at a speed-up of $9.05\times$ in the L5000000 dataset. This outstanding result requires further analysis.

The `DuoHash_parallel` method deserves a separate discussion due to its

unique performance characteristics. This method shows its true potential with longer readings. Speed-up becomes particularly advantageous when the length of reads reaches and exceeds 5,000bp. This significant performance improvement highlights the method's ability to efficiently handle large, complex data sets with long reads. The parallelisation strategy employed by `DuoHash_parallel` allows it to process such data sets more efficiently, making it an ideal choice for scenarios involving long genomic sequences.

5.2.2 Performance Evaluation with Varying Seed Weight

The weight of a spaced seed is a critical factor that can influence the sensitivity and specificity of the software. To assess the impact of the weight of the spaced seed, four seed sets with a fixed length of 31 and different weights were used: W14L31, W18L31, W22L31 and W26L31.

For the "L" group datasets, the speed-up shows fluctuations without a definite pattern. In general, there is a slight decrease in speed-up as the seed weight increases. However, there are significant upward deviations for seedset W18L31, indicating that this particular seedset performs exceptionally well under certain conditions. Figure 5.2 graphically depicts the speed-up variations for the `DuoHash` method across the seedsets and datasets considered in this section. The highest speed-up achieved by this dataset was 11.23× with seedset W18L31.

For the datasets in the "R" group, speed-up follows a similar trend to that observed in the "L" group. The general trend is a decrease in speed-up with increasing seed weight. However, the upward deviations observed for seedset W18L31 in the "L" group datasets are much less pronounced in the "R" group datasets. This suggests that, although the seedset W18L31 continues to perform well, its relative advantage is reduced when it comes to variable read lengths. Figure 5.3 graphically depicts the speed-up variations for the `DuoHash` method across the seedsets and datasets considered in this section. The maximum speed-up achieved for the "L" group datasets was 10.35× with seedset W18L31.

5.2.3 Performance Evaluation with Varying Seed Length

The length of the spaced seed is another crucial parameter that can influence software performance. To assess the impact of spaced seed length, seedsets with differ-

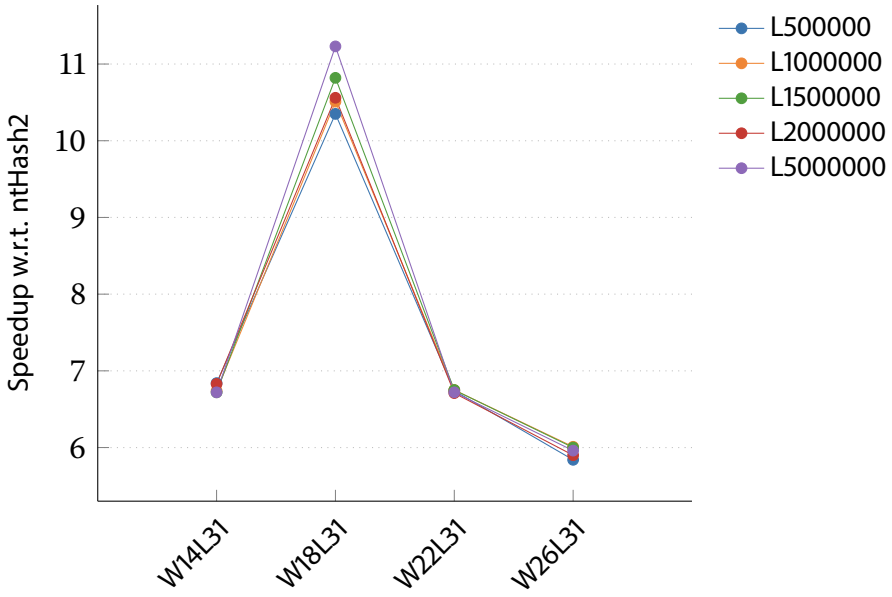


Figure 5.2: Speed-up graph for method `DuoHash` among seedset with varying weight and dataset of "L" group.

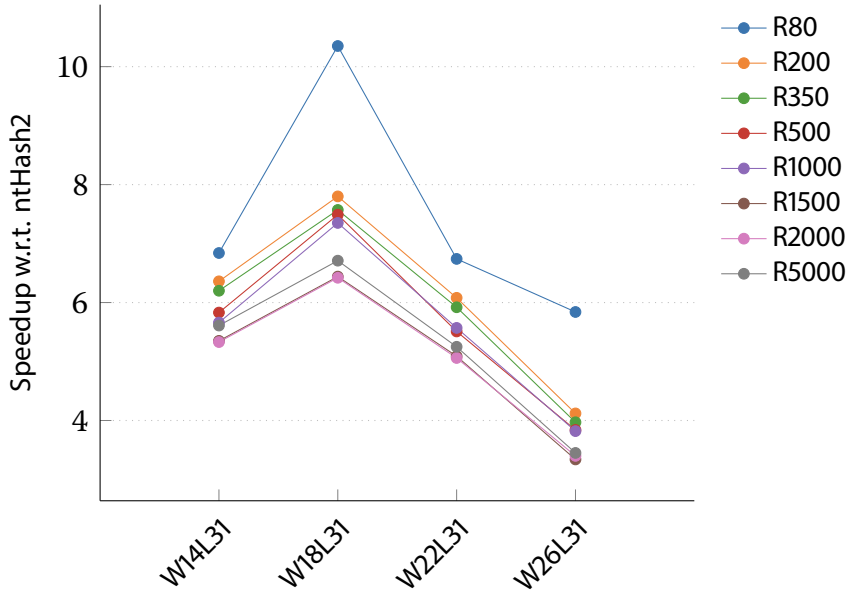


Figure 5.3: Speed-up graph for method `DuoHash` among seedset with varying weight and dataset of "R" group.

ent lengths and fixed weights were used: W10L15, W22L31 and W32L45. As there are only three seedsets, a variation in one of them can significantly influence the

overall trend. Although the pattern of behaviour is clear and consistent in all the cases analysed (see Figure 5.4), we cannot speak of a well-defined trend. This figure clearly shows the "step" pattern with seedset W10L15 showing lower speed-up values than seedsets W22L31 and W32L45, which tend to stabilise at higher values.

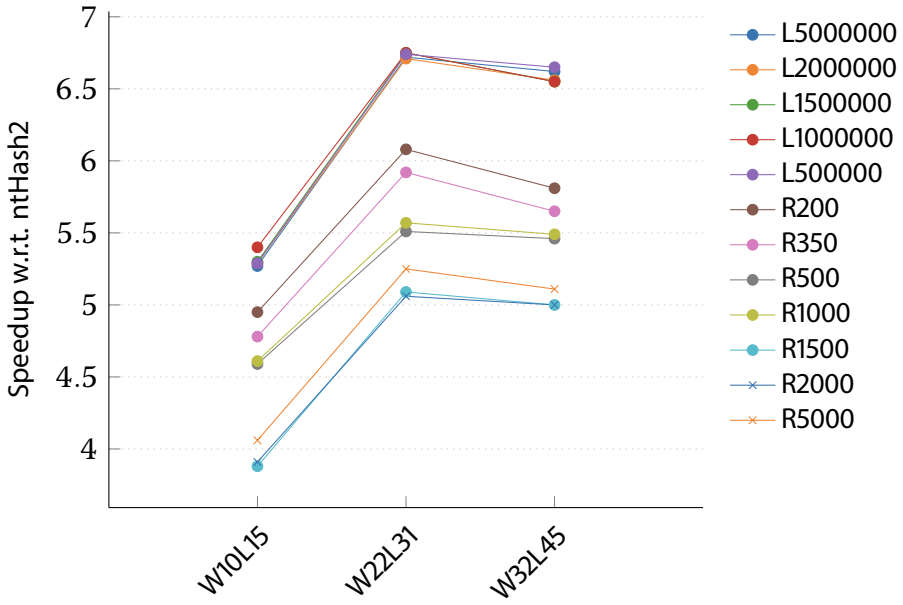


Figure 5.4: Speed-up graph for method `DuoHash` among seedset with varying length and dataset of both "L" and "R" groups.

For the L-group datasets, a "stepped" pattern is observed in the speed-up, with the first value (W10L15) being reduced by approximately 20 per cent compared to the next two values (W22L31 and W32L45), which tend to remain constant. This behaviour might suggest that, beyond a certain seed length, the efficiency of the tool remains stable. The maximum speed-up achieved for the L-group datasets was 6.75× with seedset W22L31.

For the R-group datasets, the speed-up also shows a similar "step" pattern. The initial value (W10L15) is about 20 per cent lower than the other two seedsets (W22L31 and W32L45), which maintain constant values. For this group of datasets, a maximum speed-up of 6.08× was achieved with seedset W22L31.

5.2.4 Performance Comparison: Multiple-Seed vs. Single-Seed

To confirm the validity of the multiple-seed version of the software, the performance between the best method for single-seed mode and the best method for multiple-seed mode was compared. This comparison is crucial to determine whether the multiple-seed implementation offers a significant advantage over the single-seed version. The following were chosen:

- the `ISSH` method for the single-seed mode;
- the `DuoHash` method for the multi-seeded mode;

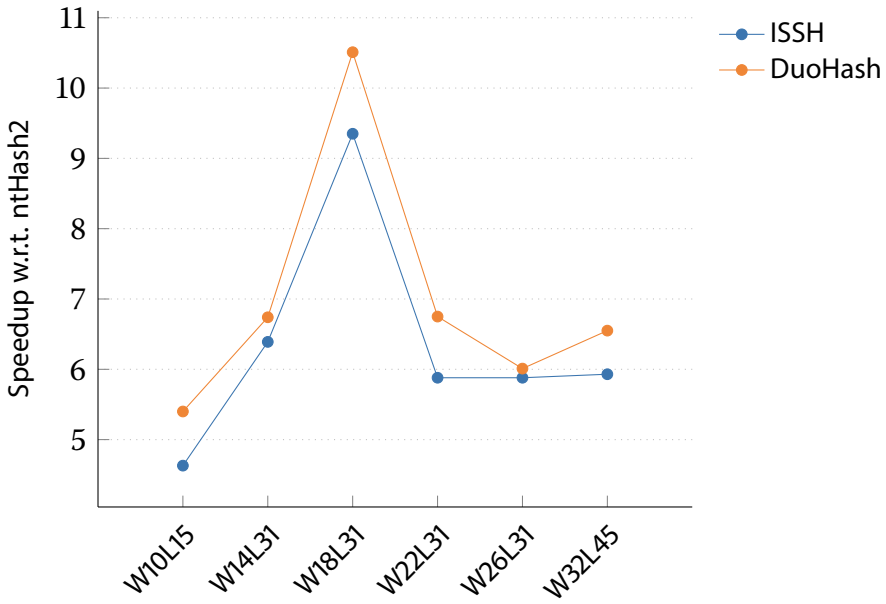


Figure 5.5: Speed-up comparison between single-seed and multiple-seed versions (L1000000 dataset).

The results represented in Figure 5.5 show that the speed-up of the two versions, even if restricted to the L1000000 dataset only, does not differ substantially, but the multiple-seed version prevails in terms of overall performance.

5.3 Analysis of the time performances in k-mer Counting

In this section, a detailed comparison will be made between two tools for extracting spaced k-mer from nucleotide sequences: DuoHash and MaskJelly [36]. Both

tools are set to output a FASTA file containing the spaced *k*-mer extracted from the nucleotide sequence provided as input. The generated FASTA file will serve as input for JellyFish, a software tool known for counting *k*-mer. This comparison will allow us to evaluate the efficiency and performance of the two tools in the context of preprocessing sequences for *k*-mer counting.

MaskJelly is a tool developed in C++ with similar functionality to DuoHash. It is also designed for spaced *k*-mer extraction, but differs in that it only works with one spaced seed at a time. This limitation makes MaskJelly less flexible than DuoHash, and a direct comparison in single-seed mode will make it possible to assess the actual performance differences between the two tools. For a fair comparison with MaskJelly, DuoHash will be used in single-seed mode, limiting its operation to one spaced seed at a time.

JellyFish is a popular *k*-mer counting software in bioinformatics. It is known for its speed and efficiency in *k*-mer counting, thanks to the use of advanced data structures such as hash tables. It should be noted that, to date, JellyFish does not directly support the handling of spaced seeds, limiting itself to the counting of contiguous *k*-mer. This limitation underlines the importance of integration with tools such as MaskJelly and DuoHash, which pre-process sequences to extract spaced *k*-mer, representing a substantial step forward in genomic research.

For this comparison, both tools, DuoHash and MaskJelly, were configured to process a series of nucleotide sequences and generate FASTA files containing the extracted spaced *k*-mer. These files were then used as input for JellyFish, which performed the *k*-mer count. The entire process was evaluated in terms of execution time and resource utilisation in order to determine which tool offers better performance in preprocessing sequences. To perform a detailed comparison between the DuoHash and MaskJelly tools, four datasets were selected from those presented above: L500000, L20000, R500 and R2000. This selection provides a comprehensive overview of the performance of the tools. Representing the different seedsets, seedset W22L31 was chosen. Both tools were configured to process these nucleotide sequences and generate FASTA files containing the extracted spaced *k*-mer. These files were subsequently used as input for JellyFish, which performed the *k*-mer count. The entire process was evaluated in terms of execution time, in order to determine which tool offers better performance in preprocessing sequences. The speed-up of

DuoHash compared to MaskJelly is depicted in Figure 5.6. As can be seen from the graph, DuoHash offers a significant performance improvement in terms of execution time compared to MaskJelly on all four datasets considered, with an average speed-up ranging between 5.19× and 6.46×.

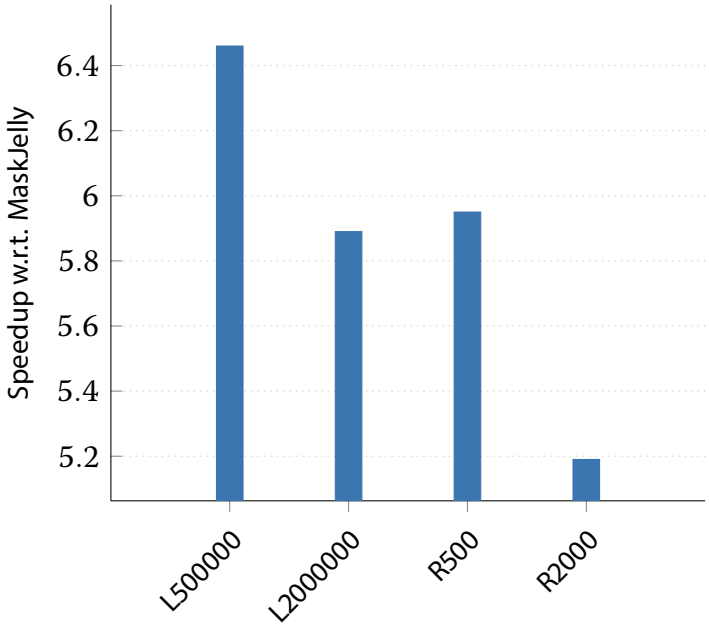


Figure 5.6: Speed-up graph for DuoHash with respect to MaskJelly (pre-processing only).

To gain a comprehensive understanding of the overall performance improvement provided by DuoHash, it is also crucial to compare the speed-up of the entire process, which includes both the pre-processing of the nucleotide sequences and the subsequent *k*-mer counting by JellyFish. By evaluating the total execution time for the combination of spaced *k*-mer extraction and *k*-mer counting, we can assess the impact of DuoHash's faster pre-processing on the overall workflow. Figure 5.7 illustrates this comparison, showing the total execution time for each dataset when using DuoHash and MaskJelly. As the graph indicates, the integration of DuoHash significantly accelerates the complete process, offering a speed-up ranging between 1.90× and 2.14×. This enhancement underscores DuoHash's efficiency not only in the preprocessing stage but also in the context of the entire *k*-mer counting workflow, making it a highly effective tool for genomic sequence analysis.

In addition to verifying the speed-up of DuoHash compared to MaskJelly, it is

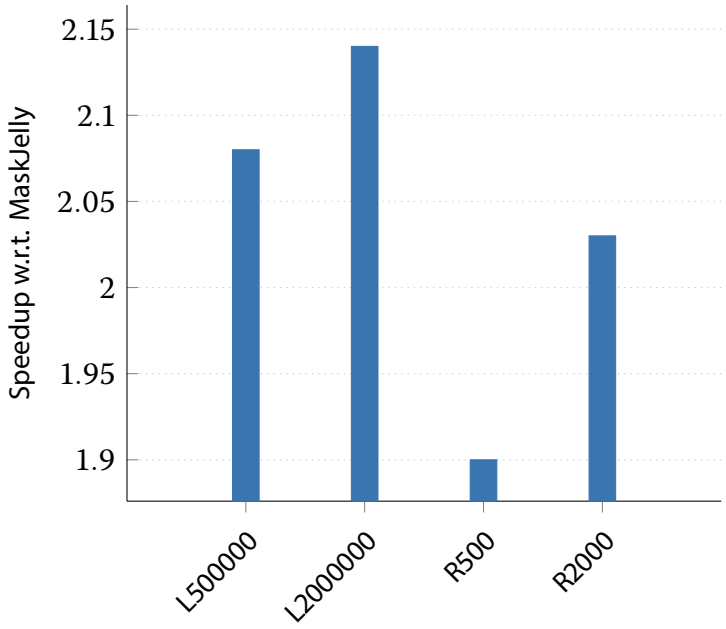


Figure 5.7: Speed-up graph for DuoHash with respect to MaskJelly (entire process).

interesting to evaluate the incidence of preprocessing on the entire counting process, which includes the extraction of the spaced *k*-mer and the execution of JellyFish. In Figure 5.8, each column represents the fraction of time required for preprocessing, while the red horizontal line indicates the execution time of JellyFish. DuoHash, accounting for only about 20 per cent of the total process time compared to MaskJelly's 60 per cent, saves considerable time and increases overall efficiency, making it a preferred choice for pre-processing before counting with JellyFish.

The comparison between DuoHash and MaskJelly, with JellyFish used for *k*-mer counting, confirms that DuoHash offers a substantial advantage in speed and resource efficiency, even when used in single-seed mode. This makes DuoHash a superior choice for the extraction of spaced *k*-mer. The validity of the multiple-seed version of DuoHash, demonstrated in previous sections, further highlights the flexibility and power of this tool in the context of bioinformatics applications, and its integration with JellyFish represents a significant advancement in genomics research, filling the current gap in the management of spaced seeds and optimising the entire sequence analysis process.

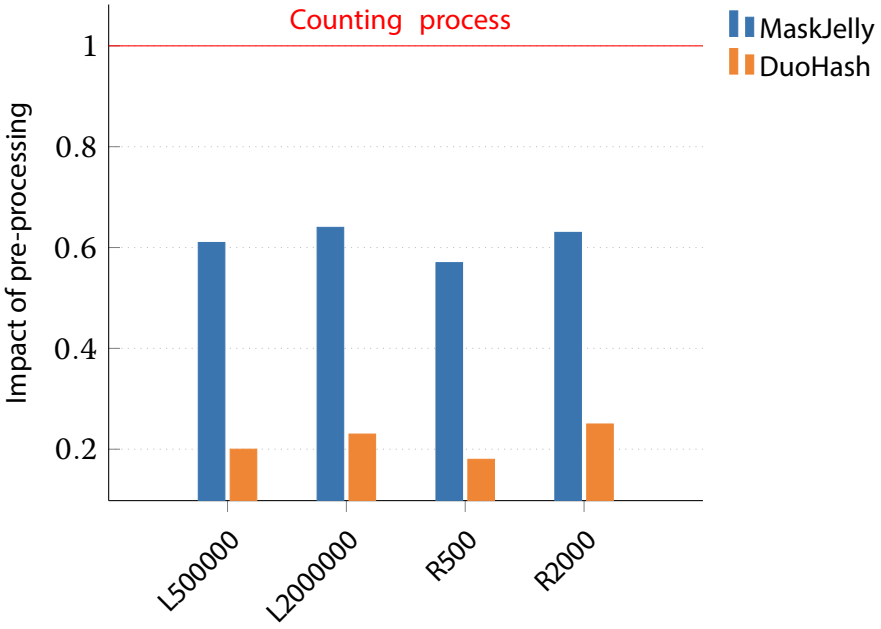


Figure 5.8: Impact of pre-processing (MaskJelly and DuoHash) on the overall counting process.

6

Conclusions

The DuoHash project led to the development of a highly efficient tool for hashing spaced *k*-mer for nucleotide sequences, significantly improving performance over existing tools such as ntHash2. Performance tests have shown that DuoHash offers a significant speed-up, with an observed maximum speed-up of about 11× compared to ntHash2. High speed-ups are particularly evident in datasets characterised by reads of constant length, where DuoHash showed marked superiority. These results were achieved thanks to significant innovations in the data architecture, such as the use of optimised hash structures and pre-computed look-up tables, which drastically improved the calculation performance. The flexibility of the system, with the possibility of easily modifying the hash function and handling both symmetric and asymmetric spaced seeds, has expanded the application potential of DuoHash, making it versatile for different types of genomic analysis. For instance, the implementation of the `getSpacedKmer` function and compatibility with JellyFish have made DuoHash a powerful tool for pre-processing nucleotide sequences for spaced *k*-mer counting. These contributions are significant for the field of bioinformatics, as they offer a solution that not only improves performance, but also expands analysis possibilities through a more flexible and integrated approach.

Along with the promising results, there are some aspects of DuoHash that could be further improved. In particular, the implementation of more advanced parallelisation techniques could further improve performance. Although DuoHash already supports parallel execution through the `DuoHash_parallel` method, there is

further scope for optimising this aspect, making better use of the available hardware resources and reducing computation times. One possible avenue could be the revision of the data-saving structure to allow the implementation of techniques such as Single Instruction Multiple Data (SIMD) instructions. Currently, the data storage structure of DuoHash does not allow the effective use of instructions. Rethinking the data architecture to align it with the requirements of instructions SIMD could therefore be a significant improvement. This could involve reorganising data into contiguous blocks in memory, optimised for parallel access and concurrent computation, thus minimising latency time and maximising throughput. These improvements would make DuoHash even more competitive and versatile, allowing it to handle ever larger datasets and to adapt to different research needs. In summary, while DuoHash already represents a significant step forward in the field of hashing genomic sequences, the adoption of advanced parallelisation techniques and the optimisation of parallel execution strategies are promising directions for the future development of DuoHash, which could lead to further performance improvements and greater efficiency in the analysis of genomic sequences.

A

Used Seedsets

In this chapter, all the seedsets used in the experiments in this study are described in detail. A full description of these spaced seed sets is provided in Chapter 5 starting on Page 50.

Seedset	Brief description
W10L15	this seedset contains spaced seeds of weight 10 and length 15
W14L31	this seedset contains seeds of weight 14 and length 31
W18L31	seedset with weight 18 and length 31
W22L31	includes spaced seed of weight 22 and length 31
W26L31	with weight 26 and length 31
W32L45	seedset with weight 32 and length 45

Q1	100111101111001
Q2	101101101101101
Q3	110011101110011
Q4	110110101011011
Q5	111001101100111
Q6	111011000110111
Q7	111100101001111
Q8	111101000101111
Q9	111110000011111

Table A.1: Seedset W10L15: spaced seeds of weight 10 and length 15.

Q1	1000101011100100010011101010001
Q2	1010000111100100010011110000101
Q3	1011001001001010101001001001101
Q4	1011010110000010100000110101101
Q5	1011100000100110110010000011101
Q6	1101111100000000000000011111011
Q7	1110000100110100010110010000111
Q8	1110010001000110110001000100111
Q9	1110110000110000000110000110111

Table A.2: Seedset W14L31: spaced seeds of weight 14 and length 31.

Q1	1111000011001110111001100001111
Q2	1101101001100110110011001011011
Q3	1011010011001110111001100101101
Q4	1010101011010110110101101010101
Q5	1100111010110010100110101110011
Q6	1100111011010010100101101110011
Q7	1101011010101100011010101101011
Q8	1100111011001100011001101110011
Q9	1100111010110100010110101110011

Table A.3: Seedset W18L31: spaced seeds of weight 18 and length 31.

Q1	1110111001101110111011001110111
Q2	1110111011101100011011101110111
Q3	1111011101110010100111011101111
Q4	1111010111011010101101110101111
Q5	1111011110011010101100111101111
Q6	1111101001011110111101001011111
Q7	1111101001110110110111001011111
Q8	1111101011100110110011101011111
Q9	1111110101101010101011010111111

Table A.4: Seedset W22L31: spaced seeds of weight 22 and length 31.

Q1	1111101110111110111110111011111
Q2	1110111111011110111101111110111
Q3	1111110111111010101111110111111
Q4	1111011111101110111011111101111
Q5	1101111101111110111111011111011
Q6	1011111111110110110111111111101
Q7	1111011111111100011111111101111
Q8	1111111101110110110111011111111
Q9	1101111111011110111101111111011

Table A.5: Seedset W26L31: spaced seeds of weight 26 and length 31.

Q1	1010111111000011111111011111111000011111110101
Q2	1011011101111011101101010110111011110111011101
Q3	1101110110111011011101010111011011101101110111
Q4	1110110111011011101101010110111011011101101110111
Q5	1111001111001111001111011110011110011110011111
Q6	1111011110111101111000000011110111101111011101111
Q7	1111100011111000111111101111110001111100011111
Q8	1111100100111111001111110111110011111100100111111
Q9	1111111100000011111111011111111000000011111111

Table A.6: Seedset W32L45: spaced seeds of weight 32 and length 45.

B

Additional Times Tables

In this chapter, all times of ntHash2 tool and the new DuoHash tool, obtained in the experiments conducted in the course of this study, are presented in detail. The datasets, already described in the Chapter 5 on pages 49 and following, are given again here. Times are expressed in milliseconds.

Dataset	Number of reads	Reads length
L500000	500,000	80
L1000000	1,000,000	80
L1500000	1,500,000	80
L2000000	2,000,000	80
L5000000	5,000,000	80
R80	500,000	80
R200	500,000	250
R350	500,000	350
R500	500,000	500
R1000	500,000	1,000
R1500	500,000	1,500
R2000	500,000	2,000
R5000	500,000	5,000

B.1 Times for the "L" group datasets

		single-seed				multiple-seed		
		ntHash2	naive	FSH	ISSH	ntHash2	naive	FSH
L500000	W10L15	1,006	262	231	228	7,688	2,195	1,956
	W14L31	1,294	260	252	224	10,273	2,257	2,203
	W18L31	1,912	317	290	233	15,010	2,791	2,568
	W22L31	1,292	372	310	218	10,360	3,262	2,833
	W26L31	1,303	433	274	222	8,919	3,810	2,546
	W32L45	1,127	384	252	190	9,041	3,240	2,146
L1000000	W10L15	2,058	507	451	445	15,504	4,407	3,879
	W14L31	2,554	510	496	400	20,286	4,388	4,296
	W18L31	3,877	623	569	415	30,288	5,439	5,048
	W22L31	2,545	738	614	433	20,380	6,469	5,677
	W26L31	2,606	868	546	443	18,225	7,589	5,091
	W32L45	2,203	753	491	372	17,569	6,338	4,218
L1500000	W10L15	2,988	755	672	662	22,814	6,509	5,819
	W14L31	3,773	776	754	611	30,076	6,577	6,461
	W18L31	5,781	932	863	717	46,806	8,141	7,627
	W22L31	3,797	1106	923	649	30,491	9,703	8,413
	W26L31	3,909	1282	814	660	27,182	11,250	7,618
	W32L45	3,305	1130	735	555	26,339	9,522	6,322
L2000000	W10L15	3,985	1,014	900	887	30,278	8,635	7,725
	W14L31	5,035	1,021	991	801	40,343	8,774	8,598
	W18L31	7,613	1,241	1,133	829	60,734	10,816	10,102
	W22L31	5,071	1,473	1,230	864	40,640	12,916	11,258
	W26L31	5,180	1,718	1,081	880	35,741	15,070	10,208
	W32L45	4,407	1,506	981	741	35,108	12,684	8,396
L5000000	W10L15	9,946	2,528	2,243	2,208	75,656	21,637	19,287
	W14L31	12,568	2,593	2,520	2,044	100,080	21,964	21,498
	W18L31	19,349	3,108	2,839	2,077	161,847	27,146	25,220
	W22L31	12,734	3,712	3,101	2,174	101,950	32,459	28,247
	W26L31	12,968	4,309	2,715	2,208	91,764	38,011	25,345
	W32L45	11,182	3,808	2,468	1,866	89,255	31,872	20,940

Table B.1: Overall time table (in milliseconds) for the "L" group datasets

multiple-seed					
ISSH	MFSH	DuoHash	DuoHash_col	DuoHash_par	DuoHash_row
1,922	2,002	1,453	1,803	13,286	1,967
1,744	2,298	1,502	1,694	13,650	2,211
1,824	2,498	1,450	1,666	13,550	1,803
1,879	2,667	1,536	1,912	13,301	1,971
1,896	2,582	1,526	1,792	13,421	1,968
1,479	2,269	1,359	1,528	13,408	1,608
3,817	3,983	2,874	3,596	26,522	3,922
3,393	4,525	3,008	3,339	26,471	4,324
3,534	4,971	2,881	3,264	26,544	3,581
3,688	5,320	3,020	3,734	25,974	3,924
3,769	5,142	3,034	3,594	26,456	3,891
2,896	4,466	2,681	2,951	25,465	3,171
5,688	5,931	4,307	5,338	38,712	5,851
5,097	6,833	4,475	5,033	39,489	6,495
5,306	7,445	4,324	4,879	39,642	5,368
5,534	7,944	4,520	5,606	39,561	5,906
5,620	7,702	4,532	5,346	39,122	5,864
4,350	6,732	4,020	4,433	38,168	4,778
7,590	7,881	5,719	7,092	51,671	7,797
6,793	9,044	5,904	6,677	52,750	8,736
7,057	9,895	5,752	6,492	52,263	7,106
7,368	10,561	6,054	7,539	52,190	7,843
7,520	10,405	6,055	7,118	52,067	7,755
5,802	8,781	5,356	5,900	50,818	6,381
18,938	19,720	14,367	17,738	128,635	19,475
16,990	22,572	14,885	16,600	131,708	21,399
17,642	24,716	14,410	16,244	135,462	17,884
18,460	26,402	15,175	18,786	131,871	19,692
18,866	25,815	15,386	17,930	131,846	19,785
14,568	22,149	13,486	14,843	128,820	15,911

B.2 Times for the "R" group datasets

		single-seed				multiple-seed		
		ntHash2	naive	FSH	ISSH	ntHash2	naive	FSH
R80	W10L15	1,006	262	231	228	7,688	2,195	1,956
	W14L31	1,294	260	252	224	10,273	2,257	2,203
	W18L31	1,912	317	290	233	15,010	2,791	2,568
	W22L31	1,292	372	310	218	10,360	3,262	2,833
	W26L31	1,303	433	274	222	8,919	3,810	2,546
	W32L45	1,127	384	252	190	9,041	3,240	2,146
R200	W10L15	2,463	684	596	607	19,121	5,992	5,208
	W14L31	3,589	833	762	634	29,558	7,471	6,816
	W18L31	4,326	1,021	885	671	36,040	9,161	7,964
	W22L31	3,562	1,218	986	697	29,198	10,950	9,188
	W26L31	2,552	1,415	849	714	20,057	12,785	7,830
	W32L45	3,629	1,528	900	688	29,996	13,659	8,344
R350	W10L15	4,285	1,231	1,062	1,093	33,489	10,866	9,515
	W14L31	6,504	1,551	1,405	1,191	53,676	13,870	12,712
	W18L31	7,898	1,908	1,634	1,270	65,747	17,090	14,799
	W22L31	6,396	2,280	1,829	1,317	52,895	20,469	17,116
	W26L31	4,555	2,654	1,586	1,346	36,011	23,845	14,664
	W32L45	6,781	2,969	1,732	1,353	56,618	26,675	15,877
R500	W10L15	6,709	1,760	1,516	1,569	48,854	16,015	13,816
	W14L31	9,552	2,276	2,058	1,750	78,972	20,785	18,820
	W18L31	11,754	2,790	2,383	1,855	97,277	25,492	21,687
	W22L31	9,379	3,353	2,689	1,939	77,650	31,123	25,613
	W26L31	6,607	3,900	2,330	1,981	52,587	35,814	21,543
	W32L45	10,184	4,449	2,567	2,008	85,117	39,945	23,516

Table B.2: Overall time table (in milliseconds) for the "R" group datasets - part one

multiple-seed					
ISSH	MFSH	DuoHash	DuoHash_col	DuoHash_par	DuoHash_row
1,922	2,002	1,453	1,803	13,286	1,967
1,744	2,298	1,502	1,694	13,650	2,211
1,824	2,498	1,450	1,666	13,550	1,803
1,879	2,667	1,536	1,912	13,301	1,971
1,896	2,582	1,526	1,792	13,421	1,968
1,479	2,269	1,359	1,528	13,408	1,608
5,316	5,477	3,866	4,930	15,327	5,274
5,622	7,743	4,650	5,475	17,205	6,822
5,937	7,740	4,618	5,414	16,096	5,720
6,207	8,698	4,801	6,276	16,587	6,337
6,333	7,925	4,869	5,949	16,683	6,402
6,100	9,055	5,163	6,135	16,863	6,459
9,731	9,990	6,999	8,915	18,974	9,599
10,765	14,706	8,654	10,213	21,302	12,627
11,391	14,638	8,683	10,211	20,720	10,772
11,930	16,303	8,940	11,866	21,317	11,951
12,053	14,892	9,068	11,128	21,482	12,010
12,236	17,515	10,028	12,027	22,179	12,733
14,245	14,752	10,642	13,047	21,691	14,147
15,973	22,178	13,550	15,251	24,747	18,797
16,888	21,608	12,989	15,146	23,868	16,103
17,921	24,306	14,100	18,103	25,637	17,845
17,979	22,121	13,681	16,765	26,192	17,998
18,292	25,548	15,603	18,059	27,493	18,939

		single-seed				multiple-seed		
		ntHash2	naive	FSH	ISSH	ntHash2	naive	FSH
R1000	W10L15	12,461	3,566	3,046	3,160	98,256	32,581	27,670
	W14L31	19,457	4,690	4,201	3,583	161,851	42,907	38,186
	W18L31	23,563	5,727	4,848	3,790	198,326	52,282	44,070
	W22L31	18,860	6,863	5,474	3,950	156,438	62,409	51,042
	W26L31	13,463	8,004	4,722	4,034	106,942	72,702	43,537
	W32L45	20,502	9,225	5,300	4,178	171,830	83,684	48,083
R1500	W10L15	18,379	5,371	4,586	4,763	144,493	48,867	41,351
	W14L31	28,996	7,058	6,318	5,392	240,301	64,419	57,081
	W18L31	35,609	8,751	7,410	5,785	298,499	79,488	66,868
	W22L31	28,494	10,508	8,372	6,048	239,908	95,355	77,387
	W26L31	20,022	12,167	7,176	6,142	159,079	110,578	65,780
	W32L45	31,274	14,115	8,086	6,367	261,604	127,294	72,944
R2000	W10L15	24,616	7,118	6,119	6,366	193,444	65,179	55,086
	W14L31	38,824	9,472	8,461	7,230	322,433	86,230	76,394
	W18L31	47,334	11,704	9,859	7,738	396,448	106,338	89,209
	W22L31	38,292	13,998	11,174	8,053	318,130	127,283	103,427
	W26L31	26,975	16,266	9,581	8,187	215,261	146,821	87,326
	W32L45	41,539	18,818	10,791	8,498	347,748	169,776	97,074
R5000	W10L15	60,992	18,498	15,739	16,417	479,736	165,640	139,518
	W14L31	98,330	24,513	21,966	18,768	817,744	219,369	194,687
	W18L31	118,888	29,948	25,302	19,868	989,897	269,583	224,841
	W22L31	95,118	35,891	28,565	20,721	789,914	324,546	261,281
	W26L31	67,074	41,596	24,545	21,162	532,221	375,486	221,502
	W32L45	105,520	48,697	27,848	22,088	885,443	472,203	247,809

Table B.3: Overall time table (in milliseconds) for the "R" group datasets - part two

			multiple-seed		
ISSH	**MFSH**	**DuoHash**	**DuoHash_col**	**DuoHash_par**	**DuoHash_row**
28,805	30,217	21,305	26,623	31,197	28,451
32,484	45,010	28,606	32,359	43,030	38,554
34,492	44,441	26,991	31,246	39,758	32,859
36,183	49,156	28,092	36,262	43,571	36,390
36,706	44,864	28,008	34,067	43,842	36,871
37,970	52,976	31,297	37,782	47,350	39,529
43,208	47,452	37,194	41,907	46,042	43,201
48,597	70,589	44,926	49,560	54,159	57,977
52,194	70,669	46,337	48,952	53,130	50,498
54,787	77,589	47,131	57,135	56,587	55,625
55,614	69,954	47,599	53,759	58,444	56,498
57,718	83,590	52,336	59,533	63,589	60,494
57,518	63,344	49,521	56,114	57,001	57,327
64,938	95,171	60,533	66,550	64,333	77,560
69,736	94,604	61,748	65,377	65,929	67,089
73,541	104,239	62,932	75,711	71,566	74,446
73,868	93,149	63,400	71,210	74,132	75,400
76,874	111,398	69,582	79,585	78,772	80,408
145,940	155,622	118,225	137,300	105,907	141,894
166,597	235,155	145,859	163,848	124,134	193,988
176,439	231,218	147,629	160,377	127,099	165,692
185,141	254,632	150,594	185,419	140,275	184,707
187,509	229,037	154,256	175,784	149,189	186,226
196,281	289,741	173,218	199,558	162,670	203,770

C

Additional Speed-up Tables

In this chapter, all speed-ups of the new DuoHash tool compared to the ntHash2 tool, obtained in the experiments conducted in the course of this study, are presented in detail. The datasets, already described in the Chapter 5 on pages 49 and following, are given again here.

Dataset	Number of reads	Reads length
L500000	500,000	80
L1000000	1,000,000	80
L1500000	1,500,000	80
L2000000	2,000,000	80
L5000000	5,000,000	80
R80	500,000	80
R200	500,000	250
R350	500,000	350
R500	500,000	500
R1000	500,000	1,000
R1500	500,000	1,500
R2000	500,000	2,000
R5000	500,000	5,000

C.1 Speed-up for the "L" group datasets

		single-seed			multiple-seed			
		naive	FSH	ISSH	naive	FSH	ISSH	MFSH
L500000	W10L15	3.84	4.36	4.41	3.50	3.93	4.00	3.84
	W14L31	4.97	5.13	5.78	4.55	4.66	5.89	4.47
	W18L31	6.03	6.60	8.20	5.38	5.85	8.23	6.01
	W22L31	3.47	4.16	5.92	3.18	3.66	5.51	3.88
	W26L31	3.01	4.76	5.87	2.34	3.50	4.70	3.45
	W32L45	2.93	4.47	5.95	2.79	4.21	6.11	3.98
L1000000	W10L15	4.06	4.56	4.63	3.52	4.00	4.06	3.89
	W14L31	5.01	5.15	6.39	4.62	4.72	5.98	4.48
	W18L31	6.22	6.81	9.35	5.57	6.00	8.57	6.09
	W22L31	3.45	4.14	5.88	3.15	3.59	5.53	3.83
	W26L31	3.00	4.78	5.88	2.40	3.58	4.84	3.54
	W32L45	2.92	4.49	5.93	2.77	4.17	6.07	3.93
L1500000	W10L15	3.96	4.45	4.52	3.50	3.92	4.01	3.85
	W14L31	4.86	5.01	6.17	4.57	4.66	5.90	4.40
	W18L31	6.21	6.70	8.06	5.75	6.14	8.82	6.29
	W22L31	3.43	4.11	5.85	3.14	3.62	5.51	3.84
	W26L31	3.05	4.80	5.92	2.42	3.57	4.84	3.53
	W32L45	2.93	4.50	5.95	2.77	4.17	6.05	3.91
L2000000	W10L15	3.93	4.43	4.49	3.51	3.92	3.99	3.84
	W14L31	4.93	5.08	6.28	4.60	4.69	5.94	4.46
	W18L31	6.14	6.72	9.19	5.62	6.01	8.61	6.14
	W22L31	3.44	4.12	5.87	3.15	3.61	5.52	3.85
	W26L31	3.02	4.79	5.89	2.37	3.50	4.75	3.44
	W32L45	2.93	4.49	5.95	2.77	4.18	6.05	4.00
L5000000	W10L15	3.93	4.43	4.50	3.50	3.92	3.99	3.84
	W14L31	4.85	4.99	6.15	4.56	4.66	5.89	4.43
	W18L31	6.23	6.82	9.31	5.96	6.42	9.17	6.55
	W22L31	3.43	4.11	5.86	3.14	3.61	5.52	3.86
	W26L31	3.01	4.78	5.87	2.41	3.62	4.86	3.55
	W32L45	2.94	4.53	5.99	2.80	4.26	6.13	4.03

Table C.1: Overall speed-up table for the "L" group datasets

multiple-seed			
DuoHash	**DuoHash_col**	**DuoHash_par**	**DuoHash_row**
5.29	4.26	0.58	3.91
6.84	6.07	0.75	4.65
10.35	9.01	1.11	8.32
6.74	5.42	0.78	5.26
5.84	4.98	0.66	4.53
6.65	5.91	0.67	5.62
5.40	4.31	0.58	3.95
6.74	6.07	0.77	4.69
10.51	9.28	1.14	8.46
6.75	5.46	0.78	5.19
6.01	5.07	0.69	4.68
6.55	5.95	0.69	5.54
5.30	4.27	0.59	3.90
6.72	5.98	0.76	4.63
10.82	9.59	1.18	8.72
6.75	5.44	0.77	5.16
6.00	5.08	0.69	4.64
6.55	5.94	0.69	5.51
5.29	4.27	0.59	3.88
6.83	6.04	0.76	4.62
10.56	9.36	1.16	8.55
6.71	5.39	0.78	5.18
5.90	5.02	0.69	4.61
6.56	5.95	0.69	5.50
5.27	4.27	0.59	3.88
6.72	6.03	0.76	4.68
11.23	9.96	1.19	9.05
6.72	5.43	0.77	5.18
5.96	5.12	0.70	4.64
6.62	6.01	0.69	5.61

C.2 Speed-up for the "R" group datasets

		single-seed			multiple-seed			
		naive	FSH	ISSH	naive	FSH	ISSH	MFSH
R80	W10L15	3.84	4.36	4.41	3.50	3.93	4.00	3.84
	W14L31	4.97	5.13	5.78	4.55	4.66	5.89	4.47
	W18L31	6.03	6.60	8.20	5.38	5.85	8.23	6.01
	W22L31	3.47	4.16	5.92	3.18	3.66	5.51	3.88
	W26L31	3.01	4.76	5.87	2.34	3.50	4.70	3.45
	W32L45	2.93	4.47	5.95	2.79	4.21	6.11	3.98
R200	W10L15	3.60	4.13	4.06	3.19	3.67	3.60	3.49
	W14L31	4.31	4.71	5.66	3.96	4.34	5.26	3.82
	W18L31	4.24	4.89	6.44	3.93	4.53	6.07	4.66
	W22L31	2.93	3.61	5.11	2.67	3.18	4.70	3.36
	W26L31	1.80	3.01	3.58	1.57	2.56	3.17	2.53
	W32L45	2.37	4.03	5.28	2.20	3.60	4.92	3.31
R350	W10L15	3.48	4.03	3.92	3.08	3.52	3.44	3.35
	W14L31	4.19	4.63	5.46	3.87	4.22	4.99	3.65
	W18L31	4.14	4.83	6.22	3.85	4.44	5.77	4.49
	W22L31	2.80	3.50	4.86	2.58	3.09	4.43	3.24
	W26L31	1.72	2.87	3.38	1.51	2.46	2.99	2.42
	W32L45	2.28	3.91	5.01	2.12	3.57	4.63	3.23
R500	W10L15	3.81	4.43	4.28	3.05	3.54	3.43	3.31
	W14L31	4.20	4.64	5.46	3.80	4.20	4.94	3.56
	W18L31	4.21	4.93	6.34	3.82	4.49	5.76	4.50
	W22L31	2.80	3.49	4.84	2.49	3.03	4.33	3.19
	W26L31	1.69	2.84	3.34	1.47	2.44	2.92	2.38
	W32L45	2.29	3.97	5.07	2.13	3.62	4.65	3.33

Table C.2: Overall speed-up table for the "R" group datasets - part one

multiple-seed			
DuoHash	**DuoHash_col**	**DuoHash_par**	**DuoHash_row**
5.29	4.26	0.58	3.91
6.84	6.07	0.75	4.65
10.35	9.01	1.11	8.32
6.74	5.42	0.78	5.26
5.84	4.98	0.66	4.53
6.65	5.91	0.67	5.62
4.95	3.88	1.25	3.63
6.36	5.40	1.72	4.33
7.80	6.66	2.24	6.30
6.08	4.65	1.76	4.61
4.12	3.37	1.20	3.13
5.81	4.89	1.78	4.64
4.78	3.76	1.77	3.49
6.20	5.26	2.52	4.25
7.57	6.44	3.17	6.10
5.92	4.46	2.48	4.43
3.97	3.24	1.68	3.00
5.65	4.71	2.55	4.45
4.59	3.74	2.25	3.45
5.83	5.18	3.19	4.20
7.49	6.42	4.08	6.04
5.51	4.29	3.03	4.35
3.84	3.14	2.01	2.92
5.46	4.71	3.10	4.49

		single-seed			multiple-seed			
		naive	FSH	ISSH	naive	FSH	ISSH	MFSH
R1000	W10L15	3.49	4.09	3.94	3.02	3.55	3.41	3.25
	W14L31	4.15	4.63	5.43	3.77	4.24	4.98	3.60
	W18L31	4.11	4.86	6.22	3.79	4.50	5.75	4.46
	W22L31	2.75	3.45	4.77	2.51	3.06	4.32	3.18
	W26L31	1.68	2.85	3.34	1.47	2.46	2.91	2.38
	W32L45	2.22	3.87	4.91	2.05	3.57	4.53	3.24
R1500	W10L15	3.42	4.01	3.86	2.96	3.49	3.34	3.05
	W14L31	4.11	4.59	5.38	3.73	4.21	4.94	3.40
	W18L31	4.07	4.81	6.16	3.76	4.46	5.72	4.22
	W22L31	2.71	3.40	4.71	2.52	3.10	4.38	3.09
	W26L31	1.65	2.79	3.26	1.44	2.42	2.86	2.27
	W32L45	2.22	3.87	4.91	2.06	3.59	4.53	3.13
R2000	W10L15	3.46	4.02	3.87	2.97	3.51	3.36	3.05
	W14L31	4.10	4.59	5.37	3.74	4.22	4.97	3.39
	W18L31	4.04	4.80	6.12	3.73	4.44	5.68	4.19
	W22L31	2.74	3.43	4.75	2.50	3.08	4.33	3.05
	W26L31	1.66	2.82	3.30	1.47	2.47	2.91	2.31
	W32L45	2.21	3.85	4.89	2.05	3.58	4.52	3.12
R5000	W10L15	3.30	3.88	3.72	2.90	3.44	3.29	3.08
	W14L31	4.01	4.48	5.24	3.73	4.20	4.91	3.48
	W18L31	3.97	4.70	5.98	3.67	4.40	5.61	4.28
	W22L31	2.65	3.33	4.59	2.43	3.02	4.27	3.10
	W26L31	1.61	2.73	3.17	1.42	2.40	2.84	2.32
	W32L45	2.17	3.79	4.78	1.88	3.57	4.51	3.06

Table C.3: Overall speed-up table for the "R" group datasets - part two

multiple-seed			
DuoHash	**DuoHash_col**	**DuoHash_par**	**DuoHash_row**
4.61	3.69	3.15	3.45
5.66	5.00	3.76	4.20
7.35	6.35	4.99	6.04
5.57	4.31	3.59	4.30
3.82	3.14	2.44	2.90
5.49	4.55	3.63	4.35
3.88	3.45	3.14	3.34
5.35	4.85	4.44	4.14
6.44	6.10	5.62	5.91
5.09	4.20	4.24	4.31
3.34	2.96	2.72	2.82
5.00	4.39	4.11	4.32
3.91	3.45	3.39	3.37
5.33	4.85	5.01	4.16
6.42	6.06	6.01	5.91
5.06	4.20	4.45	4.27
3.40	3.02	2.90	2.85
5.00	4.37	4.41	4.32
4.06	3.49	4.53	3.38
5.61	4.99	6.59	4.22
6.71	6.17	7.79	5.97
5.25	4.26	5.63	4.28
3.45	3.03	3.57	2.86
5.11	4.44	5.44	4.35

D

Additional Speed-up Graphs

In this chapter, all speed-ups of the new DuoHash tool compared to the ntHash2 tool, obtained in the experiments conducted in the course of this study, are presented in detail. Datasets and seedsets are already described in the Chapter 5 on pages 49 and following.

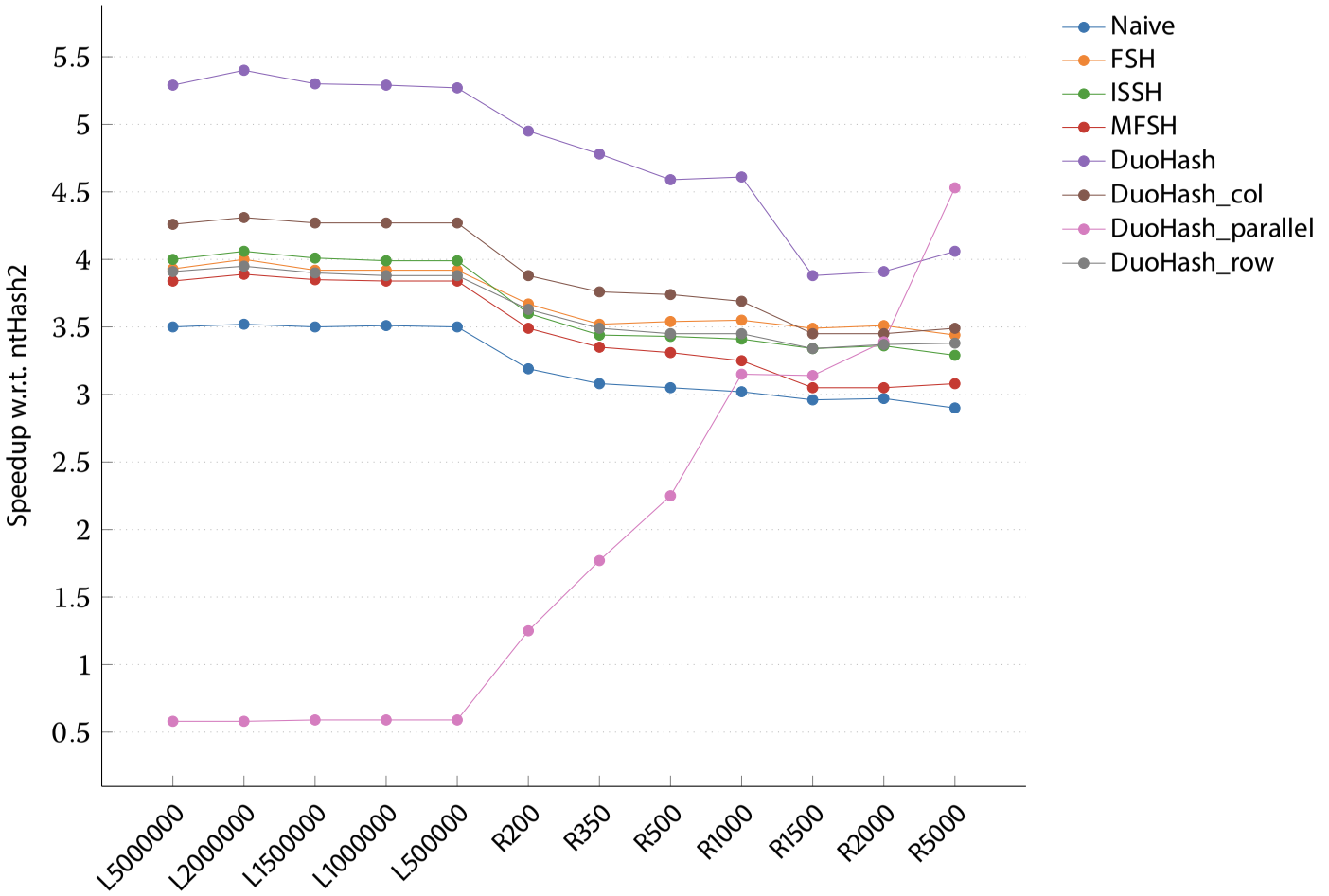


Figure D.1: Speed-up graph for seedset W10L15 (multiple-seed).

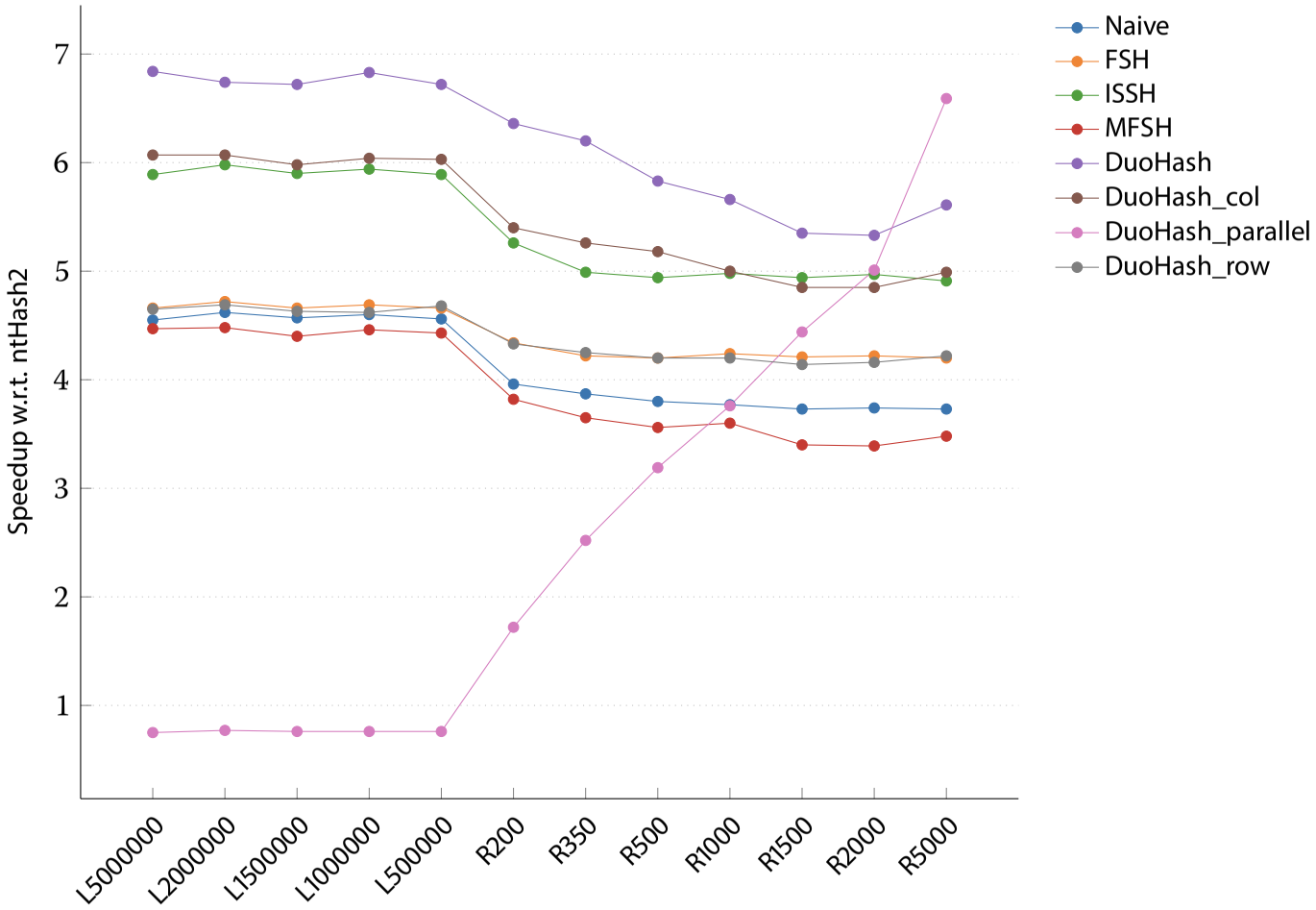


Figure D.2: Speed-up graph for seedset W14L31 (multiple-seed).

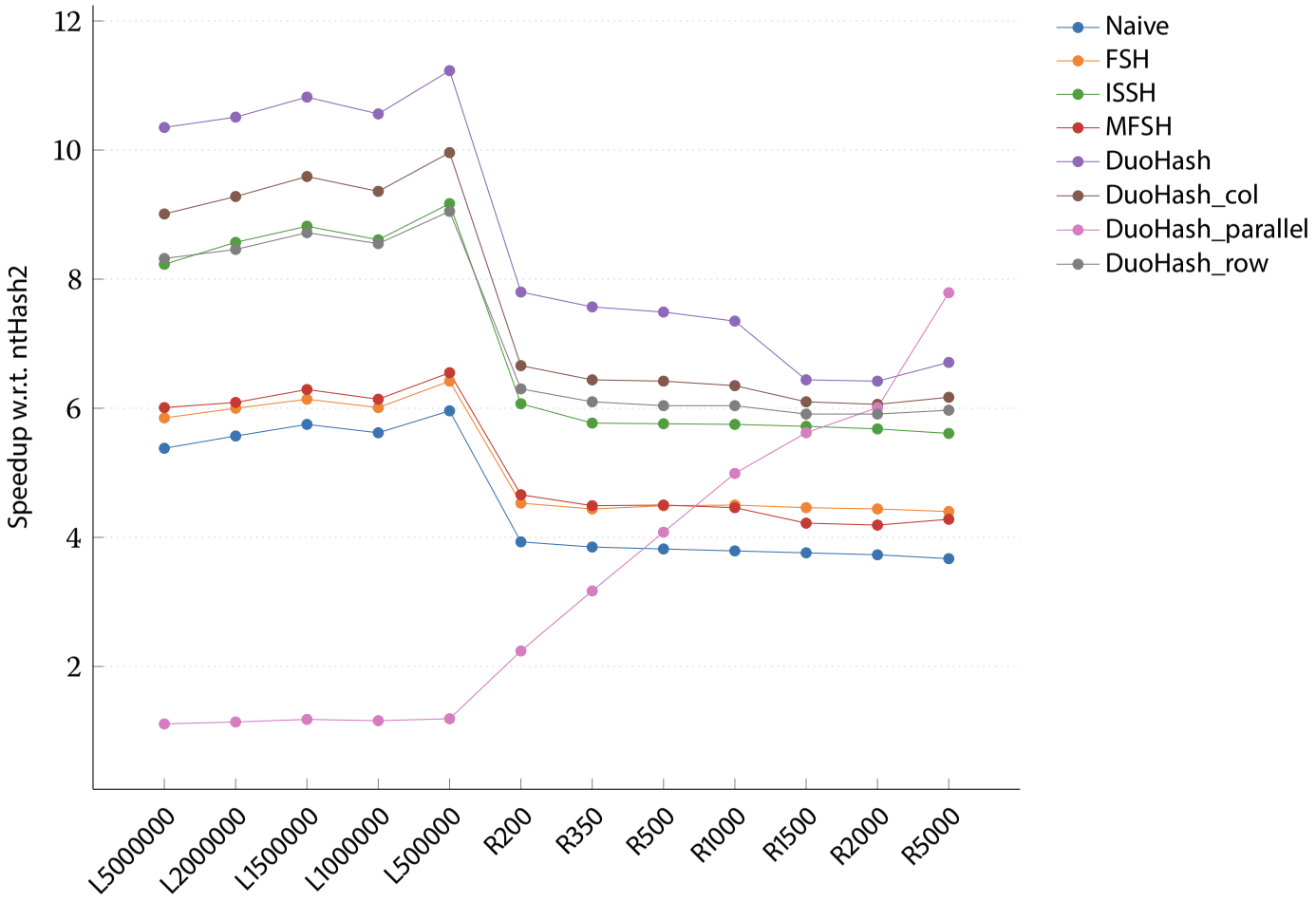


Figure D.3: Speed-up graph for seedset W18L31 (multiple-seed).

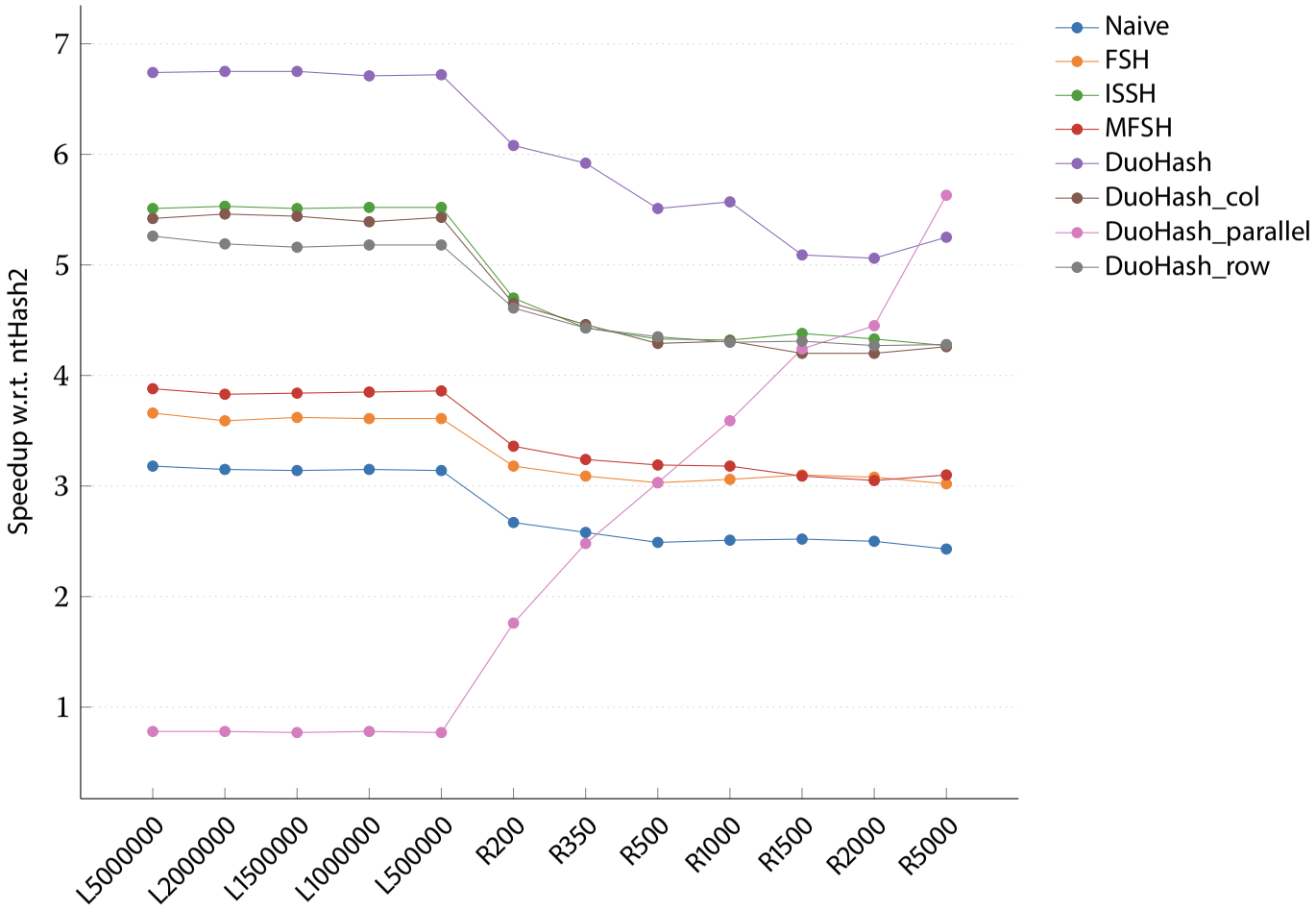


Figure D.4: Speed-up graph for seedset W22L31 (multiple-seed).

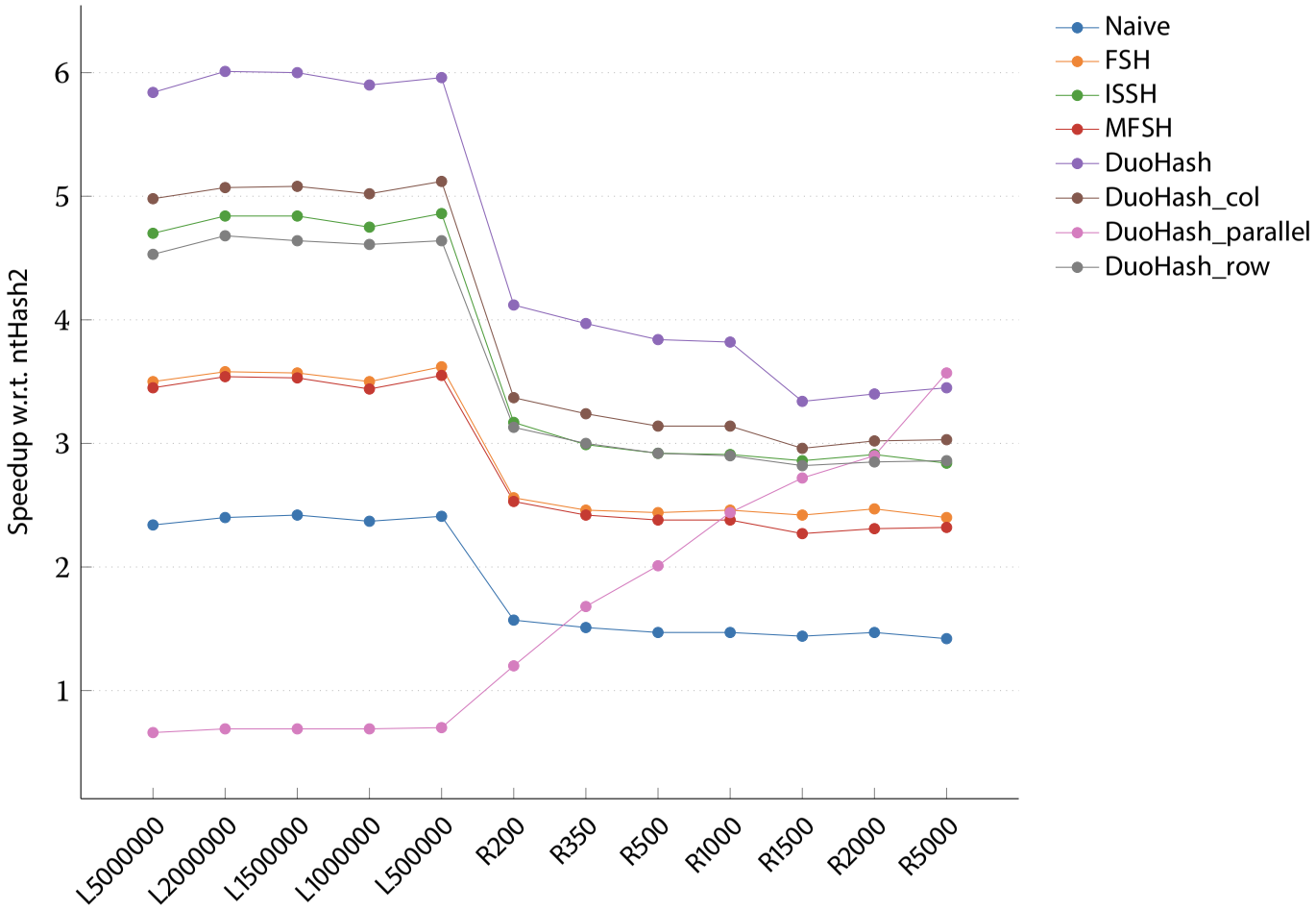


Figure D.5: Speed-up graph for seedset W26L31 (multiple-seed).

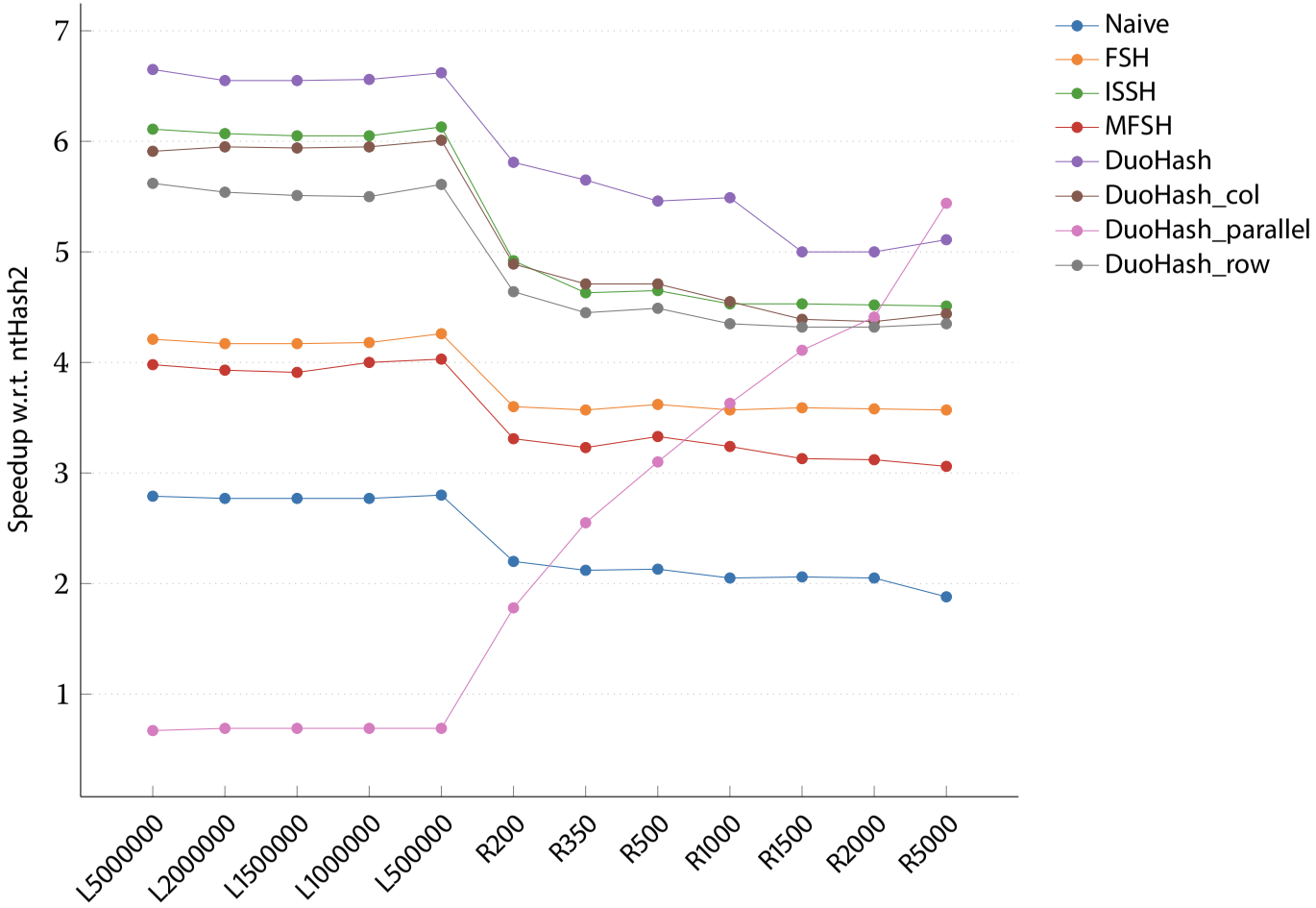


Figure D.6: Speed-up graph for seedset W32L45 (multiple-seed).

E

DuoHash in JellyFish context

In this chapter, all times and speed-ups of MaskJelly, DuoHash, and JellyFish tools, obtained in the experiments conducted in the course of this study, are presented in detail. The seedsets, already described in the Chapter 5 on pages 50 and following, are given again here. Times are expressed in milliseconds. Speed-ups are relative to DuoHash compared to MaskJelly.

	W22L31
Q1	111011100110111011101100111 0111
Q2	111011101110110001101110111 0111
Q3	111011101110010100111011101111
Q4	111010111011010101101110101111
Q5	111011111001101010110011110 1111
Q6	111110100101111011110100101 1111
Q7	111110100111011011011001011111
Q8	111110101110011011001110101 1111
Q9	111111010110101010110101 11111

			Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
L500000	MaskJelly	pre-processing	18.68	19.14	18.87	19.01	19.19	19.04	18.91	18.86	19.27
		JellyFish	11.64	12.05	11.88	12.07	11.97	11.84	12.05	11.79	12.08
	DuoHash	pre-processing	2.95	2.94	3.03	2.93	2.97	2.90	2.90	2.87	3.01
		JellyFish	11.70	11.80	11.92	12.15	11.88	11.84	12.08	11.81	11.89
L2000000	MaskJelly	pre-processing	76.67	76.94	76.65	77.04	76.72	77.83	77.21	75.99	79.43
		JellyFish	43.50	43.04	43.11	42.85	43.61	48.34	45.48	43.54	43.53
	DuoHash	pre-processing	13.45	12.91	13.24	12.84	12.75	14.06	13.00	12.89	12.77
		JellyFish	43.51	43.61	43.18	43.45	43.10	43.29	43.99	44.40	43.30
R500	MaskJelly	pre-processing	176.34	175.09	174.93	176.08	175.62	175.69	175.48	173.91	179.47
		JellyFish	143.86	116.83	117.12	117.47	140.13	141.95	141.87	139.55	140.97
	DuoHash	pre-processing	30.28	29.22	29.24	29.36	29.18	29.93	29.45	29.65	29.63
		JellyFish	140.63	117.57	117.46	117.13	140.45	140.77	140.60	140.38	140.15
R2000	MaskJelly	pre-processing	737.58	740.64	738.05	735.94	734.73	733.04	744.56	727.22	745.75
		JellyFish	436.99	439.43	439.06	437.14	436.08	433.05	439.61	432.22	433.03
	DuoHash	pre-processing	142.74	143.42	142.08	140.15	142.29	142.51	140.72	143.75	140.89
		JellyFish	436.48	444.93	435.42	435.79	436.02	433.53	437.57	433.61	435.99

Table E.1: Overall times (in milliseconds) table for MaskJelly and DuoHash pre-processing followed by JellyFish counting.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
L500000	pre-processing speed-up	6.33	6.51	6.23	6.49	6.46	6.57	6.52	6.57	6.40
	counting process speed-up	2.07	2.12	2.06	2.06	2.10	2.09	2.07	2.09	2.10
L2000000	pre-processing speed-up	5.70	5.96	5.79	6.00	6.02	5.54	5.94	5.90	6.22
	counting process speed-up	2.11	2.12	2.12	2.13	2.15	2.20	2.15	2.09	2.19
R500	pre-processing speed-up	5.82	5.99	5.98	6.00	6.02	5.87	5.96	5.87	6.06
	counting process speed-up	1.87	1.99	1.99	2.00	1.86	1.86	1.87	1.84	1.89
R2000	pre-processing speed-up	5.17	5.16	5.19	5.25	5.16	5.14	5.29	5.06	5.29
	counting process speed-up	2.03	2.01	2.04	2.04	2.02	2.02	2.05	2.01	2.04

Table E.2: Overall speed-ups table for MaskJelly and DuoHash pre-processing followed by entire counting process (pre-processing and JellyFish).

References

[1] Frederick Sanger, Steven Nicklen, and Alan R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences of the United States of America* 74.12 (Dec. 1977), pp. 5463–5467. DOI: 10.1073/pnas.74.12.5463. eprint: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC431765/pdf/pnas00043-0271.pdf. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC431765/.

[2] Bin Ma, John Tromp, and Ming Li. "PatternHunter: faster and more sensitive homology search". In: *Bioinformatics* 18.3 (Mar. 2002), pp. 440–445. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/18.3.440. eprint: https://academic.oup.com/bioinformatics/article-pdf/18/3/440/48850317/bioinformatics_18_3_440.pdf. URL: https://doi.org/10.1093/bioinformatics/18.3.440.

[3] Kelly A Frazer, Lior Pachter, Alexander Poliakov, Edward M Rubin, and Inna Dubchak. "VISTA: computational tools for comparative genomics". In: *Nucleic Acids Research* 32.suppl_2 (2004), W273–W279. DOI: 10.1093/nar/gkh458.

[4] Uri Keich, Ming Li, Bin Ma, and John Tromp. "On spaced seeds for similarity search". In: *Discrete Applied Mathematics* 138.3 (2004), pp. 253–263. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(03)00382-2. URL: https://www.sciencedirect.com/science/article/pii/S0166218X03003822.

[5] Ming Li, Bin Ma, Daniel Kisman, and John Tromp. "PatternHunter II: highly sensitive and fast homology search". In: *Journal of Bioinformatics and Computational Biology* 2.03 (2004), pp. 417–439.

[6] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. *Fowler-Noll-Vo (FNV) Hash Function*. Accessed: 2024-06-12. 2005. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[7] Yanni Sun and Jeremy Buhler. "Enhanced sensitivity of probabilistic sequence alignment using spaced seeds". In: *Journal of Computational Biology* 12.6 (2005), pp. 847–861.

[8] Stefan Burkhardt and Juha Kärkkäinen. "Enhanced sensitivity of short DNA sequence alignment through spaced seeds". In: *BMC bioinformatics* 7.1 (2006), pp. 1–11.

[9] David Mak and Gary Benson. "Improvements in Spaced Seed Design for DNA Similarity Search". In: *Bioinformatics* 22.14 (2006), pp. 1653–1659.

[10] Peter J Turnbaugh, Ruth E Ley, Micah Hamady, Claire M Fraser-Liggett, Rob Knight, and Jeffrey I Gordon. "The human microbiome project". In: *Nature* 449.7164 (2007), pp. 804–810.

[11] Jay Shendure and Hanlee Ji. "Next-generation DNA sequencing". In: *Nature Biotechnology* 26.10 (Oct. 2008), pp. 1135–1145. ISSN: 1546-1696. DOI: 10.1038/nbt1486. URL: https://doi.org/10.1038/nbt1486.

[12] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. "BLAST+: architecture and applications". In: *BMC Bioinformatics* 10.1 (2009), p. 421.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. Cambridge, MA: MIT Press, 2009, pp. 990–994.

[14] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25.14 (2009), pp. 1754–1760. DOI: 10.1093/bioinformatics/btp324.

[15] Niranjan Nagarajan and Mihai Pop. "Parametric complexity of sequence assembly: theory and applications to next generation sequencing". In: *Journal of Computational Biology* 16.7 (2009), pp. 897–908.

[16] Phillip E. Compeau, Pavel A. Pevzner, and Glenn Tesler. "How to apply de Bruijn graphs to genome assembly". In: *Nature Biotechnology* 29.11 (2011), pp. 987–991. DOI: 10.1038/nbt.2023.

[17] Guillaume Marçais and Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (Jan. 2011), pp. 764–770. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr011. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/6/764/48866141/bioinformatics_27_6_764.pdf. URL: https://doi.org/10.1093/bioinformatics/btr011.

[18] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. "Genome graphs and the evolution of genome inference". In: *Genome Research* 27.5 (2011), pp. 665–676.

[19] Sergey A Chumakov and Mikhail P Ponomarenko. "Computational challenges in the analysis of sequence data using spaced seeds". In: *Bioinformatics and Computational Biology* 8.3 (2012), pp. 194–206.

[20] Lucian Ilie and Martin Molnar. "RACER: Rapid and accurate correction of errors in reads". In: *Bioinformatics* 29.19 (2013), pp. 2490–2493. DOI: 10.1093/bioinformatics/btt426.

[21] Niranjan Nagarajan and Mihai Pop. "Sequence assembly demystified". In: *Nature Reviews Genetics* 14.3 (2013), pp. 157–167. DOI: 10.1038/nrg3367.

[22] Rayan Chikhi and Paul Medvedev. "Informed and automated *k*-mer size selection for genome assembly". In: *Bioinformatics* 30.1 (2014), pp. 31–37. DOI: 10.1093/bioinformatics/btt310.

[23] Chris-Andre Leimeister, Marcus Boden, Sebastian Horwege, Sebastian Lindner, and Burkhard Morgenstern. "Fast alignment-free sequence comparison using spaced-word frequencies". In: *Bioinformatics* 30.14 (Apr. 2014), pp. 1991–1999. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu177. eprint: https://academic.oup.com/bioinformatics/

article-pdf/30/14/1991/48925419/bioinformatics_30_14_1991.pdf. URL: https://doi.org/10.1093/bioinformatics/btu177.

[24] Derrick E. Wood and Steven L. Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biology* 15.3 (Mar. 2014), R46. ISSN: 1474-760X. DOI: 10.1186/gb-2014-15-3-r46. URL: https://doi.org/10.1186/gb-2014-15-3-r46.

[25] K. Brinda, M. Sykulski, and G. Kucherov. "Spaced Seeds Improve k-mer-based Metagenomic Classification". In: *Bioinformatics* 31.22 (Nov. 2015). Epub 2015 Jul 25, pp. 3584–3592. DOI: 10.1093/bioinformatics/btv419.

[26] Ana Conesa, Paula Madrigal, Sonia Tarazona, David Gomez-Cabrero, Amelia Cervera, Andrew McPherson, Mateusz W Szcześniak, Daniel J Gaffney, Laura L Elo, Xuegong Zhang, and Ali Mortazavi. "A survey of best practices for RNA-seq data analysis". In: *Genome Biology* 17.1 (2016), p. 13. DOI: 10.1186/s13059-016-0881-8.

[27] Lars Hahn, Chris-André Leimeister, Rachid Ounit, Stefano Lonardi, and Burkhard Morgenstern. "rasbhari: Optimizing Spaced Seeds for Database Searching, Read Mapping and Alignment-Free Sequence Comparison". In: *PLOS Computational Biology* 12.10 (Oct. 2016), pp. 1–18. DOI: 10.1371/journal.pcbi.1005107. URL: https://doi.org/10.1371/journal.pcbi.1005107.

[28] OpenStax Microbiology. *Microbiology ID: e42bd376-624b-4c0f-972f-e0c57998e765@4.4*. Version 4.4. Creative Commons Attribution License (by 4.0). Nov. 2016. URL: https://commons.wikimedia.org/wiki/File:OSC_Microbio_10_02_DoubHelix.jpg.

[29] Hamid Mohamadi, Justin Chu, Benjamin P Vandervalk, and Inanc Birol. "ntHash: recursive nucleotide hashing". In: *Bioinformatics* 32.22 (July 2016), pp. 3492–3494. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw245. eprint: https://academic.oup.com/bioinformatics/article-pdf/32/12/i243/49020222/bioinformatics_32_12_

i243.pdf. URL: https://doi.org/10.1093/bioinformatics/btw245.

[30] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. "Mash: fast genome and metagenome distance estimation using MinHash". In: *Genome Biology* 17.1 (June 2016), p. 132. ISSN: 1474-760X. DOI: 10.1186/s13059-016-0997-x. URL: https://doi.org/10.1186/s13059-016-0997-x.

[31] Rachid Ounit and Stefano Lonardi. "Higher classification sensitivity of short metagenomic reads with CLARK-S". In: *Bioinformatics* 32.24 (Aug. 2016), pp. 3823–3825. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw542. eprint: https://academic.oup.com/bioinformatics/article-pdf/32/24/3823/49027269/bioinformatics_32_24_3823.pdf. URL: https://doi.org/10.1093/bioinformatics/btw542.

[32] Samuele Girotto, Matteo Comin, and Cinzia Pizzi. "Metagenomic reads binning with spaced seeds". In: *Theoretical Computer Science* 698 (2017). Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo), pp. 88–99. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2017.05.023. URL: https://www.sciencedirect.com/science/article/pii/S0304397517304632.

[33] Yandong Li, Yi Zhang, and Dong Xu. "Efficient *k*-mer counting using a bloom filter and de Bruijn graph". In: *Journal of Computational Biology* 24.6 (2017), pp. 487–497. DOI: 10.1089/cmb.2016.0186.

[34] Samuele Girotto, Matteo Comin, and Cinzia Pizzi. "FSH: fast spaced seed hashing exploiting adjacent hashes". In: *Algorithms for Molecular Biology* 13.1 (Mar. 2018), p. 8. ISSN: 1748-7188. DOI: 10.1186/s13015-018-0125-4. URL: https://doi.org/10.1186/s13015-018-0125-4.

[35] Enrico Petrucci, Laurent Noé, Cinzia Pizzi, and Matteo Comin. "Iterative Spaced Seed Hashing: Closing the Gap Between Spaced Seed Hashing and k-mer Hashing". In: *Journal of Computational Biology* 27.2 (2020). PMID:

31800307, pp. 223–233. DOI: 10.1089/cmb.2019.0298. eprint: https://doi.org/10.1089/cmb.2019.0298. URL: https://doi.org/10.1089/cmb.2019.0298.

[36] Jan Ebler, Peter Ebert, William E Clarke, et al. "Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes". In: *Nature Genetics* 54.4 (2022), pp. 518–525. DOI: 10.1038/s41588-022-01043-w. URL: https://doi.org/10.1038/s41588-022-01043-w.

[37] Parham Kazemi, Johnathan Wong, Vladimir Nikolić, Hamid Mohamadi, René L Warren, and Inanç Birol. "ntHash2: recursive spaced seed hashing for nucleotide sequences". In: *Bioinformatics* 38.20 (Aug. 2022), pp. 4812–4813. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btac564. eprint: https://academic.oup.com/bioinformatics/article-pdf/38/20/4812/46535020/btac564.pdf. URL: https://doi.org/10.1093/bioinformatics/btac564.

[38] National Human Genome Research Institute. *The Cost of Sequencing a Human Genome*. Accessed: 2024-05-24. 2023. URL: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[39] Eleonora Mian, Enrico Petrucci, Cinzia Pizzi, and Matteo Comin. "Efficient Hashing of Multiple Spaced Seeds with Application". In: *Proceedings of the 16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2023) - BIOINFORMATICS*. INSTICC. SciTePress, 2023, pp. 155–162. ISBN: 978-989-758-631-6. DOI: 10.5220/0011632900003414.

[40] Hetal Satam, Kavita Joshi, Umang Mangrolia, Shraddha Waghoo, Gulzar Zaidi, Shwetali Rawool, Rahul P. Thakare, Suhail Banday, Ashutosh K. Mishra, Gautam Das, and Sandeep K. Malonia. "Next-Generation Sequencing Technology: Current Trends and Advancements". In: *Biology (Basel)* 12.7 (July 2023). Erratum in: Biology (Basel). 2024 Apr 24;13(5): p. 997. DOI: https://doi.org/10.3390/biology12070997.

Acknowledgments

At the end of this academic journey, I would like to express my deepest gratitude to mum and dad and my grandparents, my endless thanks for their unconditional love and support. Thank you for teaching me the value of commitment, dedication and sacrifice. You have always believed in my abilities, even when I myself doubted that I could do it. Your constant presence and words of encouragement have been an inexhaustible source of strength for me.

To my brother Giacomo, thank you for always being by my side. Your presence and support have been crucial, especially in the most difficult moments.

To all my family, for their constant affection and support. Thank you for believing in me and always encouraging me to follow my dreams. You have been a fundamental pillar in this journey.

To my friends, whose friendship has made this journey lighter and more joyful. Thank you for the laughter, the advice and for being a source of positive energy.

To my lecturer, Prof. Matteo Comin, a heartfelt thank you for the guidance and helpfulness he has always shown towards me.

To everyone, a heartfelt thank you.