
Estruturas de Dados Compactas para o Vetor de Sufixos Métrico

Frederico Rezende Rosa



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
2024

Frederico Rezende Rosa

**Estruturas de Dados Compactas para o Vetor
de Sufixos Métrico**

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Humberto Luiz Razente

Coorientador: Felipe Alves da Louza

Uberlândia

2024

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

R788 2024	<p>Rosa, Frederico Rezende, 1982- Estruturas de Dados Compactas para o Vetor de Sufixos Métrico [recurso eletrônico] / Frederico Rezende Rosa. - 2024.</p> <p>Orientador: Humberto Luiz Razente. Coorientador: Felipe Alves da Louza. Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação. Modo de acesso: Internet. Disponível em: http://doi.org/10.14393/ufu.di.2023.656 Inclui bibliografia. Inclui ilustrações.</p> <p>1. Computação. I. Razente, Humberto Luiz, 1977-, (Orient.). II. Louza, Felipe Alves da, 1988-, (Coorient.). III. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. IV. Título.</p> <p style="text-align: right;">CDU: 681.3</p>
--------------	---

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da
Computação

Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG,
CEP 38400-902

Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br



ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação de Mestrado, 3/2024, PPGCO				
Data:	19 de janeiro de 2024	Hora de início:	14:00	Hora de encerramento:	16:30
Matrícula do Discente:	12112CCP012				
Nome do Discente:	Frederico Rezende Rosa				
Título do Trabalho:	Estruturas de Dados Compactas para o Vetor de Sufixos Métrico				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Ciência de Dados				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Felipe Alves da Louza- FEELT/UFU, Bruno Augusto Nassif Travençolo- FACOM/UFU, Mayron César de Oliveira Moreira - DCC/UFLA e Humberto Luiz Razente- FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Mayron César de Oliveira Moreira - Lavras / MG . Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof . Dr. Humberto Luiz Razente, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos,

conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.



Documento assinado eletronicamente por **Bruno Augusto Nassif Travençolo, Professor(a) do Magistério Superior**, em 22/01/2024, às 09:18, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Humberto Luiz Razente, Professor(a) do Magistério Superior**, em 22/01/2024, às 09:57, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Felipe Alves da Louza, Professor(a) do Magistério Superior**, em 23/01/2024, às 07:40, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Mayron César de Oliveira Moreira, Usuário Externo**, em 24/01/2024, às 10:48, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5075734** e o código CRC **3477FFCF**.

Agradecimentos

À Capes, CNPq (CNPQ Universal 406418/2021-7) e FAPEMIG que possibilitaram a realização do projeto no qual este trabalho está inserido.

Resumo

A busca por similaridade aproximada tem sido usada em diversas disciplinas, como reconhecimento de padrões e aprendizado de máquina, e em aplicações como buscas de imagens, strings e genoma. Geralmente, essas atividades lidam com um grande volume de dados de alta dimensão, sendo relevantes tanto o tempo de execução das buscas quanto o tamanho da memória alocada pela estrutura de dados que responde a essas buscas. A busca por similaridade aproximada é realizada por meio de elementos de referência, que estabelecem um compromisso entre o nível de precisão das buscas e o tempo necessário e memória alocada. Utilizando esta técnica, propomos uma estrutura que opera busca por similaridade aproximada com uma estrutura de dados compacta que ainda apresenta um custo linear para construção e busca, e que não se limita a dados de 32 bits. Realizado os experimentos, conseguimos obter um método que requer menos memória, atingindo $1/3$ da memória requerida pelo método MSA, ao custo de um aumento no tempo de construção e busca, demandando até 2,7 e 3,5 o tempo do MSA respectivamente no melhor caso.

Palavras-chave: Busca por similaridade. Estrutura de dados compacta. Vetor de sufixos métrico.

Compact data structures for the Metric Suffix Array

Frederico Rezende Rosa



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO – FACOM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA
COMPUTAÇÃO – PPGCO

The undersigned hereby certify they have read and recommend to the PPGCO for acceptance the dissertation entitled “**Compact data structures for the Metric Suffix Array**” submitted by “**Frederico Rezende Rosa**” as part of the requirements for obtaining the **Master’s degree in Computer Science**.

Uberlândia, ___ de _____ de _____

Supervisor: _____

Prof. Dr. Humberto Luiz Razente
Universidade Federal de Uberlândia

Cosupervisor: _____

Prof. Dr. Felipe Alves da Louza
Universidade Federal de Uberlândia

Examining Committee Members:

Prof. Dr. Bruno Augusto Nassif Travençolo
Universidade Federal de Uberlândia

Prof. Dr. Mayron César de Oliveira Moreira
Universidade Federal de Lavras

Abstract

Approximate similarity searching has been used in several disciplines such as pattern recognition and machine learning, and applications such as image, strings and genome searches. Generally, these activities deal with a large volume of high-dimensional data, with both the execution time of the searches and the size of the memory allocated by the data structure that responds to these searches being relevant. The approximate similarity searching is carried out using reference elements, which establish a compromise between the level of precision of the searches and the time required and allocated memory. Using this technique, we propose a structure that operates approximate similarity searching with a compact data structure that still presents a linear cost for construction and search, and that is not limited to 32-bit data. With the experiments executed, we managed to obtain a method that requires less memory, achieving 1/3 of the memory required for the MSA, at the cost of an increase in construction and search time, demanding 2.7 and 3.5 the time required for the MSA respectively in the best case.

Keywords: Similarity searching. Compact data structures. Metric Suffix Array.

List of Figures

Figure 1 – Similarity searches. (a) Range query. (b) Nearest neighbor query	14
Figure 2 – Distance of the references set from object o_0	15
Figure 3 – Distance of the references set from object o_3	16
Figure 4 – Distance of the references set from object o_4	16
Figure 5 – Distance of the references set from query q	17
Figure 6 – Example of SFD value calculation for object o_0	18
Figure 7 – The suffix array of the string $S = 210210210102012120012012\$$	20
Figure 8 – String S as a character sequence showing the position indexes of each character	20
Figure 9 – Distribution of indexes for object o_0	22
Figure 10 – Distribution of indexes for object o_1	22
Figure 11 – Distribution of indexes for object o_2	22
Figure 12 – Distribution of indexes for object o_3	22
Figure 13 – Distribution of indexes for object o_4	23
Figure 14 – The resulting MSA divided in buckets	23
Figure 15 – Algorithm 2 executed for the first element of the bucket buk_{r1}	25
Figure 16 – Algorithm 2 executed for the second element of the bucket buk_{r1}	25
Figure 17 – Algorithm 2 executed for the last element of the bucket buk_{r1}	26
Figure 18 – Algorithm 2 executed for the first element of the bucket buk_{r0}	26
Figure 19 – Algorithm 2 executed for the first element of the bucket buk_{r2}	27
Figure 20 – Accumulator array	27
Figure 21 – Examples of decimals encoded by Elias Delta Code	28
Figure 22 – Number of bits required to Elias Delta encode an integer (power of 10). Abscissa values determine the n in 10^n	28
Figure 23 – Example of a procedure to code using a buffer to implement the Simple- 9 code	30
Figure 24 – Memory hierarchy design	30
Figure 25 – Difference array	33

Figure 26 – Obtaining first element (2) from codified bucket buk_{r_0}	35
Figure 27 – Obtaining first element (1) from codified bucket buk_{r_1}	35
Figure 28 – Obtaining second element (5) from codified bucket buk_{r_0}	36
Figure 29 – Obtaining second element (4) from codified bucket buk_{r_1}	36
Figure 30 – Obtaining third element (8) from codified bucket buk_{r_0}	36
Figure 31 – Obtaining last element (21) from codified bucket buk_{r_0}	37
Figure 32 – Tests with the sets of ANN_SIFT1B and CoPhIR for 256 references . .	41
Figure 33 – Comparing the cMSA with MSA for 256 references	42
Figure 34 – Tests with the sets of ANN_SIFT1B and CoPhIR for 2 million objects	43
Figure 35 – Comparing the cMSA with MSA for 2 million objects	44
Figure 36 – Tests with the sets of ANN_SIFT1B and CoPhIR for 1024 references .	45
Figure 37 – Comparing the cMSA with MSA for 1024 references	46
Figure 38 – Tests with the sets of ANN_SIFT1B and CoPhIR for 8 million objects	47
Figure 39 – Comparing the cMSA with MSA for 8 million objects	48

List of Tables

Table 1 – Ordered lists for every objects generated for the example	17
Table 2 – SFD values for objects	18
Table 3 – System-supported memory standards	31
Table 4 – Summary of the experiments performed	39
Table 5 – Indexing time for cMSA Simple-9 (S_9) and cMSA Delta (δ) compared to MSA obtained in experiments	49
Table 6 – Searching time for cMSA Simple-9 (S_9) and cMSA Delta (δ) compared to MSA obtained in experiments	49
Table 7 – Memory allocated by cMSA Simple-9 (S_9) and cMSA Delta (δ) in per- cent to the memory allocated by MSA obtained in experiments	50

List of Algorithms

1	Full permutation indexing	20
2	Full permutation searching	23

Contents

1	INTRODUCTION	10
1.1	Introduction	10
1.2	Goals and Challenges	11
1.3	Contributions	11
1.4	Dissertation Organization	11
2	FUNDAMENTALS	13
2.1	Similarity searches	13
2.1.1	Range Queries	14
2.1.2	Nearest Neighbor Queries	14
2.2	Permutation-based Indexing	15
2.3	Metric Suffix Array	18
2.4	Indexing	19
2.4.1	Indexing Example	21
2.5	Searching	23
2.5.1	Searching Example	24
2.6	Elias Delta Code	27
2.7	Simple-9	29
2.8	Memory hierarchy	30
2.9	Related Works	31
3	THE COMPACT METRIC SUFFIX ARRAY	33
3.1	Compact MSA: construction	34
3.2	Compact MSA: searching	34
3.3	Compact MSA: summary	37
4	EXPERIMENTAL RESULTS	38
4.1	Calculating array size beforehand	38

4.2	Tests	38
5	CONCLUSION	51
5.1	Future improvements	51
	BIBLIOGRAPHY	53

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

student name and signature

Introduction

1.1 Introduction

With the increase in the volume of data in recent times there has consequently been room for creative ways to manipulate this data. Search and insert operations on data structures containing millions of nodes can incur a high processing demand. In this context, proximity search operations may require adaptations that allow obtaining a compromise between speed and accuracy. In a similarity search, the operations to measure the distance between two objects require calculation that involves mathematical operations like squares and squared roots, as in the case of Euclidean space. These operations are potentially computational expensive and consequently can make the handling of large volume of data quite difficult.

Due to this, some alternatives are being used by applications that just require approximated results instead of exact results. Examples are applications that aim to find plagiarism by similarity between an article and texts in a database, comparison between several genomes to find similarities between one or more genes, multimedia searches where images or videos similar to a given example can be obtained (MOHAMED; MARCHAND-MAILLET, 2013).

A similarity search involving a query q and a collection of data D are usually done in two ways: *K-nearest neighbor search* (K - NN) and the range query. K - NN returns the K objects most similar to query q . The range query returns all objects located within a certain distance from the query q .

For the execution of the similarity search, a collection of data that contains an amount N of objects o_i will be considered. These objects have quantifiable attributes, that is, they are represented by numerical values, which are used in the calculation of distances. In this dissertation, these attributes will be called dimensions. Therefore, from these dimensions it is possible to calculate the distances, in a given vector space, between the data.

For a K - NN search to be carried out, it is necessary to order the distances of objects o_i in relation to a query q . The procedure for obtaining the precise ordering would be

to calculate the distances of the N objects in relation to q . However, this procedure can become computationally expensive for a large number of objects. Furthermore, in applications where very frequently the set of objects does not change, a procedure where the calculation of distances is done only once and, after that, the successive queries are carried out by an indirect procedure that reduces the computational cost is desired.

The advantage of this procedure is that the distance calculations would be performed only once. With the increase of data registers quantity with high dimensionality, the construction of data structures to perform K - NN and range searches by the exact modality presents considerable degradation concerning processing performance and data structure storage memory. Therefore, taking in account that working with exact results could bring those disadvantages, methods which takes some kind of loss of accuracy can be tolerated in many applications.

In the last years most works published in the K - NN and range searches have dealt with the development of data structures which perform those searches via approximate methods. (VADICAMO; AMATO; GENNARO, 2023) deals with permutation-based indexing as an approach to large-scale searching, transforming the original objects representation into a permutation one (MOHAMED; MARCHAND-MAILLET, 2013), thus profiting of efficient method of indexation.

1.2 Goals and Challenges

Starting from the Metric Suffix Array (MSA) (MOHAMED; MARCHAND-MAILLET, 2013), this work intended to propose, implement and test an improved version, which could provide a smaller data structure and, consequently, require less memory. For that, our challenges were to define techniques that can be used together with the MSA so a compact data structure is produced without the necessity of decompression to perform a search.

1.3 Contributions

The compact versions of the MSA, being less memory consuming, could be used as alternative to the MSA when it comes to applications that require large data and relative low memory to stores the data structure, to the point where it can be avoid or postpone the transferring of this data structure to lower levels of memory hierarchy.

1.4 Dissertation Organization

Chapter 2 describes concepts and definitions that is used to implement the compact versions of the MSA. Therefore, an approach to concepts of similarity search, permutation-

based indexing, encoding techniques and memory hierarchy is made in the chapter.

Chapter 3 gives details of the compact MSA proposed in this work, as well as further information about the construction of the cMSA data structure and the process of retrieving the k nearest neighbors of a query q .

Chapter 4 presents tests performed comparing the MSA with two versions of cMSA. These data is presented in graph form with explanation of the compared features of the three implementations.

Chapter 5 discusses about some further improvements that could be made to the cMSA versions, as well as some other techniques and new approaches that could be implemented in future contributions for the work presented here.

Fundamentals

This chapter briefly presents some concepts and definitions that provide the fundamentals required for the development of this dissertation. Section 2.1 gives the fundamentals of the concept of similarity searches and its requirements, giving basis for the range queries and nearest neighbor queries. Section 2.2 presents the principles for the distances computation used by the MSA, which is described in section 2.3. The MSA method is further described in section 2.4 and section 2.5, where the algorithm for these two steps are presented and explained with more detail. Section 2.6 and section 2.7 introduce two encoding methods used to implement the cMSA. Finally, section 2.8 briefly discusses some aspects of memory hierarchy, illustrating how the cMSA could be relevant to real world applications.

2.1 Similarity searches

The concept of similarity, which plays a fundamental role in the functioning of the Metric Suffix Array (MSA) (MOHAMED; MARCHAND-MAILLET, 2013), is defined by functions that return a value that is abstracted as a distance. This distance establishes the dissimilarity between two objects O_i and O_j , where higher values indicate greater dissimilarity.

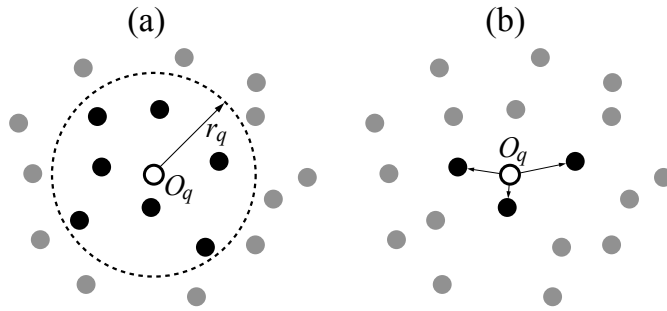
Therefore, a similarity query is understood as a procedure to obtain a set of objects with respect to a definition of a multidimensional space (SAMET, 2006). In this space, the distance function establishes a hierarchy between objects in a query, which is used to compare the similarity between a given object and others that are located nearby. For the query to function properly, it is necessary that this function employs a metric that satisfies the properties of symmetry, identity, non-negativity and triangular inequality as defined below:

Let $\mathbf{D} = \{a_1, a_2, a_3\}$ be a set of objects and $d : \mathbf{D} \times \mathbf{D} \rightarrow \mathbb{R}$ be a distance, properties that define a Metric Space are:

1. Symmetry: $d(a_1, a_2) = d(a_2, a_1)$
2. Identity: $d(a_1, a_1) = 0$
3. Non-negativity: $0 \leq d(a_1, a_2) < \infty$
4. Triangular inequality: $d(a_1, a_2) \leq d(a_1, a_3) + d(a_3, a_2)$

The types of similarity queries that will be addressed in this work will be the range query and the k-nearest neighbor query, defined in subsection 2.1.1 and subsection 2.1.2.

Figure 1 – Similarity searches. (a) Range query. (b) Nearest neighbor query



2.1.1 Range Queries

Given a set of objects O , a query q and a search radius r in a metric space, a Metric Range Query (MRQ) returns all objects in O that lie within the range r of q , or:

$$\text{MRQ}(q, r) = \{o \mid o \in O \wedge d(q, o) \leq r\} \quad (1)$$

2.1.2 Nearest Neighbor Queries

Given a set O , a query q , and an integer k , a k-nearest neighbor query (MkNNQ) finds k objects in O that are closest to q , or:

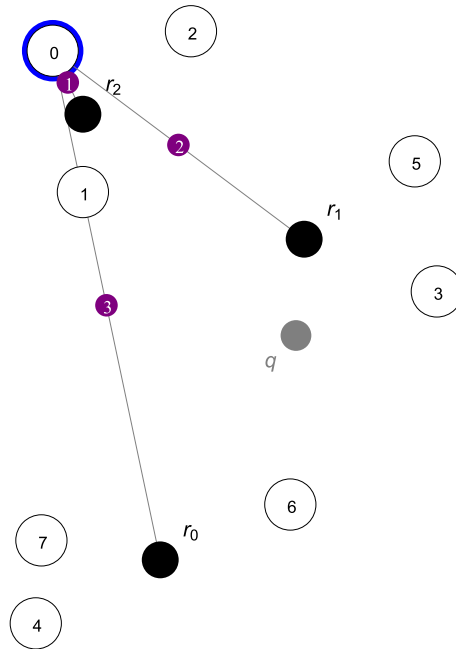
$$\text{MkNNQ}(q, k) = \{S \mid S \subseteq O \wedge |S| = k \wedge \forall s \in S, \forall o \in O - S, d(q, s) \leq d(q, o)\} \quad (2)$$

Figure 1(a) represents the queries defined in subsection 2.1.1 and Figure 1(b) represents the queries defined in subsection 2.1.2. In Figure 1(a), the object o_q is central to the search region, whose radius is r_q . In this case, we have a scope query $\text{MRQ}(o_q, r_q)$, since all objects that are at a distance less than or equal to r_q are returned in the search. Figure 1(b) shows the representation of a query to the $k = 3$ nearest neighbors, that is, $\text{MkNNQ}(o_q, 3)$ and, in this case, the search will return the 3 elements with the smallest distance, that is, possibly greater similarity with o_q .

2.2 Permutation-based Indexing

The concept behind the *permutation-based indexes* is to obtain the distances between elements utilizing a set of reference objects (FIGUEROA et al., 2017). These distances are estimated based on the order each element is from the set of references, then this procedure is repeated for every element, finally establishing the overall distances. To illustrate this procedure, take for example the case illustrated in Figure 2. This scenario shows a set of eight objects numbered from zero to seven. Also, there are three reference objects. Finally, the point q represents a search, i.e. a position from where the objects must be ranked in relation of the distance from there to each object.

Figure 2 – Distance of the references set from object o_0



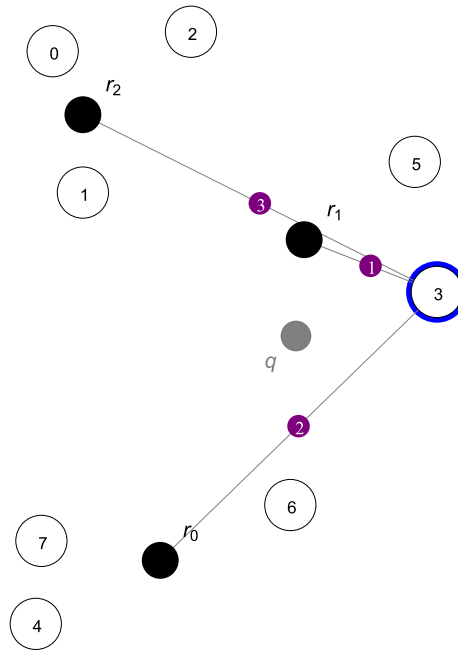
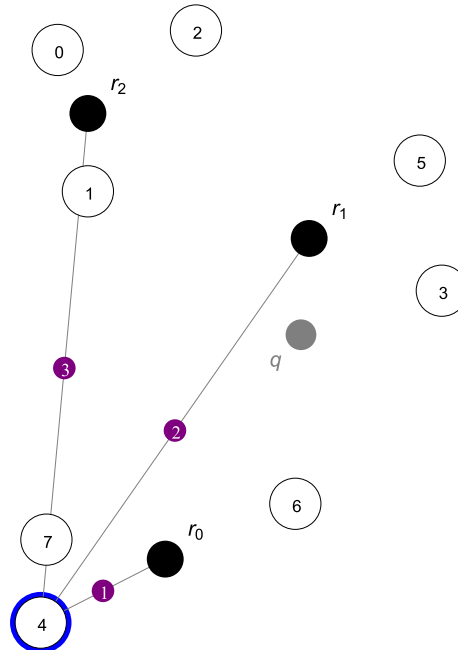
The procedure starts calculating the distance between an element o_i and each reference r_i . Then, a list of references is created, sorted from the nearest to the farthest references. So, in this case, for o_1 , we get $L_{o_1} = \{r_2, r_1, r_0\}$.

Now, for object o_3 , Figure 3 shows that the reference r_1 is the nearest, so this is the first reference appearing in ordered list L_{o_3} . Reference r_0 is the next and reference r_2 is the farthest. So the ordered list for object o_3 generated by this process is $L_{o_3} = \{r_1, r_0, r_2\}$.

For object o_4 , Figure 4 shows that the ordered list generated is $L_{o_4} = \{r_0, r_1, r_2\}$ as reference r_0 is the nearest and reference r_2 is the farthest.

Proceeding like this for every object o_n , a list of ordered list is obtained as shown in Table 1.

After this process, we get the L_q , which is the ordered list of the reference distances to the query q . With this, we proceed to comparing for each reference the position of the respective reference occupy in the ordered list L_{o_n} of each element. As a *permu-*

Figure 3 – Distance of the references set from object o_3 Figure 4 – Distance of the references set from object o_4 

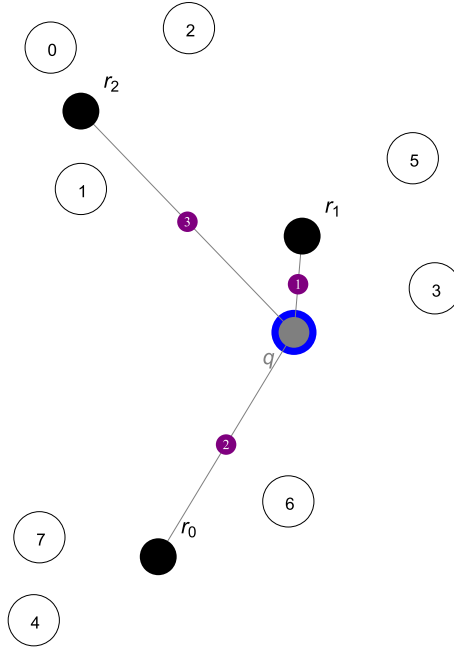
tation based algorithm (PBA), it requires a metric between permutations (FIGUEROA; PAREDES; REYES, 2018), and for this work, the similarity between the query q and all the objects o_i is measured using the *Spearman Footrule Distance (SFD)*, a metric that provides good approximations, with a simple and intuitive procedure. This is made by computing Equation 3 when comparing the ordered lists showed in Table 1. The *SFD* method is executed for every object o_i and is given by the formula:

Table 1 – Ordered lists for every objects generated for the example

Obj	near	mid	far
o_0	2	1	0
o_1	2	1	0
o_2	2	1	0
o_3	1	0	2
o_4	0	1	2
o_5	1	2	0
o_6	0	1	2
o_7	0	1	2

$$\text{SFD}(o_i, q) = \sum |p(L_{o_i}, r_j) - p(L_q, r_j)| \quad (3)$$

where L_{o_i} is the ordered references list of object o_i , L_q is the sorted references list of the query q , $p(L_{o_i}, r_j)$ returns the position of r_j from the ordered list L_{o_i} and $p(L_q, r_j)$ returns the position of r_j from the ordered list L_q . Figure 5 shows how the L_q is constructed for the example in analysis: the reference r_1 is the nearest from the query position followed by reference r_0 and reference r_2 being the farthest one, thus resulting in $L_q = \{r_1, r_0, r_2\}$.

Figure 5 – Distance of the references set from query q 

After obtained L_q , the process calculates a SFD value for each object related to the query q as described in Equation 3. For exemplify how to calculate a SFD value, take the ordered list of object o_0 , $L_{o_0} = 2, 1, 0$, as shown in Table 1. This list should be compared with the ordered list of query, $L_q = 1, 0, 2$, so starting from the first reference appearing in this list, we have reference r_1 , which have $p(L_q, r_1) = 0$ (because r_1 appears at the first position at L_q). Looking at L_{o_0} , reference r_1 appears at second position, hence

$p(L_{o_0}, r_1) = 1$. Then, the first term of sum shown in Equation 3 is the module of the difference between those values, that is, $|1 - 0| = 1$. This process is shown in Figure 6(a). Proceeding to the next reference in L_q , we get reference r_0 at position 1. Reference r_0 at ordered list L_{o_0} is at position 2 and, as show in Figure 6(b), the second term of the sum is $|2 - 1| = 1$. Finally, the last position at L_q is reference r_2 , which is the first position at L_{o_0} , therefore $|0 - 2| = 2$, as shown in Figure 6(c). The sum of terms results 4, which is the SFD of object o_0 .

Figure 6 – Example of SFD value calculation for object o_0

(a)	(b)	(c)																								
<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td>0</td><td style="border: 2px solid red;">1</td><td>2</td></tr> <tr><td>o_0</td><td>r_2</td><td style="border: 2px solid red;">r_1</td><td>r_0</td></tr> </table>	P	0	1	2	o_0	r_2	r_1	r_0	<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td>0</td><td>1</td><td style="border: 2px solid red;">2</td></tr> <tr><td>o_0</td><td>r_2</td><td>r_1</td><td style="border: 2px solid red;">r_0</td></tr> </table>	P	0	1	2	o_0	r_2	r_1	r_0	<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td style="border: 2px solid red;">0</td><td>1</td><td>2</td></tr> <tr><td>o_0</td><td style="border: 2px solid red;">r_2</td><td>r_1</td><td>r_0</td></tr> </table>	P	0	1	2	o_0	r_2	r_1	r_0
P	0	1	2																							
o_0	r_2	r_1	r_0																							
P	0	1	2																							
o_0	r_2	r_1	r_0																							
P	0	1	2																							
o_0	r_2	r_1	r_0																							
<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td style="border: 2px solid red;">0</td><td>1</td><td>2</td></tr> <tr><td>q</td><td style="border: 2px solid red;">r_1</td><td>r_0</td><td>r_2</td></tr> </table>	P	0	1	2	q	r_1	r_0	r_2	<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td>0</td><td style="border: 2px solid red;">1</td><td>2</td></tr> <tr><td>q</td><td>r_1</td><td style="border: 2px solid red;">r_0</td><td>r_2</td></tr> </table>	P	0	1	2	q	r_1	r_0	r_2	<table border="1" style="border-collapse: collapse; margin: auto;"> <tr><td>P</td><td>0</td><td>1</td><td style="border: 2px solid red;">2</td></tr> <tr><td>q</td><td>r_1</td><td>r_0</td><td style="border: 2px solid red;">r_2</td></tr> </table>	P	0	1	2	q	r_1	r_0	r_2
P	0	1	2																							
q	r_1	r_0	r_2																							
P	0	1	2																							
q	r_1	r_0	r_2																							
P	0	1	2																							
q	r_1	r_0	r_2																							
$ 1 - 0 = 1$	$ 2 - 1 = 1$	$ 0 - 2 = 2$																								

Table 2 – SFD values for objects

Obj	near	mid	far	SFD
o0	2	1	0	4
o1	2	1	0	4
o2	2	1	0	4
o3	1	0	2	0
o4	0	1	2	2
o5	1	2	0	2
o6	0	1	2	2
o7	0	1	2	2

Table 2 shows the SFD value for the objects. Finally, sorting those values permits to estimate the distance of the elements from the query position.

As mentioned before, this process allows to estimate a ranking of the distance of the objects to the query at the cost of precision related to the direct mode, which involve computationally expensive operations, like square roots. Therefore, this method allows for faster queries, in a large data environment, at the expense of precision. However, the level of precision can be improved with the increase of the number of references.

2.3 Metric Suffix Array

The procedure described in section 2.2 can still be quite computationally expensive when handling a large volume of objects, thus some type of optimization can be used to improve the time required to processing these distance ranking procedure.

Let S be a string of length $m = |S|$ comprised of an *alphabet* Σ . Taking the character $\$$ to signal the end, so $S[m] = \$$, and $S[i]$ indicating the character at position i in S , for $0 \leq i < m$, therefore $S[i..j]$ represents the substring S starting at i and ending at j . The i -th suffix of S is the substring $S[i..m]$ and is denoted by $S(i)$. The suffix array `suff` of the string S is an array of integers in the range 0 to m where the suffixes are in lexicographic order (MANBER; MYERS, 1993).

The Metric Suffix Array (MSA) take advantage of the principles of indexing with a suffix array. In this method, we concatenate in the ordered lists L_{o_i} , as obtained by the process described in section 2.2, for all the database objects in an array. We obtain one single string S over a finite set $\Sigma = R$. The length of the string is $m = n \times N$, where n is the number of reference points and N is the number of objects. At each position of this data structure, a value of the position of a reference to an object is stored, and this allows for fast recovering of $p(L_{o_i}, r_j)$.

Therefore, for instance, suppose that all lists L_{o_i} shown in Table 1 are concatenated and a character is added to signal the end ($\$$ in this case). The result is the following string:

$$\mathbf{S = 210210210102012120012012\$}$$

The sequence of suffixes of S in ascending lexicographic order is denoted by $S(\text{suff}[0])$, $S(\text{suff}[1])$, ..., $S(\text{suff}[m])$ as shown in Figure 7. The `suff` value denotes the order of generation of a substring, as show in Figure 7. Suffix arrays are used to locate every occurrence of substring $S(\text{suff}[i])$ in an effective way for many applications.

These principles are used to efficiently implement the indexation of the MSA which is described in the section 2.4.

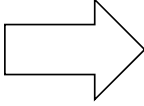
2.4 Indexing

The method described in section 2.2 works as the fundamentals to the procedure described in this dissertation, because, to optimal memory usage. To start, we construct an MSA of size $m = n \times N$, where n is the number of references and N is the number of objects. This array contain a list of indexes, which is separated in n groups, called buckets. Each bucket is a subarray of size N . The procedure in this methods places the references in the L_{o_i} in the i_{th} bucket position. Each bucket is related to a reference so the n -th bucket defined as buk_{r_n} receive indexes labeled with n . So, for instance, buk_{r_1} and buk_{r_2} contain indexes labeled as 1 and 2, respectively.

Disposing the indexes in that fashion eases the calculation of the SFD value (Equation 3) of each object in an iterative procedure. This works by comparing the indexes with the position of the references in the ordered list L_q .

Figure 7 – The suffix array of the string $S = 210210210102012120012012\$$

i	suff	S(suff[i])
0		210210210102012120012012\$
1		10210210102012120012012\$
2		0210210102012120012012\$
3		210210102012120012012\$
4		10210102012120012012\$
5		0210102012120012012\$
6		210102012120012012\$
7		10102012120012012\$
8		0102012120012012\$
9		102012120012012\$
10		02012120012012\$
11		2012120012012\$
12		012120012012\$
13		12120012012\$
14		2120012012\$
15		120012012\$
16		20012012\$
17		0012012\$
18		012012\$
19		12012\$
20		2012\$
21		012\$
22		12\$
23		2\$
24		\$



i	suff	S(suff[i])
0	17	0012012\$
1	8	0102012120012012\$
2	18	012012\$
3	12	012120012012\$
4	21	012\$
5	10	02012120012012\$
6	5	0210102012120012012\$
7	2	0210210102012120012012\$
8	7	10102012120012012\$
9	9	102012120012012\$
10	4	10210102012120012012\$
11	1	10210210102012120012012\$
12	15	120012012\$
13	19	12012\$
14	13	12120012012\$
15	22	12\$
16	16	20012012\$
17	11	2012120012012\$
18	20	2012\$
19	6	210102012120012012\$
20	3	210210102012120012012\$
21	0	210210210102012120012012\$
22	14	2120012012\$
23	23	2\$
24	24	\$

Figure 8 – String S as a character sequence showing the position indexes of each character

$S =$	2	1	0	2	1	0	2	1	0	1	0	2	0	1	2	1	2	0	0	1	2	0	1	2
Index =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Obtained in section 2.3, the sequence of characters S is shown in Figure 8 with the position index for each character. The number stored in each position indicates the reference in that position, therefore, for instance, positions 0 and 3 store reference r_2 and positions 1 and 4 store reference r_1 .

Computationally, the MSA is obtained by running through each of the N objects and for each of the n references and calculated by Equation 4.

$$MSA[(r_j \times N) + o_i.id] = (o_i.id \times n) + p(L_{o_i}, r_j) \quad (4)$$

where $p(L_{o_i}, r_j)$ returns the position of r_j in the ordered list L_{o_i} , $o_i.id$ returns the object o_i identification and $r_j.id$ returns the object o_i identification.

Algorithm 1 Full permutation indexing

- 1: **for** each $o \in D$ **do**
 - 2: $L_{o_i} = CreateOrderedList(o_i, R)$
 - 3: **for** each $r \in L_{o_i}$ **do**
 - 4: $MSA[(r_j.id \times N) + o_i.id] = (o_i.id \times n) + p(L_{o_i}, r_j)$
 - 5: **end for**
 - 6: **end for**
-

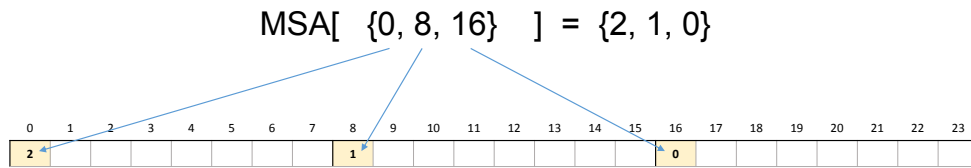
Algorithm 1 presents the procedure used to construct the MSA. Line 1 defines that for every object input, line 2 creates an ordered list for the current object and line 3 and 4, for every reference of this ordered list, the SFD value (Equation 3) is calculated and stored at an MSA position.

More specifically, line 2 invokes function $CreateOrderedList(o_i, R)$, which receives as arguments the current object o_i and the reference set R , returning this reference set R in a list ordered from the nearest to the farthest to object o_i . Line 4 computes the value as the $o_i.id$ times n , which obtains the first position occupied by the i -th object at the original sequence of character S (Figure 8) plus the distance ranking provided by function $p(L_{o_i}, r_j)$. Function P takes as arguments the ordered list L_{o_i} and reference r_j and returns the distance ranking of reference r_j at L_{o_i} . If r_j is the nearest of object o_i , the function returns 0; if r_j is the farthest, the function returns $n - 1$. This calculated value is stored at the specific bucket buk_{r_j} and object o_i position: the first position of the bucket buk_{r_j} is obtained from $(r_j.id \times N)$, which is added to $o_i.id$, arriving at the aforementioned position. As the Algorithm 1 requires for all objects a run through every reference, in which the function P is invoked, so function P is invoked $n \times N$ times. Knowing that this function is a simple linear search, the average time required is $n/2$. Therefore, for the entire indexing processing, the computational cost is $O(n^2 \times N)$, where n is the number of references and N is the number of objects.

For instance, if we take the reference r_2 and the object o_5 , then the position of the MSA array is 21, because $(r_j.id \times N) + o_i.id = (2 \times 8) + 5 = 21$. The reference r_2 appears at position 1 in the ordered list $L_{o_5} = \{1, 2, 0\}$ and, therefore, for the position 21 of the MSA array, we have the index 16, because $(o_i.id \times n) + p(L_{o_i}, r_j) = (5 \times 3) + 1$. Proceeding this way for every object and every reference, we obtain the MSA.

2.4.1 Indexing Example

The process to construct the MSA array works as follows: each reference r_i is associated with a bucket, which is a subarray that contains the position occupied by that reference i in the string S . So, for instance, in the case being analyzed, if the reference r_0 occupies the position 2 in the string, so it means that for object o_0 , the reference r_0 is the third nearest reference from object o_0 and, in the r_0 bucket, the first position (or the o_0 position) will be filled with the 2 index, as it can be seen in Figure 9. Yet for the first object, proceed as described, we find that the index 1 and index 0 are placed at the first position of buk_{r_1} and buk_{r_2} respectively (Figure 9), meaning that the references r_1 and r_2 are second and first nearest to object o_0 . The line $MSA[\{0, 8, 16\}] = \{2, 1, 0\}$ means that at positions 0, 8 and 16 of MSA are stored the indexes 2, 1 and 0 respectively, because in the sequence of characters S shown in Figure 8 these are the indexes storing values of 0, 1 and 2 respectively.

Figure 9 – Distribution of indexes for object o_0 

For object o_1 , Figure 10 shows that reference r_0 appears at position 5 in the string and is the farthest element in the ordered list L_{o_1} as $5 \bmod 3 = 2$. Reference r_1 appears at position 4 in the string, which means that the value 4 is put in next position at bucket buk_{r_1} and the reference r_2 , the nearest, as $3 \bmod 3 = 0$, is related with the index 3. The line $MSA[\{1, 9, 17\}] = \{5, 4, 3\}$ means that at positions 1, 9 and 17 of MSA are stored the indexes 5, 4 and 3 respectively, because in the sequence of characters S shown in Figure 8 these are the indexes storing values of 0, 1 and 2 respectively.

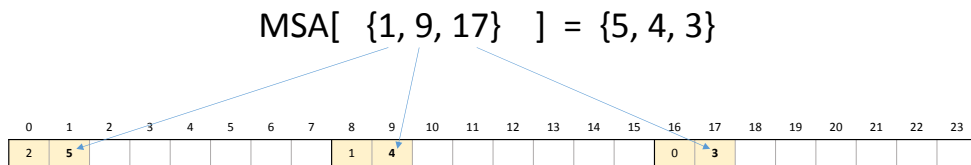
Figure 10 – Distribution of indexes for object o_1 

Figure 11 shows the distribution for object o_2 , where index 8 is placed at the next position of bucket buk_{r_0} , index 7, at the next position of bucket buk_{r_1} and index 6, at the next position of bucket buk_{r_2} .

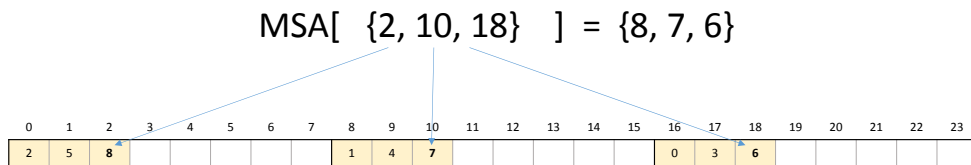
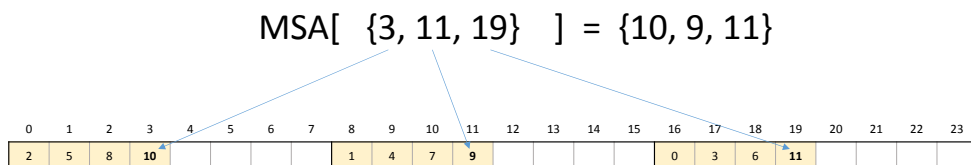
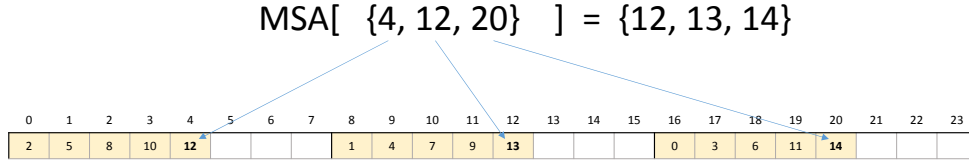
Figure 11 – Distribution of indexes for object o_2 

Figure 13 shows the distribution for object o_3 , the next position of bucket buk_{r_0} , which is position 3 of MSA, receives the index 10, which reference is the second nearest, as $10 \bmod 3 = 1$; the next position of bucket buk_{r_1} , which is position 11 of MSA, receives the index 9, which reference is the nearest, as $9 \bmod 3 = 0$ and the next position of bucket buk_{r_2} , which is position 19 of MSA, receives the index 11, which reference is the farthest, as $11 \bmod 3 = 2$.

Figure 12 – Distribution of indexes for object o_3 

Another example, if reference r_1 appears at position 13 in the string, so it means that the reference r_1 is the second nearest from object o_4 . This is so because for this position, $13 \bmod 3 = 1$. Also we can predict that is object o_4 because, as Figure 8 shows, the index 13 indicates a position that received a reference related to object o_4 and this can be calculated by $\lfloor \text{index}/n \rfloor$. Therefore, the object related to this index is $\lfloor 13/3 \rfloor = 4$. Thus, the index 13 is filled in the position 4 of the r_1 bucket, as shown in Figure 13.

Figure 13 – Distribution of indexes for object o_4 

Observing Figure 9, Figure 10, Figure 11 and Figure 12 it is easy to see that for the i -th object, the indexes are placed at the i -th position of each bucket. This procedure is repeated for every index in the string so we obtain the MSA which in the case analyzed

buk _{r₀}					buk _{r₁}					buk _{r₂}													
2	3	3	2	2	5	1	3	1	3	3	2	4	2	4	3	0	3	3	5	3	2	4	3

2.5 Searching

Once the MSA array is built, queries can be executed directly on the MSA. Therefore the remaining part of the procedure begins with the calculation of the ordered list L_q which features a sorted nearest references from the query position. Once the L_q is obtained, the process to obtain the SFD value for each object from the MSA is performed by Algorithm 2.

Algorithm 2 Full permutation searching

- 1: **for** $r_j \in L_q$ **do**
 - 2: $\text{RefPosQ} \leftarrow j$
 - 3: $O_{id} \leftarrow 0$
 - 4: **for** $k \leftarrow (r_j.id \times N)$ to $k < (r_j.id \times N) + N$ **do**
 - 5: $\text{RefPos} \leftarrow (\text{MSA}[k] \bmod n) + 1$
 - 6: $\text{Acc}[O_{id}] \leftarrow \text{Acc}[O_{id}] + |\text{RefPosQ} - \text{RefPos}|$
 - 7: $O_{id} \leftarrow O_{id} + 1$
 - 8: **end for**
 - 9: **end for**
-

Line 1 of Algorithm 2 proceeds for the each reference, assigning the variables RefPosQ and O_{id} to j and 0 respectively (Lines 2 and 3) and then running through the bucket buk_{r_j} ,

from the first element, $r_j.id \times N$, to the last of this bucket, $(r_j.id \times N) + (N - 1)$. At Line 5, `RefPos` receives $(MSA[k] \bmod n) + 1$, decoding the ranking position of the reference r_j for object O_{id} , Line 6 compares this ranking position with the value `RefPosQ` and adds to accumulator `Acc` at the O_{id} position. Line 7 increments O_{id} . The inner loop (Line 4) repeats N times (N is the number of objects), from o_0 to o_{N-1} , updating accumulator `Acc`. The outer loop (Line 1) repeats n times (n is the number of references), from r_0 to r_{n-1} , finally achieving the final state of accumulator `Acc`. The computational cost for searching is $O(n \times N)$. The memory required for the `Acc` is $O(N)$. The method in its original form requires the calculation of `Acc` array in its entirety once the algorithm performs Line 6 once for each object and then for each reference. Further, in the dissertation, we present alterations to the original proposition such that the generation of the entire `Acc` is not required anymore.

2.5.1 Searching Example

For instance, taking the analyzed case, the query returns the ordered list $L_q = \{1, 0, 2\}$, which means that the reference r_1 is the nearest and the reference r_2 is the farthest. Therefore, we start with the reference r_1 , which means that we will run through every element in bucket buk_{r_1} , from the element 8 ($r_j.id \times N = 1 \times 8 = 8$) until the element 15 (before $r_j.id \times N + N = r \times 8 + 8 = 16$). Following, we proceed for each element of the bucket which the first one is related with object o_0 and the last one, with object o_7 .

It is obtained the position of r_1 in each ordered list L_{o_i} . This will be achieved thanks to the disposition that each bucket in the MSA presents its elements. Following the case presented in Figure 15, the position of the reference r_1 in the ordered list L_{o_0} is obtained by taking the first element of the bucket, that it is 1. As there is 3 references in total, we take the modulo of 3 contained in an element to find the position of the reference relative to the current bucket to obtain its rank. In this case the value of `RefPos` is obtained, meaning that the reference r_1 is the second nearest of the object o_0 , that is, the position $p(L_{o_0}, r_1) = 2$.

In the sequence, the value obtained is compared with the position of reference r_1 relative to the query q , that is, to the position $p(L_q, r_1)$, which in the case shown in Figure 15 has a value of 1. In the Algorithm 2 this is made by taking the absolute of the difference between `RefPosQ` and `RefPos`, and then, the result is accumulated in an array with N elements.

By the Figure 16, proceeding for the second element in bucket buk_{r_1} (element 9 in the MSA), we get the value 4, related to the object o_1 . Applying the calculation, that is $(MSA[9] \bmod 3) + 1$, we get the value 2. Taking the absolute of the difference of the two positions, we get 1, which is added to the accumulator array in the second element.

Yet inside bucket buk_{r_1} and proceeding like this for the rest of elements, the last one is achieved in Figure 17, which have the value 22, meaning that the object o_7 has the

Figure 15 – Algorithm 2 executed for the first element of the bucket buk_{r_1}

$$L_q = \{1, 0, 2\}$$

$$n = 3$$

For r_1

$$p(L_q, r_1) = 1$$

MSA:

	buk ₀							buk ₁							buk ₂									
	2	5	8	10	12	17	18	21	1	4	7	9	13	15	19	22	0	3	6	11	14	16	20	23

$$O_{id} = 0$$

$$RefPos = p(L_{O_0}, r_1) = (MSA[k] \bmod n) + 1 = (1 \bmod 3) + 1 = 2$$

$$\Delta = |p(L_{O_0}, r_1) - p(L_q, r_1)| = |2 - 1| = 1$$

$$Acc = Acc + 1$$

	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇
Acc:	0+1	0	0	0	0	0	0	0

Figure 16 – Algorithm 2 executed for the second element of the bucket buk_{r_1}

$$L_q = \{1, 0, 2\}$$

$$n = 3$$

For r_1

$$p(L_q, r_1) = 1$$

MSA:

	buk ₀							buk ₁							buk ₂									
	2	5	8	10	12	17	18	21	1	4	7	9	13	15	19	22	0	3	6	11	14	16	20	23

$$O_{id} = 1$$

$$RefPos = p(L_{O_1}, r_1) = (MSA[k] \bmod n) + 1 = (4 \bmod 3) + 1 = 2$$

$$\Delta = |p(L_{O_1}, r_1) - p(L_q, r_1)| = |2 - 1| = 1$$

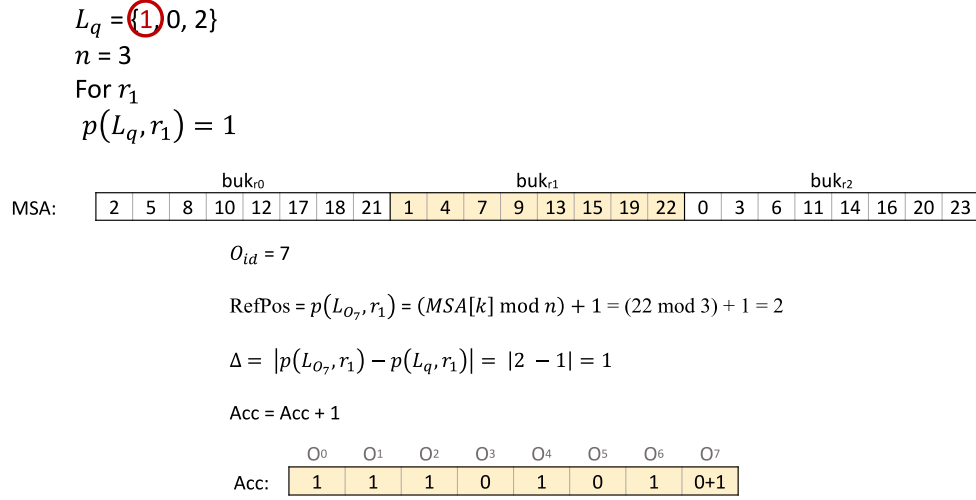
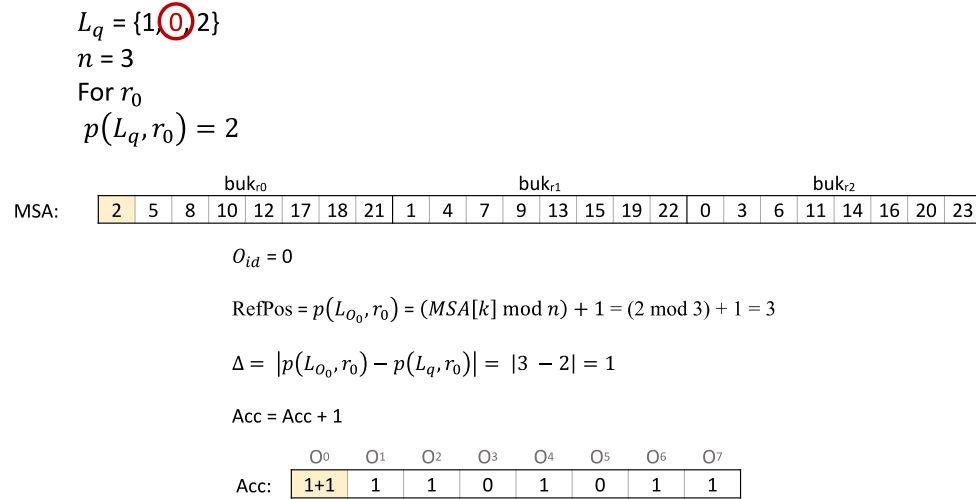
$$Acc = Acc + 1$$

	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇
Acc:	1	0+1	0	0	0	0	0	0

reference r_1 as the second nearest, because $(22 \bmod 3) + 1 = 2$. Taking the absolute of the difference of the two positions, we get 1, which is added to the accumulator array in the last element.

After finishing all calculations for reference r_1 , we proceed to the next reference, which is r_0 as shown in Figure 18. This cycle starts at the very first position of the MSA, which have the value of 2, meaning that the object o_0 has the reference r_0 as the farthest, because $(2 \bmod 3) + 1 = 3$. Taking the absolute of the difference of the two positions, we get 1, which is added to the accumulator array in the first element.

Proceeding similarly to all elements of bucket buk_{r_0} and, after that, for bucket buk_{r_2} , starting with the first element of value 0, as shown in Figure 19, the value added to the accumulator array is 2, which makes a total of 4 for object o_0 . Still at bucket buk_{r_2} , proceeding the same way for the remaining objects, we obtain the final result in the accumulator array, which is exactly the same as calculating directly the SFD (Equation 3) for

Figure 17 – Algorithm 2 executed for the last element of the bucket buk_{r_1} Figure 18 – Algorithm 2 executed for the first element of the bucket buk_{r_0} 

each object by using Equation 3. In the case analyzed, we get $Acc = \{4, 4, 4, 0, 2, 2, 2, 2\}$, as shown in Figure 20. Therefore, this result means that the first three objects have a distance of 4 from the query q , object o_3 has a distance of zero and the last four objects have a distance of 2 from query q .

Finally, after performing Algorithm 2, we obtain a value for each object that can be compared with the other object values to determine the proximity of those objects related to a position determined by a query q . At the end, sorting the accumulator array, we obtain a sequence in which the first elements are the nearest ones. Then, we can solve K -NN queries with the k first elements of Acc .

Figure 19 – Algorithm 2 executed for the first element of the bucket buk_{r_2}

$L_q = \{1, 0, \textcircled{2}\}$
 $n = 3$
 For r_2
 $p(L_q, r_2) = 3$

MSA:

	buk ₀						buk ₁						buk ₂										
2	5	8	10	12	17	18	21	1	4	7	9	13	15	19	22	0	3	6	11	14	16	20	23

$O_{id} = 0$

$RefPos = p(L_{O_0}, r_2) = (MSA[k] \bmod n) + 1 = (0 \bmod 3) + 1 = 1$

$\Delta = |p(L_{O_0}, r_2) - p(L_q, r_2)| = |1 - 3| = 2$

$Acc = Acc + 2$

Acc:

	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇
2+2	2	2	0	2	1	2	2	

Figure 20 – Accumulator array

Acc:

	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇
4	4	4	0	2	2	2	2	

2.6 Elias Delta Code

In this section, we show how to use the Elias Delta Code (NAVARRO, 2016) to compress the integers stored in the MSA. In this compression the smaller integers have the least number of bits. This is very convenient as the new difference MSA proposed is very likely to display small values compared to N .

The Elias Delta Code of an integer x is constructed by the concatenation of the Gamma Code of the length of the binary code of x and the binary code of x without the most significant bit (msb), as shown in Equation 5.

$$\delta(x) = \gamma(|x|) \cdot [x]_{|x|-1} \quad (5)$$

where $|x|$ is the length of x in binary and the subscript represents the number of digits taken from the binary representation, therefore $[x]_{|x|-1}$ means that the binary code of x is taken without the most significant bit. The Gamma (γ) Code is obtained from an integer x by concatenating zeros with the binary code of x , as show in Equation 6.

$$\gamma(x) = \mathbf{0}^{|x|-1} \cdot [x]_{|x|} \quad (6)$$

Thus, to illustrate how the Elias Delta Code works take the decimal 19: in binary this integer is **10011**, which has length 5. So to compute the Delta Code it is required to calculate $\gamma(|x|) = \gamma(5)$ and knowing that the binary code for 5 is **101**, by Equation 6 we get $\gamma(5) = \mathbf{0}^{3-1} \cdot \mathbf{101} = \mathbf{00101}$. Now, to obtain the Delta Code for the decimal 19, we used the previously result in $\delta(19) = \gamma(5) \cdot [\mathbf{10011}]_4 = \mathbf{001010011}$.

The Elias Delta Code provides that an integer x will be coded to have length $|\delta(x)|$ bits such as:

$$|\delta(x)| = |x| + 2\lfloor \log_2(|x|) \rfloor \quad (7)$$

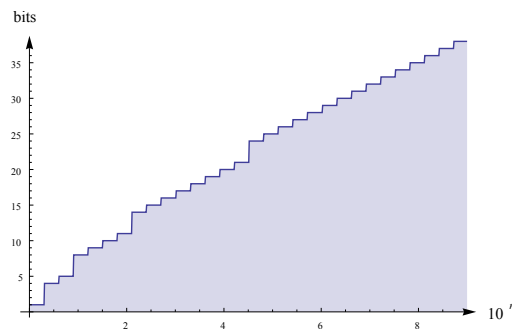
As integers coded will display variable length, the average length in bits will likely be smaller than the standard fixed 32 bits used for coding integers. In fact, with 32 bits, the larger number generated by the Elias Delta Code is $16,777,215 = 2^{24} - 1$. Therefore, every decimal smaller will require 32 bits or less, which means that at that range, the Elias Delta Code will produce smaller codes and, hence, spare memory. Some examples of decimals coded with Elias Delta are shown in Figure 21, which can give an idea of how this codification bit size increase with the grow of the input integer and also how small numbers can be codified using just a few bits. Figure 22 shows the number of bits required to encode a decimal by power of 10: for instance, 10^4 is encoded with 20 bits, while 10^6 , with 28 bits.

Figure 21 – Examples of decimals encoded by Elias Delta Code

1 = 1	10 = 00100010
2 = 0100	11 = 00100011
3 = 0101	12 = 00100100
4 = 01100	13 = 00100101
5 = 01101	14 = 00100110
6 = 01110	15 = 00100111
7 = 01111	16 = 001010000
8 = 00100000	17 = 001010001
9 = 00100001	18 = 001010010
	19 = 001010011
	20 = 001010100

$$16,777,215 = \underbrace{00001100011111111111111111111111}_{32 \text{ bits}}$$

Figure 22 – Number of bits required to Elias Delta encode an integer (power of 10). Abscissa values determine the n in 10^n .



2.7 Simple-9

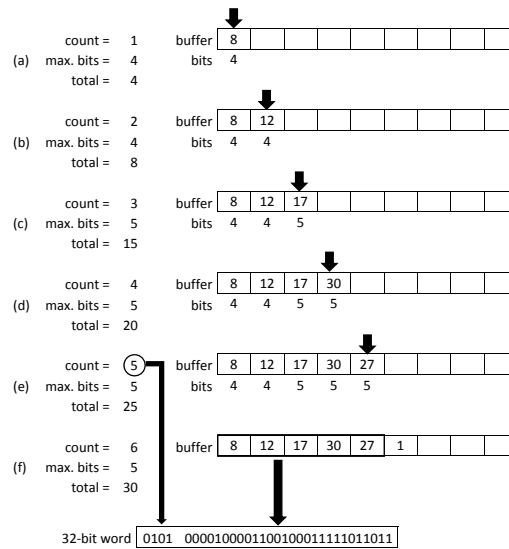
As an alternative to the Elias Delta Code, the Simple-9 (NAVARRO, 2016) is a technique that also provides gain in memory while delivers fast decoding, so achieving good performance. A key concept for this method is the word-RAM, which is memory space consisting of 32 bits. The strategy of this method is to encode as many numbers as possible in a 32-bit word and for that it reserves the highest 4 bits of the word to set how many number are encoded. The next 28 bits are used to accommodate the number of integers set. So, if the next 28 values to encode are 1 or 2, then one bit for each can be used. Otherwise, if the next 14 values are up to 4, then 2 bits for each can be used. If not, if the next 9 numbers are up to 8, then 3 bits for each can be used. In this case a bit is wasted. The same logic can be applied for a total of 9 possibilities where it can encode the next $\lfloor 28/m \rfloor$ values using $m = 1, 2, 3, 4, 5, 7, 9, 14,$ or 28 bits per value. Numbers requiring more than 28 bits cannot be encoded.

As the Elias Delta Code, the Simple-9 encoding is also a data compression standard. However, differently from the Elias Delta Code, where each value is encoded individually, this method requires a short sequence of numbers before applying the algorithm to encoding. Thus, to comply with the work presented here, it is necessary the implementation of a buffer, where the numbers are accumulate before they can be coded in a 32-bit word.

To illustrate how the Simple-9 method, an example is provided in Figure 23: at (a) the first number put in the buffer is 8, which needs 4 bits to be stored in a 32-bit word. As more integers can still be stored, we proceed with the next numbers. At (b) 12 is the next value stored in buffer, which also requires 4 bits to be stored as a binary number, so the maximum number of bits required for each value is still 4, with a total o 8 bits. At (c) the third value put in the buffer is 17, which requires 5 bits to be coded in binary. So, this value is updated, defining the minimal number of bits required to represent those three values in the buffer. Now these three integers require 15 bits to be stored (5 bit to each value). Thus, the process continues, always testing if the total number of bits is equal or larger than 28. At (d) the forth value is 30, which requires 5 bits to be coded. Therefore, 20 bits are required to store the 4 values. At (e) it is required 5 bits to store each value, with a total of 25 bits. At this point, any other value put in the buffer will exceed 28 bits as seen in (f). Therefore, the algorithm proceeds to flush these 5 integers requiring 5 bits each to a 32-bit word, where the first 4 bits stores the value 5, which is the number of values that are stored in the next 28 bits.

The memory sparing comes from using 32 to store multiple integers. This method can store integers up to 28 bits, that is, $2^{28} = 268,435,456$. Larger numbers cannot be stored. This limit do not represent a problem to our proposal as this value is far beyond what is expect to be stored: the differences obtained are equal or smaller than $2n - 1$, where n is the number of references, which are typically of the order of hundreds or thousands. More so, this technique proves to be easily decoded, as it only requires to split the 28 bits least

Figure 23 – Example of a procedure to code using a buffer to implement the Simple-9 code



significant bits (lsb) by the number indicated at the head of the word. This guarantees a constant time at decoding stage.

2.8 Memory hierarchy

The memory hierarchy is the disposition of several levels of memory such that it can minimize the access time. The memory design is divided into two main types: external memory, comprising magnetic and optical disks, and internal memory, comprising main memory, cache memory and CPU registers. As show in Figure 24, the capacity and access time increases as we move from top to bottom in the memory hierarchy, while the cost per bit increases as we move from bottom to top. The internal memory is costlier than external memory.

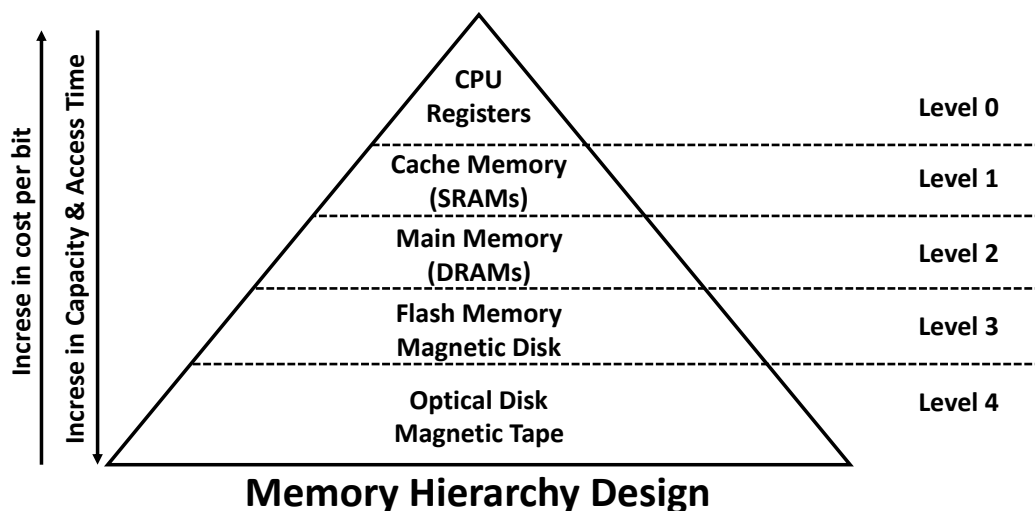


Table 3 shows features of different levels of memory and, as it can be noted, the secondary memory has a much larger access time than the superior memory levels. Therefore, this notable increase in access time motivates the development of compact data structures that can postpone its storage at that level of memory.

Table 3 – System-supported memory standards

Level	0	1	2	3
Name	Register	Cache	Main Memory	Secondary Memory
Size	≤ 1 kB	≤ 16 MB	≤ 16 GB	≤ 2 TB
Access Time	0.25 ns to 0.5 ns	0.5 ns to 25 ns	80 ns to 250 ns	50×10^5 ns
Bandwidth	2×10^4 to 10^5 MB/s	5000 to 15000 MB/s	1000 to 5000 MB/s	20 to 150 Mb/s

2.9 Related Works

In recent decades, several data structures have been proposed to accelerate similarity queries (HJALTASON; SAMET, 2003; SAMET, 2006), including main memory structures for finding k -nearest neighbors, such as the Vantage-Point Tree (VP-Tree) (YIANILOS, 1993) and the Multi Vantage-Point Tree (MVP-Tree) (BOZKAYA; ÖZSOYOGLU, 1999), among others, which are structures that extend binary trees for indexing based on a distance function, whose objective is to solve similarity queries with logarithmic complexity of time.

Other research, focused on secondary storage, presented algorithms for similarity queries in R-trees (GUTTMAN, 1984), that are multidimensional disk-based data structures built in a bottom-up fashion that store the nodes on data blocks of fixed size, such as B+trees. K -nearest neighbor algorithms were proposed for such structures, in a *branch-and-bound* depth-first traversal strategy (ROUSSOPOULOS; KELLEY; VINCENT, 1995; CIACCIA; PATELLA; ZEZULA, 1997; HJALTASON; SAMET, 2003) or in a best-first traversal strategy guided by a global active branch list maintained with a priority queue (HJALTASON; SAMET, 1999; HJALTASON; SAMET, 2003). These query algorithms were also supported by M-trees (CIACCIA; PATELLA; ZEZULA, 1997), which extends the R-trees for the metric case, where data is organized only by their relative distances among each other. These data structures and search algorithms aim at finding exact answers (CHEN et al., 2022), and they suffer from the dimensionality curse (KORN; PAGEL; FALOUTSOS, 2001), that limits the scalability of these algorithms.

In the last decade, research has focused on data structures for approximate similarity queries (Approximate k -Nearest Neighbor Queries – ANNQ), aiming at scalability. (FIGUEROA et al., 2017) proposes several experiments with permutant searching variations to determinate optimal values of the parameters for k -nearest neighbor queries. (VADICAMO; AMATO; GENNARO, 2023) proposes a technique that uses space transformation to perform the permutations in an approximate metric searching context, achieving notable efficiency in the search phase. (AGUERREBERE et al., 2023) proposes a

compression method which results in smaller graph-based indices, allowing for better computational performance and memory storage.

There are specific applications where an exact similarity search is indispensable due to the critical nature of the tasks involved like in biomedical research and drug discovery, where the identification of similar molecular structures is vital, ensuring precision in matching biological sequences or chemical structures; forensic analysis, where the matching patterns with an exact similarity search is unavoidable; and other applications related to security. Similarity search in metric spaces often requires expensive and complex methods, which still do not provide reasonable response time for large applications. Therefore, for many applications where inaccurate results are tolerated and, many times, subjective, it is sufficient to perform *approximate similarity search*, which normally can be performed much faster. Approximate similarity search techniques provide improved efficiency when compared with precise similarity search, at an expense of some imprecision in results, and use different approaches: (1) by exploiting transformation of the metric space and (2) by reducing the subset of data to be examined (ZEZULA et al., 2005), the latter being the mainly approach used in this dissertation. In this context the Spearman Footrule Distance (SFD) is a method for assessing the accuracy of approximate similarity by measuring the discrepancy between two ordered (ranked) lists.

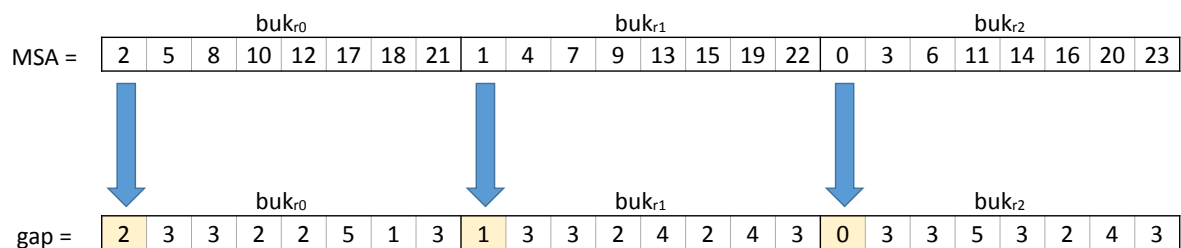
Simultaneously to improve of searching performance, we also pursuit ways of spare memory for the data structure constructed for those approximate similarity searches. So, for this, we propose the development of a compact structure that handles approximate similarity searches directly, without data unpacking. (NAVARRO, 2016) presents several data coding methods and techniques and, from those, the Elias Delta and Simple-9 are employed in our compact data structure due to its relative simplicity to implement, but still providing notable memory spare performance. Also those two are well know coding techniques, which make them standard methods in the data compression field.

The Compact Metric Suffix Array

In this chapter we present new strategies intended to reduce memory in spite of time to proceed in the building of the MSA and performing a query. This gain in memory could be quite beneficial to applications which could not afford to deal with secondary memory and can suffer in performance when such data structure need to be divided in different memory hierarchy.

Therefore, the first feature that can be exploited is that the MSA is partially sorted, that is, due to its specific bucket-distributed elements, one can note that inside a bucket, the elements display an ascending order. So, instead of storing those indexes in the original form, we can get the differences between consecutive positions, but maintaining the first element of the bucket, in the way shown in the Figure 25.

Figure 25 – Difference array



Due to how the MSA is built, it is easy to note that, when taking the differences between neighbor indexes, the maximum value would be $2 \times n - 1$, where n is the number of references, because in the worst case scenario a reference r_i could be the nearest to object o_k and the farthest to the next object o_{k+1} , which case the two indexes in the MSA would be in a difference of $2 \times n - 1$. So, for the case in analyses, shown in Figure 25, the largest value is 5, because $(2 \times 3) - 1 = 5$. Also, in the best case scenario, the difference value would be 1, which occurs when a reference r_i is the farthest to object o_k and the nearest to the next object o_{k+1} .

This property is very convenient because, usually, the number of references is much smaller than the number of objects. For instance, in our experiments, objects are in the order of millions while references are in the order of hundreds. Therefore, we can avoid

storing very large numbers in the MSA and instead have a maximum value of $2 \times n - 1$, which is considerably smaller, hence allowing us to reduce memory by using less bits to store those values.

3.1 Compact MSA: construction

The construction of the MSA starts by taking the indexes of the first object and stores these indexes at the respective first position of each bucket as it is originally done in the MSA, as show in Figure 25. However, in compressed MSA, these values are stored using a method of compression so the construction of this data structure is made directly without the necessity of constructing the original MSA and then compress it.

The next elements of each bucket are obtained by taking the differences of consecutive elements. For instance, as shown in Figure 25, in original MSA, the second position of bucket buk_{r_0} stores value 5; in compressed MSA the value stored is the difference between the second and first positions, which results 3. The same is applied to every remaining positions of a bucket, using some method of codification, assuring that this data structure is constructed directly compressed.

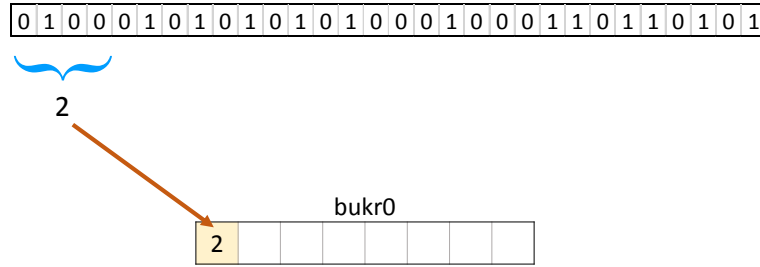
For the Elias Delta Code version of compressed MSA, the values are immediately coded and stored as an array of 1-bit elements. For the Simple-9 version of compressed MSA, an auxiliary buffer array is used to store integers until they can be coded encapsulated in a 32-bit word. In both cases, the processes allows a direct construction of compressed MSA.

3.2 Compact MSA: searching

To perform a K -NN search, the values stored in the compressed MSA can obtained by starting from each bucket of the compressed MSA and reversing the codification process.

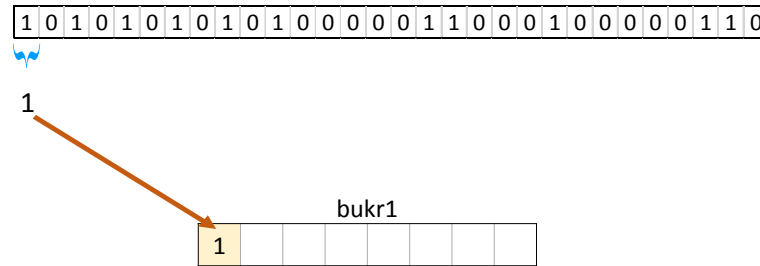
The original MSA proceeds by reading sequentially a bucket at a time, obtaining only at the end of the entire process, the Acc array (Figure 20), which would require N elements (where N is the number of elements of the dataset). This procedure poses an issue to the compressed MSA, as the structure is constructed using differences. Instead, we propose to perform an object at a time, so the reading could be performed directly, without the necessity to retrieve the uncompressed MSA to perform a searching.

Therefore, for the Elias Delta Code version, Figure 26 shows the first step which takes the first 4 bits of bucket buk_{r_0} , reversing the code and obtaining the index 2. From this value, we get $2 \bmod 3 = 2$, which means that reference r_0 is the farthest from object o_0 . For instance, suppose we are executing the query q shown in Figure 5: in this case, the query ordered list obtained is $L_q = 1, 0, 2$. Therefore, the algorithm proceeds by

Figure 26 – Obtaining first element (2) from codified bucket buk_{r_0} 

comparing the reference r_0 rank for object o_0 (value 2) with the reference r_0 rank for query (value 1) as established by Equation 3, so we get $acc = |2 - 1| = 1$.

Following, the algorithm proceeds to bucket buk_{r_1} , takes the first bit, reversing the code and obtaining the index 1, as show in Figure 27. As $L_q = 1, 0, 2$, to parameter acc is added the value $|1 - 0| = 1$. Finally, inspecting bucket buk_{r_2} we obtain value 0 as a rank to reference r_2 to object o_0 , comparing to L_q we get value $|0 - 2| = 2$, which is added to acc , resulting in $acc = 4$, which is the SFD distance to query q to object o_0 .

Figure 27 – Obtaining first element (1) from codified bucket buk_{r_1} 

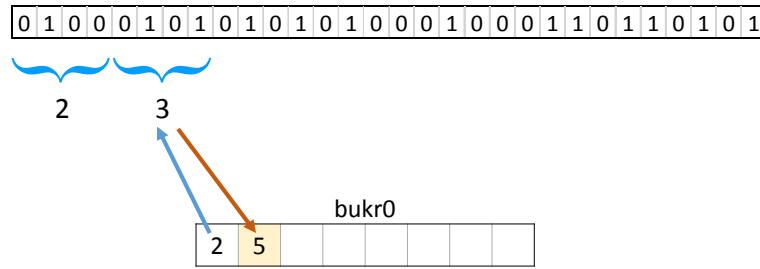
In a $K - NN$ search, we can spare memory by storing just the K nearest objects from query q . To do so, once an object SFD distance is calculated, this object is sent to a minimum priority queue, where it is compared with the K objects stored. The natural ordering of priority queue has the object with the smallest acc value at the head of the queue and thus the ordering is ascending.

Once object o_0 is sent to priority queue, we proceed to the second index of bucket buk_{r_0} , which is obtaining from the next four bits as shown in Figure 28. The value decoded is 3 which is added to the last index to obtain the index 5 which is placed in the second position of the bucket.

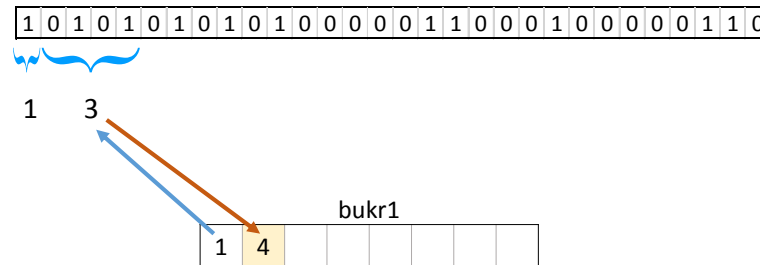
Note that the reading of a bucket is made sequentially, therefore only the last calculated index is stored: it is not required to decode and store the entire bucket. Hence, at Figure 28 stage, only the value 5 is stored for bucket buk_{r_0} for the next object.

As $5 \bmod 3 = 2$, that means reference r_0 is the farthest from object o_1 , and comparing with $L_q = 1, 0, 2$, we get $acc = |2 - 1| = 1$.

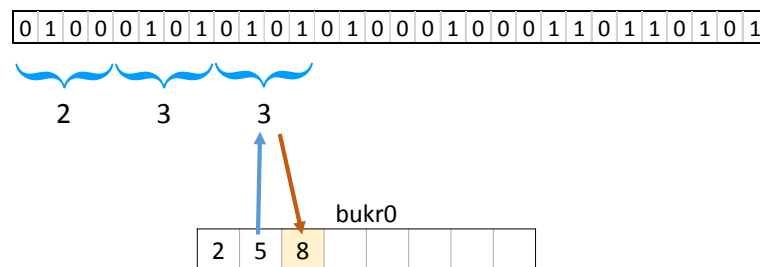
Figure 29 shows the next step, which gets the next 4 bits of bucket buk_{r_1} , which decoded has value 3. Accumulate with the last stored value, 1, results in 4, meaning that

Figure 28 – Obtaining second element (5) from codified bucket buk_{r_0} 

reference r_1 is rank 1 ($4 \bmod 3 = 1$). Therefore, comparing with L_q , $|1 - 0| = 1$, which is added to acc . Finally, proceeding the same for bucket buk_{r_2} results in $acc = 4$.

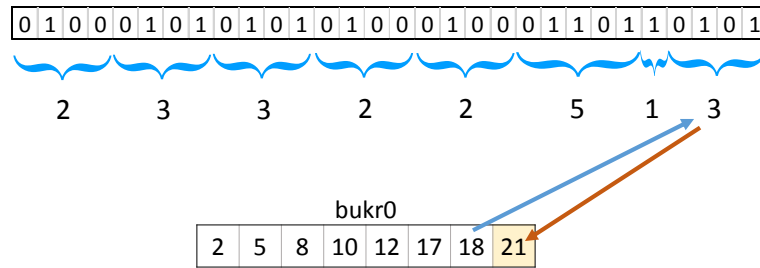
Figure 29 – Obtaining second element (4) from codified bucket buk_{r_1} 

Now the procedure repeats for the object o_2 for bucket buk_{r_0} , as shown in Figure 30. The value stored for bucket buk_{r_0} is 5, which is accumulated to the decoded value 3, resulting 8. As $8 \bmod 3 = 2$, $acc = |2 - 1| = 1$. This is repeated for buckets buk_{r_1} and buk_{r_2} , resulting in $acc = 4$. Object o_2 is sent to the priority queue.

Figure 30 – Obtaining third element (8) from codified bucket buk_{r_0} 

Repeating the process for the remaining string we obtain the complete bucket buk_{r_0} directly as shown in Figure 31. The last object, o_7 is compared to L_q resulting in $acc = 2$, then this object is sent to the priority queue. At the end, the priority queue stores only the K nearest objects from query q .

Note that although Figure 31 shows bucket buk_{r_0} entirely decoded, this is just for presentation sake: in reality, the procedure stores just the previous object index for each bucket. This guarantees that only n values are stored instead of the much larger $n \times N$ values that would be necessary for the entire decoding of the uncompressed MSA. That way, the searching process is concluded directly, without decompressing the cMSA.

Figure 31 – Obtaining last element (21) from codified bucket buk_{r_0} 

3.3 Compact MSA: summary

The compact MSA proposed in this work uses the data structure established by the MSA but, instead of storing the permutation position number directly in the array, as $N \times n$ typically achieving or even surpassing the order of 2^{32} , our strategy is based on the idea of taking the difference between consecutive elements, which results in much smaller decimals and then coding those values using standard coding methods. As a result our proposal are able to achieve better performance in memory storage than the MSA, at a linear cost of construction and searching. It is worth to mention that the compact data structure generated by our strategy can perform queries without the necessity of unzip the data structure: the procedure of searching is performed directly from the compact array. As another benefit, our data structure is not limited to 32- or 64-bit primitives, as the number of references or objects can achieve any size, being limited only by the storage memory size.

Experimental Results

For experiments the MSA and the two versions of the cMSA using Elias Delta and Simple-9 are implemented using C/C++. The three versions were implemented in a way to maximize resource sharing to obtain the fairest possible comparison: same functions and variables were used as much as possible. It is worth mention that before the final C/C++ code, Wolfram Mathematica was used to prototype some parts of the code, as well to provide some of the figures used in this work and to validate integer encoding for the Elias Delta. The three versions were also implemented in Java before, due to its debug easiness. Some aspects of C++ were expected to be used, like dynamic arrays, but ultimately were not, making the code mainly C compatible.

4.1 Calculating array size beforehand

The first draws of cMSA using the Elias Delta Code and Simple-9 was implemented using C++ native dynamic structures as `vector`. However, this approach exhibit unexpected memory allocation random behavior when compared with the estimated theoretical use of memory: the peak memory could achieve almost two fold of what was expected. This happens due to how those aforementioned C++ dynamic structures behave behind scenes. Therefore, as the the goal of this work is gain in memory allocation, another approach was established: instead of using those structures, we calculate the size of final structure before and, only then, allocate an standard C++ array. This approach increased the time required to construct the cMSA structure, however proved to solve the erratic memory allocation behavior as the memory peak allocates exactly what was expected.

4.2 Tests

The central idea behind the tests is to compare how the two versions proposed in this work stands against the MSA. As the goal of these projects are to spare memory, so it is expected that the time for generate the main data structure (what will be called here

indexing) that will be subsequently used for answering demands like k-nearest objects from a position (will be called here searching) be larger than the MSA. Therefore, we tested three versions: the MSA and the two compact versions proposed in this dissertation, the cMSA Delta and the cMSA Simple-9.

To summarize the experiments performed, Table 4 shows that for each subset of the data employed as input (from 1 to 16 million objects), five experiments with different number of references were performed, from 128 to 2048 references. It was performed five different object number inputs, therefore 25 experiments were performed in total. The value 3 in Table 4 denotes those experiments where all three version were able to perform. The value 2 denotes those experiments where the MSA was not performed due to the restriction of the 32-bit primitive type, which prevents experiments where $N \times n > 2^{32}$. For every experiment, it was registered the indexing time, searching time and memory usage for each version. The searching time values are obtained as an average for 20 queries.

Table 4 – Summary of the experiments performed

Number of objects (N)	Number of references (n)				
	128	256	512	1024	2048
1M	3	3	3	3	3
2M	3	3	3	3	3
4M	3	3	3	3	2
8M	3	3	3	2	2
16M	3	3	2	2	2

The tests were executed on a computer configured with an AMD Ryzen 9 5950X 16-Core processor, 64 GB memory RAM, SSD NVMe 1 TB, Arch Linux kernel 6.4.7-arch1-3 (64 bits). Due to memory limitation, tests that require a memory peak larger than 64 GB were not performed. Also, it is worth mention that the MSA rely on the limitation of 32 bit integers, so tests with N objects and n references, where $N \times n > 2^{32}$ were not possible to be performed. Such limitation is not observed in the two versions of cMSA because those utilize the differences as described in Chapter 3.

For the set of tests it was employed two datasets: ANN_SIFT1B¹ (JÉGOU; DOUZE; SCHMID, 2011) and CoPhIR² (BOLETTIERI et al., 2009). The ANN_SIFT1B *Learning Set* dataset contains 100 million objects with 128 dimensions. The CoPhIR dataset contains 10 million objects with 64 dimensions. These datasets were chosen because they emulate image database of real world applications, that are likely to contain such number of high dimensional entries.

In this section, for purpose of naming definition, a *test* is a set of 20 queries performed in a context of an specific number of objects and references. For instance, for a case with

¹ The ANN_SIFT1B is divided in 4 sets: the set used here is the *Learning Set*, which has 100 million registers. Source: <http://corpus-texmex.irisa.fr>

² Source: <http://cophir.isti.cnr.it>

256 references and 1 million objects, we perform 20 queries to complete a *test*. Changing the number of references or objects creates another *test*.

As a first case, it was used a 256 references fixed to different number of objects. To yield Figure 32 it were tested 5 inputs of objects: 1, 2, 4, 8 and 16 million of objects. The goal in this case is to analyse the behavior of the cMSA versions when the object number is increased. The aspects analyzed for the cMSA versions are indexing time, searching time and peak memory.

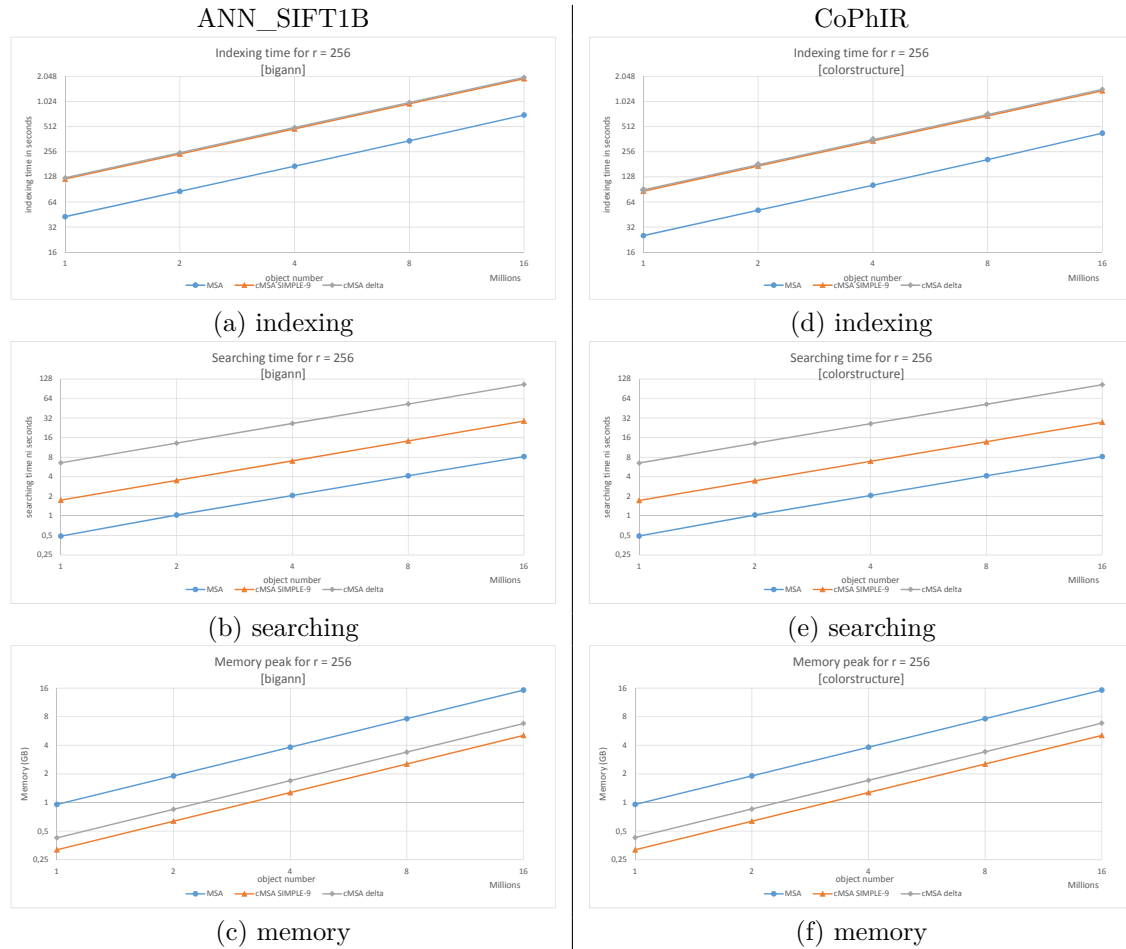
In Figure 32(a), tested with the ANN_SIFT1B dataset as input, is observed that the indexing time for cMSA Delta and cMSA Simple-9 are larger than the MSA as expected. Also, it can be noted that the MSA and cMSA versions increase in time as the number of objects gets larger. When comparing with the tests performed using the CoPhIR dataset as input, as shown in Figure 32(d), it can be observed a slight reduction in the indexing time of the MSA and cMSA versions, which is explained by the smaller dimensionality of this dataset (64 against 128). Both Figure 32(a) and Figure 32(d) show a linear behavior in the relation of indexing time and object numbers input. Also, it can be observed for both datasets that both cMSA versions take similar indexing time.

Figure 32(b) and Figure 32(d) show the searching time for the three algorithms to 256 references varying the number of objects input of the ANN_SIFT1B and CoPhIR datasets respectively. As expected, both graphs are similar once, differently from the indexing time, where the dimensionality of the input data makes an impact on the number of mathematical calculations. Now, at the searching step, the number of input dimensions become indifferent once the data structure obtained do not rely on the dimension size of the input. As in the indexing step, the searching time increases as the number of objects increases, in a linear behaviour. Differently of the indexing step, the searching time of cMSA versions are distinct, with a considerable advantage for the cMSA Simple-9. When compared to the cMSA Delta, the cMSA Simple-9 requires a simpler decoding, which allowed faster searching time.

Figure 32(c) and Figure 32(f) show the peak memory for the MSA and the two solutions proposed in this work. The expression 'memory peak' are used here to express the maximum value of memory required when running an algorithm. As expected the two graphs are similar once the dimensionality difference between the two dataset delivers negligible variation on the data structure generated by the algorithms. Figure 32(c) and Figure 32(f) demonstrate a better performance of the cMSA algorithms compared to the MSA, where the cMSA Simple-9 required slightly less memory than cMSA Delta.

Figure 33(a) and Figure 33(d) show for both datasets the comparison of indexing time of the two cMSA algorithms to the MSA. The MSA is showed as a line at value 1. Figure 33(a) shows that for the ANN_SIFT1B dataset the cMSA Simple-9 takes approximately 2.8 times the time required for the MSA to generate the data structure, while the cMSA Delta takes slightly more time, around 2.9 times. One can note an slight

Figure 32 – Tests with the sets of ANN_SIFT1B and CoPhIR for 256 references



decrease in the relative indexing time for both cMSA algorithms as the number of objects input increases.

Figure 33(b) and Figure 33(e) show the searching time ratio of the three algorithms to the MSA. Again, the MSA is shown as a constant line at value 1. To perform a query the cMSA Simple-9 requires approximately 3.5 times the time required by the MSA. As expected the cMSA Delta, due to a more time consuming decoding step, takes more time to perform the same operation and requires around 13.7 times more time than the MSA.

Figure 33(c) and Figure 33(f) compare the two cMSA versions with the MSA in terms of memory usage by showing the compression rate. Both datasets display similar behavior as the dimensionality different makes almost no impact on the size of the data structure generated. At 256 references the number of objects input shows no significant difference (less than 1% difference) and cMSA Delta takes 44% of the MSA memory usage, while cMSA Simple-9 achieves 33% of the MSA memory usage.

Now, to get notion on how the algorithms behave with variation of the number of references, Figure 34 shows results for an input of 2 million objects for both ANN_SIFT1B and CoPhIR datasets. As in Figure 32 it can be observed a linear behavior between the number of references and the output attributes of time.

Figure 33 – Comparing the cMSA with MSA for 256 references

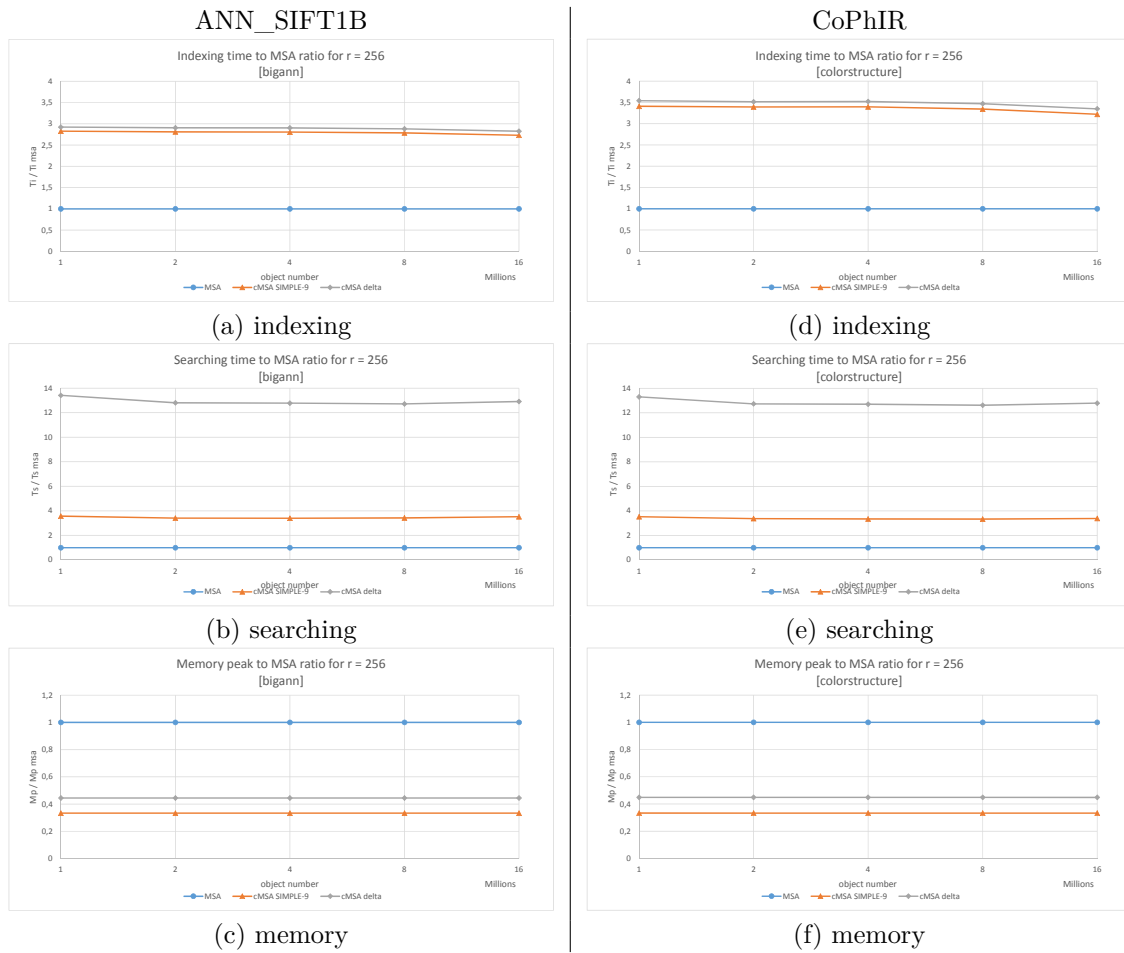


Figure 34(a) and Figure 34(d) show how the increase of references translates in the increase of indexing time. Again, the CoPhIR dataset tests require less time than the ANN_SIFT1B due to the lower dimensionality. Both cMSA algorithms require almost the same time at the indexing step and both take more time than the MSA as expected.

Figure 34(b) and Figure 34(e) show the test results for the three algorithms searching time. The cMSA Delta takes more time at this step than the cMSA Simple-9 as the later offers a simpler decoding procedure which reflects in a better performance. Again, both cMSA versions take more time at searching than the MSA. It is worth to mention the slight spike at 512 references, which makes an otherwise straight line a little jagged. This happens because of the transition of the average number of integers encoded inside a 32-bit word: as the number of references increases, the average differences between the neighbour elements at the *buckets* of the data structure also increases and eventually they achieve a value that requires more bits to represent, consequently decreasing the number of values encoded inside a 32-bit word than it is required at 256 references. So it takes more time at searching because the data structure had a sudden increase in size compared to the 256 references test.

Figure 34(c) and Figure 34(f) display the memory required for the three versions:

both cMSA algorithms show almost the same memory usage and they are smaller than the MSA. Here the slight spike in memory usage by cMSA Simple-9 at 256 references is explained again by the sudden decrease in the number of integers encoded in a 32-bit word at 512 references.

Figure 34 – Tests with the sets of ANN_SIFT1B and CoPhIR for 2 million objects

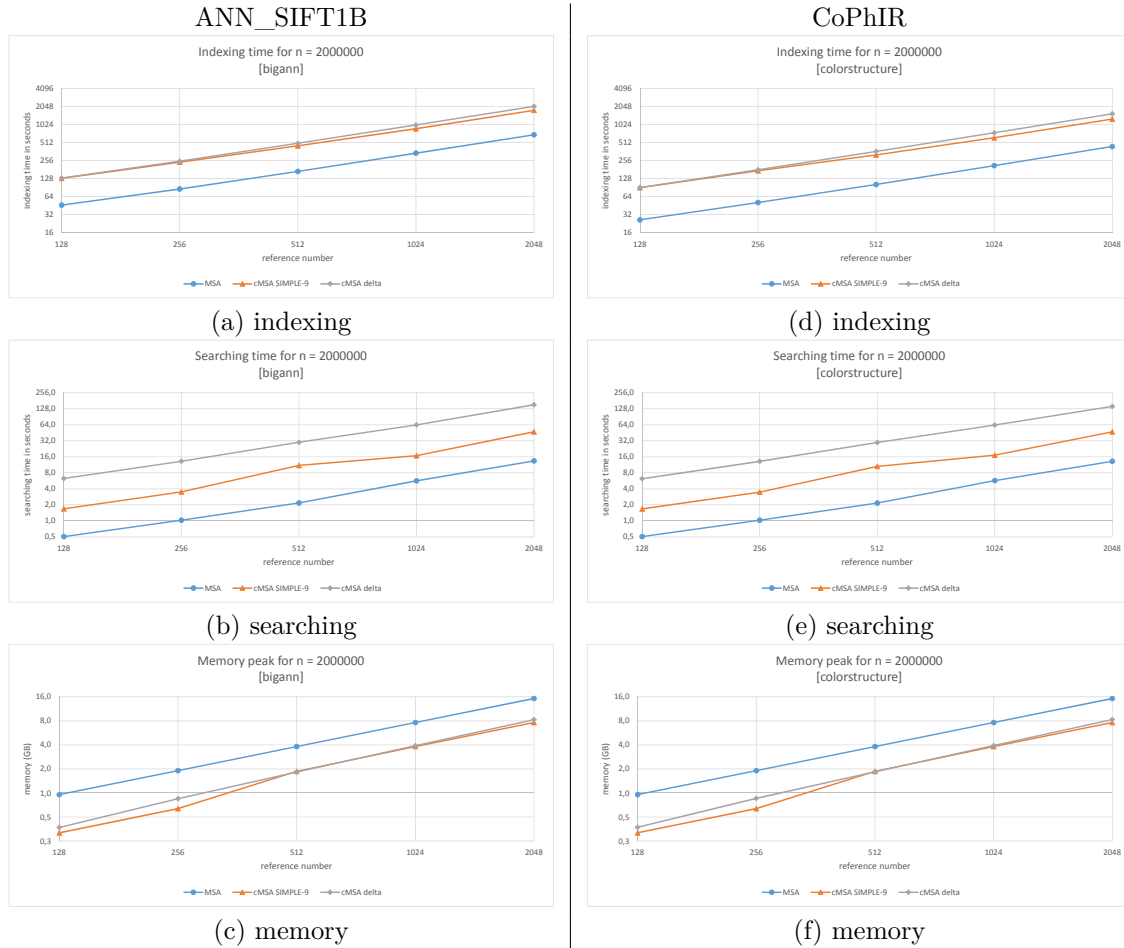


Figure 35 displays how the cMSA versions perform compared with the MSA for ANN_SIFT1B and CoPhIR datasets to an input of 2 million objects varying the number of references.

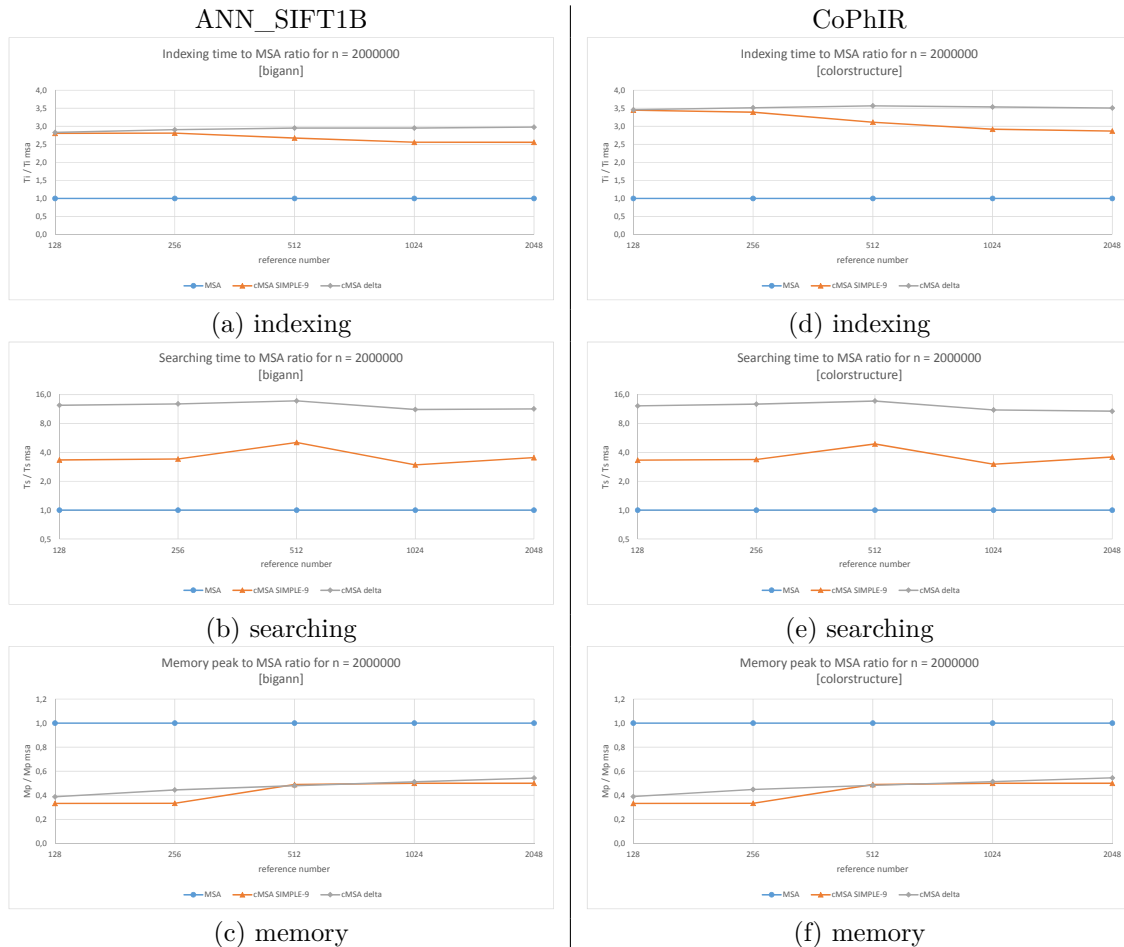
Figure 35(a) and Figure 35(d) show indexing time ratio for the three algorithms compared to the MSA. The MSA is shown as a line at value 1. For both datasets, it can be seen that both cMSA versions require similar indexing time at 128 references, but then, the cMSA Simple-9 shows a better performance with the increase of references. At 2048 references the cMSA Delta version approaches 3 times the indexing time required for the MSA, while the cMSA Simple-9 achieves 2.5 times for the ANN_SIFT1B dataset. The two graphs also show interesting information: comparatively, the higher dimensionality provides better performance for both cMSA versions, that is, increases in dimensionality implies a decrease in indexing time for cMSA when compared to the MSA. That is why for the CoPhIR dataset, with data with smaller dimensionality, the cMSA Simple-9 and

cMSA Delta achieve 2.9 and 3.5 times respectively the searching time required by the MSA.

Figure 35(b) and Figure 35(e) show the performances of the versions at the searching step compared to the MSA. Tests for both datasets behaved similarly, with the cMSA Delta requiring more searching time around 12 times required by the MSA, while the cMSA Simple-9 requiring approximately 3.5 times the searching time required by the MSA, with a spike of 5.1 with 512 references due the transition to larger difference integers encoded in a 32-bit word.

Figure 35(c) and Figure 35(f) show the comparison of memory usage of the three algorithms. Again, the performances are similar for both datasets, with both cMSA versions requiring less memory than the MSA, but showing an increasing relative memory usage as the number of references increases. These graphs show a jagged behavior of the cMSA line due to the transitions of integer binary sizes required for this method. At 128 references, the cMSA Delta and cMSA Simple-9 require 0.4 and 0.3 times memory the MSA, as this values increase as the number of references increases, achieving 0.5 for both cMSA versions with 2048 references.

Figure 35 – Comparing the cMSA with MSA for 2 million objects

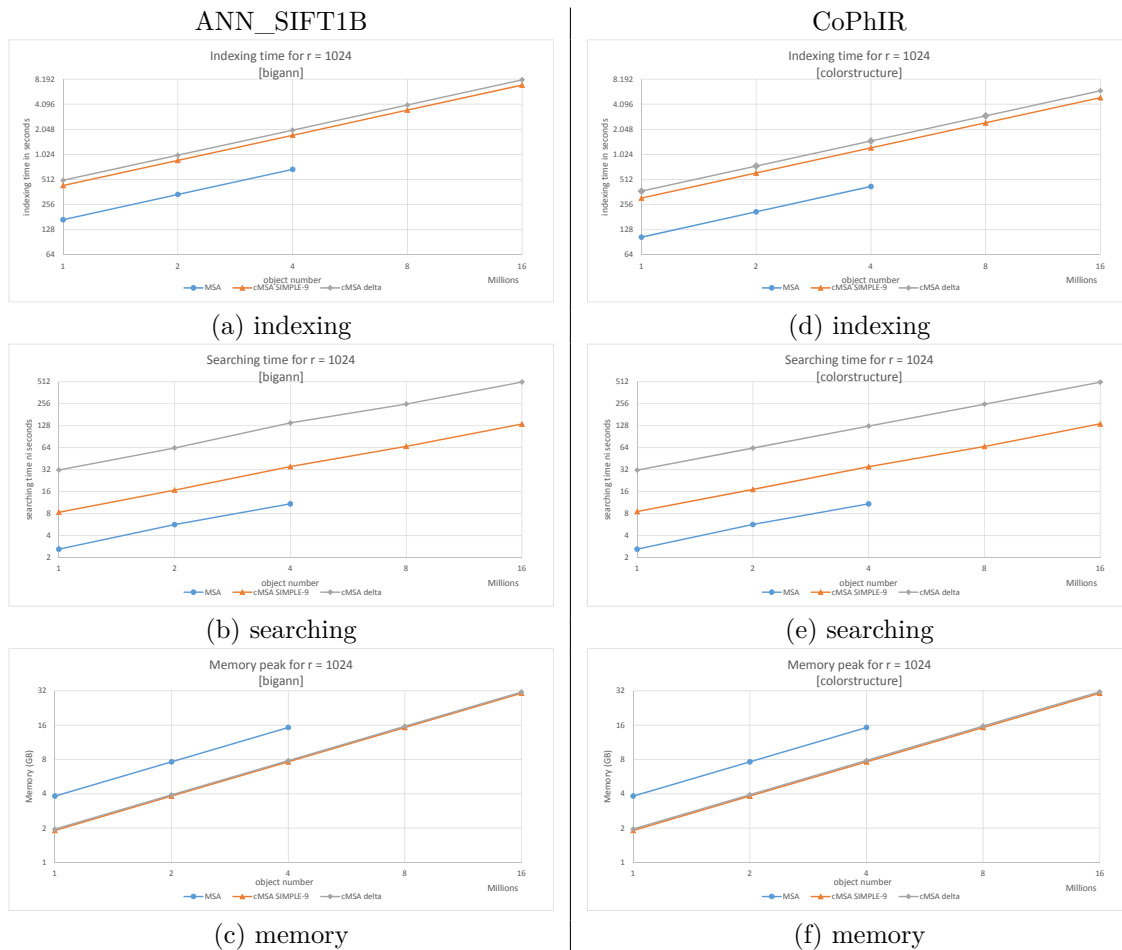


As a second case, it was used a 1024 references fixed to different number of objects.

Figure 36 shows tests for different number of objects. However, differently from the previous experiment, the MSA cannot be tested beyond 4 millions objects input, as the MSA data structure uses 32-bit integers. To accommodate large number of references \times objects it would result in a larger data structure: for instance, using 64-bit integers would result with the MSA requiring twice the memory, which would require an entire new batch of tests that would rend the MSA to consume even more memory when compared to the two cMSA versions.

Figure 36(a) and Figure 36(d) show indexing time, where it can be seen that for $r = 1024$ and $n = 16$ millions, cMSA Delta achieves 8000 seconds and cMSA Simple-9 requires 7000 seconds. At $n = 4$ millions, the MSA, cMSA Simple-9 and cMSA Delta require 680, 1730 and 2000 seconds respectively. Figure 36(b) and Figure 36(e) show searching time, where it can be seen cMSA Simple-9 performs better than the cMSA Delta. Figure 36(c) and Figure 36(f) show memory usage, where it can be seen that the MSA achieve 16 GB for $n = 4$ millions objects while both cMSA versions go to slight over 32 GB for $n = 16$ millions objects. Doubling this number would require slight more than 64 GB which was beyond the memory supported by the test machine.

Figure 36 – Tests with the sets of ANN_SIFT1B and CoPhIR for 1024 references



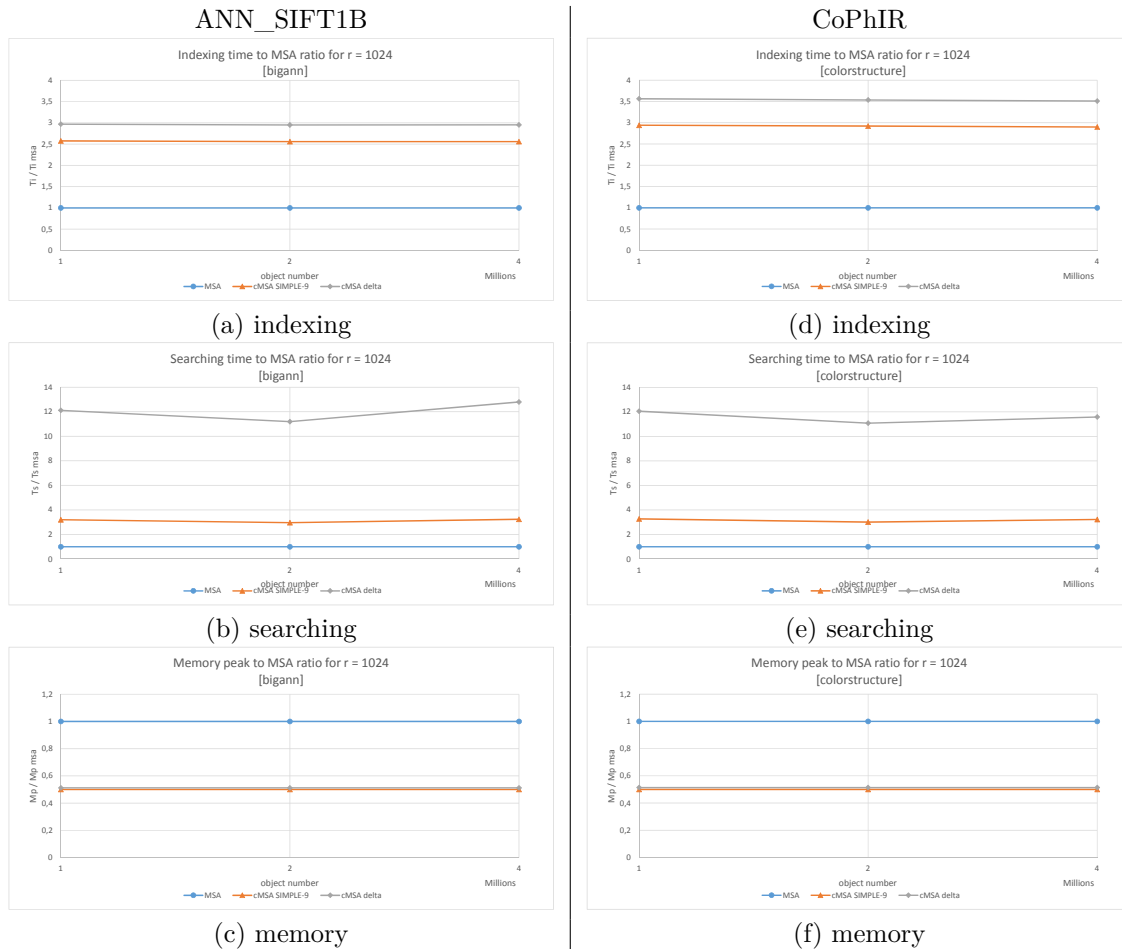
Now, Figure 37 compares the cMSA versions with the MSA for 1024 references. Fig-

ure 37(a) shows that the cMSA Delta requires 3 times the indexing time of the MSA while the cMSA Simple-9 takes 2.5 times the indexing time of the MSA for the ANN_SIFT1B dataset. Figure 37(d) shows that the cMSA Delta requires 3.5 times the indexing time of the MSA while the cMSA Simple-9 takes 3 times the indexing time of the MSA for the CoPhIR dataset. This shows that, comparatively to the MSA, the cMSA versions performs better with the increase of data dimensionality.

Figure 37(b) and Figure 37(e) show that the cMSA Delta requires around 11 times the MSA searching time while the cMSA Simple-9 performs much better, requiring around 3 times the MSA searching time.

Figure 37(c) and Figure 37(f) show that for 1024 references, the two cMSA versions require around 50% of the memory used by the MSA. When compared with Figure 33 it can be seen that increasing the reference number reduces the data compression capability of both cMSA versions as expected, as the increasing of reference numbers imply in increasing of the difference integers encoded.

Figure 37 – Comparing the cMSA with MSA for 1024 references



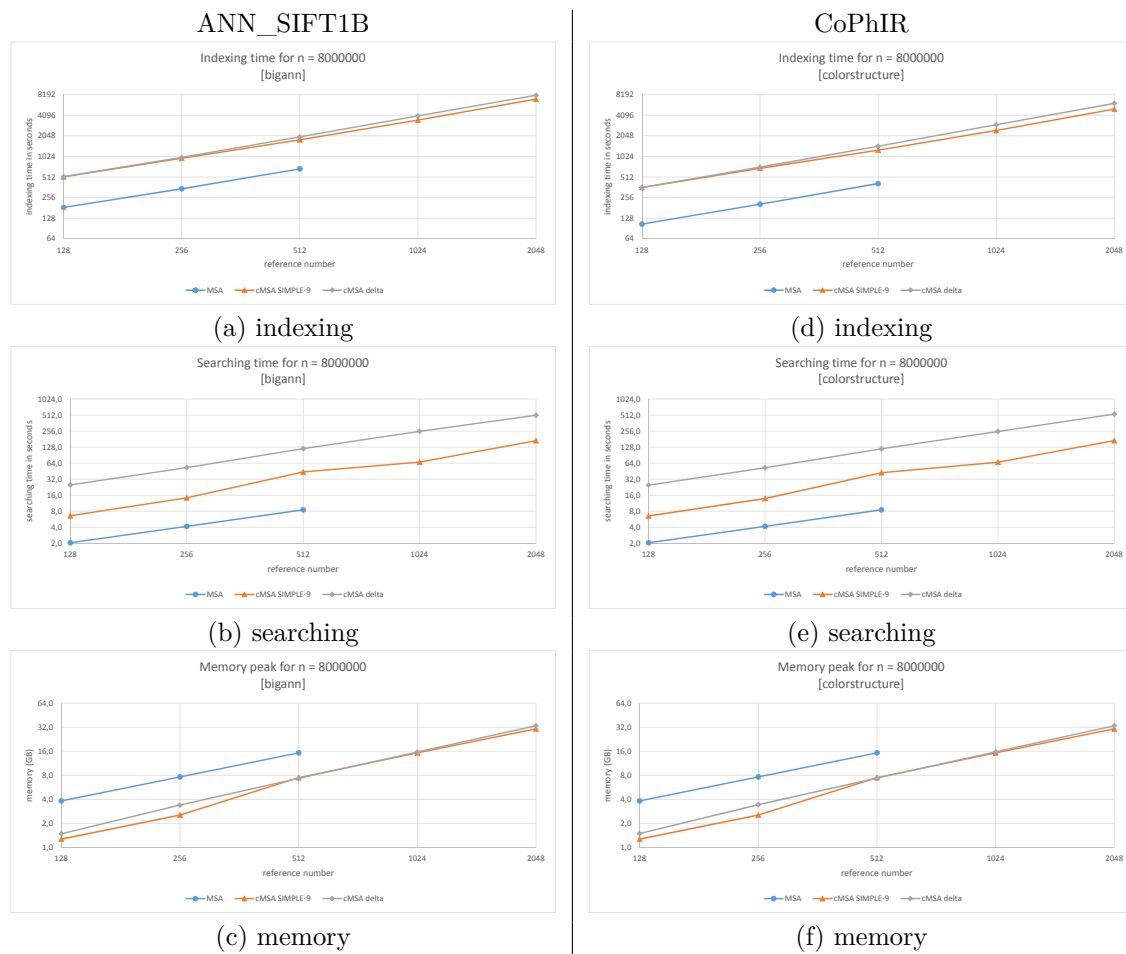
For the second case, Figure 38 shows tests with 8 million objects while varying the number of references. Here again, the MSA was tested to 512 references only as it is limited to number of references \times number of objects, which must be smaller than 2^{32}

due to the 32-bit integer data structure limitation. Both Figure 38(a) and Figure 38(d) show similar performance for both cMSA versions, where the ANN_SIFT1B dataset tests taking more time due to its larger data dimensionality.

Figure 38(b) and Figure 38(e) show searching time, displaying better performance for the cMSA Simple-9.

Figure 38(c) and Figure 38(f) show memory usage, where both cMSA versions achieving 32 GB for 2048 references, while the MSA could only went for 512 references, taking 16 GB.

Figure 38 – Tests with the sets of ANN_SIFT1B and CoPhIR for 8 million objects



Comparing the MSA results for 8 million objects, Figure 39 shows the performance of the cMSA versions. Figure 39(a) and Figure 39(d) show again that higher dimensionality delivers better performance from both cMSA versions as for the ANN_SIFT1B dataset both versions stay peak under 3 times the indexing time required by the MSA while for the CoPhIR requires around 3.5 times the indexing time at same step.

Figure 39(b) and Figure 39(e) show that the cMSA Delta demands around 13 times the MSA searching time while the cMSA Simple-9 requires 4 times the time.

Figure 39(c) and Figure 39(f) show reduction in memory saving for both cMSA versions as the references number increases, reaching 50% of the MSA memory usage when 512

references are used.

Figure 39 – Comparing the cMSA with MSA for 8 million objects

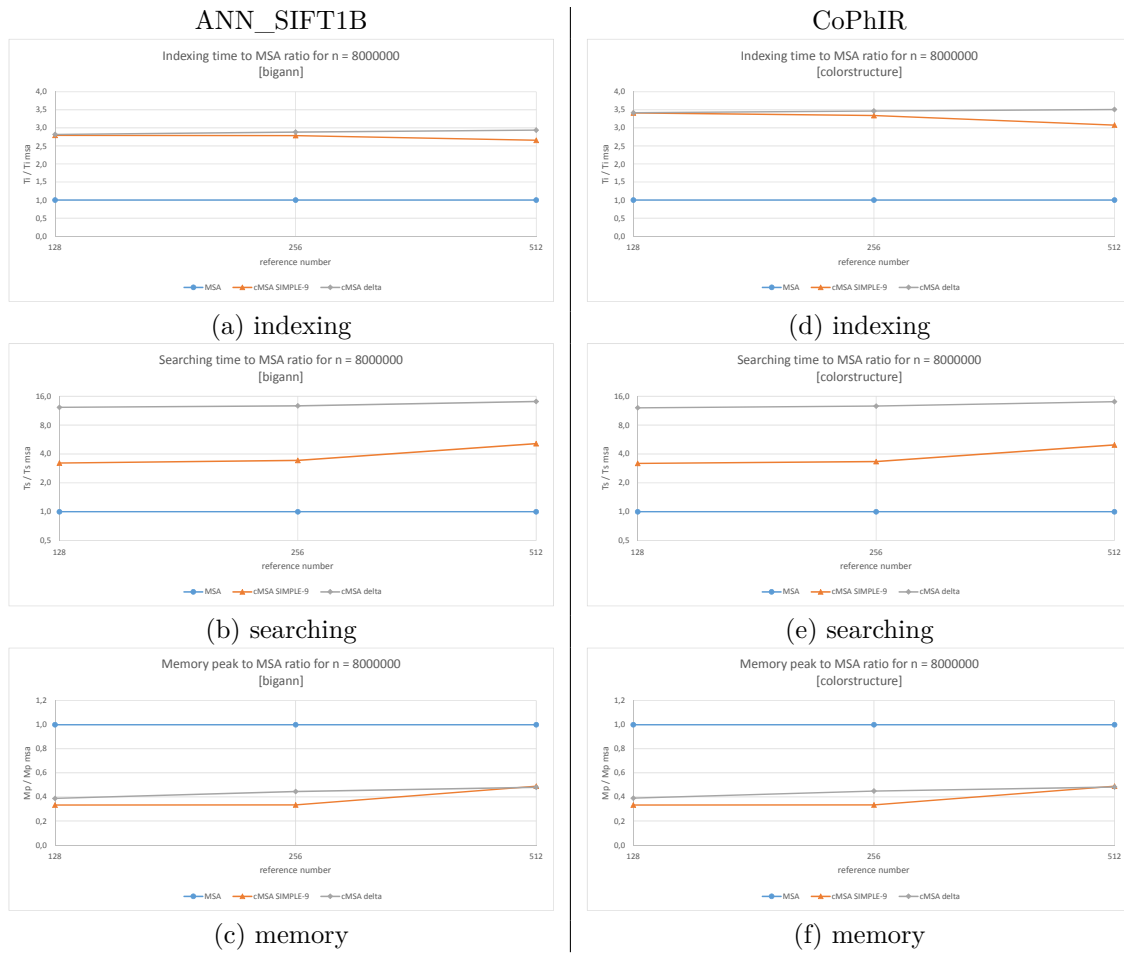


Table 5 shows the indexing time required by cMSA Simple-9 (shown as S_9) and cMSA Delta (shown as δ) varying the number of objects from 1 to 16 millions and the number of references from 128 to 2048 compared with the indexing time required by the MSA at comparable scenario. As the MSA stores values up to $N \times n$, experiments were those values surpasses 2^{32} are not executed. Therefore, despite the equivalent experiments with the compact versions are indeed executed, they are no shown because of the lack of a MSA equivalent experiment to compare.

For instance, Table 5 shows that for 1 million objects and 128 references, the cMSA Simple-9 (S_9) and the cMSA Delta (δ) took 2.8 times the indexing time required by the MSA running at the same condition. Another example, for 2 million objects and 2048 references, the cMSA Simple-9 (S_9) and the cMSA Delta (δ) took 2.6 and 3.0 times respectively the indexing time required by the MSA running at the same condition.

Table 6 shows, compared to MSA, the searching time required by cMSA Simple-9 (shown as S_9) and cMSA Delta (shown as δ) varying the number of objects from 1 to 16 millions and the number of references from 128 to 2048. For instance, for 1 million objects

and 128 references, the cMSA Simple-9 (S_9) and the cMSA Delta (δ) took 2.6 and 3.0 times respectively the indexing time required by the MSA running at the same condition.

It is worth to mention the similarity of the results between the two datasets: what distinguish the two datasets are the number of dimensions, that is, the number of parameters related to each object. The ANN_SIFT1B dataset has 128 parameters for each object, two times the number of parameters for object of the CoPhIR dataset. After indexing step, the data structure produced by the MSA and cMSA versions are dimensionally equivalent for both datasets. Therefore, all operations and calculations are equivalent time and space wise, which explains the similarity showed by the results presented here.

The results obtained for the memory required by cMSA Simple-9 (shown as S_9) and cMSA Delta (shown as δ) when compared to MSA with the same input is shown in Table 7. The best results were obtained, as expected, with smaller number of references, where, at 128 and 256 references, the cMSA Simple-9 (S_9) achieved 33% of the memory required by the MSA and the cMSA Delta (δ) required 39% of the memory used by the MSA when running with 128 references and 44% with 256 references. On the other side, with 2048 references, the cMSA Simple-9 (S_9) and the cMSA Delta (δ) required 50% and 54% of the memory used by the MSA respectively.

Table 5 – Indexing time for cMSA Simple-9 (S_9) and cMSA Delta (δ) compared to MSA obtained in experiments

Ref	128		256		512		1024		2048	
	S_9	δ	S_9	δ	S_9	δ	S_9	δ	S_9	δ
1M	2.8	2.8	2.8	2.9	2.7	3.0	2.6	3.0	2.6	3.0
2M	2.8	2.8	2.8	2.9	2.7	3.0	2.6	3.0	2.6	3.0
4M	2.8	2.8	2.8	2.9	2.7	2.9	2.6	3.0	-	-
8M	2.8	2.8	2.8	2.9	2.7	2.9	-	-	-	-
16M	2.7	2.8	2.7	2.8	-	-	-	-	-	-

Table 6 – Searching time for cMSA Simple-9 (S_9) and cMSA Delta (δ) compared to MSA obtained in experiments

Ref	128		256		512		1024		2048	
	S_9	δ	S_9	δ	S_9	δ	S_9	δ	S_9	δ
1M	3.4	12.0	3.6	13.4	5.6	15.3	3.2	12.1	4.1	11.1
2M	3.3	12.4	3.4	12.8	5.1	13.8	3.0	11.2	3.5	11.4
4M	3.3	12.3	3.4	12.8	5.0	13.7	3.2	12.8	-	-
8M	3.2	12.3	3.4	12.7	5.1	14.1	-	-	-	-
16M	3.2	12.2	3.5	12.9	-	-	-	-	-	-

Table 7 – Memory allocated by cMSA Simple-9 (S_9) and cMSA Delta (δ) in percent to the memory allocated by MSA obtained in experiments

Ref	128		256		512		1024		2048	
Obj	S_9	δ	S_9	δ	S_9	δ	S_9	δ	S_9	δ
1M	33%	39%	33%	44%	49%	48%	50%	51%	50%	54%
2M	33%	39%	33%	44%	49%	48%	50%	51%	50%	54%
4M	33%	39%	33%	44%	49%	48%	50%	51%	-	-
8M	33%	39%	33%	44%	49%	48%	-	-	-	-
16M	33%	39%	33%	44%	-	-	-	-	-	-

Conclusion

Several applications could benefit from even modest memory reductions as it can mean the difference between maintain the entire data structure at RAM memory or have to send chunks of data to disk, which can result in very slow accesses. That is why alternative solutions like those proposed in this text, that can spare until two thirds of the memory require by the MSA method, even at the cost slower operations of indexing and searching, can be considered viable: the more they postpone memory pagination, the better.

The work presented in this dissertation was published and presented at the *38° Simpósio Brasileiro de Bancos de Dados (SBBD) 2023* (ROSA; LOUZA; RAZENTE, 2023) as a short paper describing features of the MSA method and the encoding techniques used to implement the cMSA. It was briefly mentioned some test results and some applications for the cMSA.

5.1 Future improvements

The cMSA could benefit from some further analyzes, taking some improvements like the implementation of more sophisticated data encoding techniques or use of parallel computing.

As a further step to this work, it can be mentioned the parallelization of the indexation and searching steps. The construction of the data structure could be easily divided into buckets, as the bucket elements are not coupled with each other. In other words, there is no dependency between the elements of different buckets, as each bucket stores information related to a unique reference and, by taking the differences of the consecutive elements, only elements contained inside a bucket correlate to each other. This feature allows for multithread processing, avoiding that data conflict would not occur between different computational threads. Also, the decoding processes could be divided without problems, thus resulting in faster indexation and searching.

Another improvement to this work could be the use of more sophisticated data encoding techniques, that could achieve even higher data compression ratios, so the aspect of

memory spare could be enforced even more.

As another idea to improve the cMSA would be the use of some C++ library that could handle dynamically the data structure with no memory penalty as observed with the C++ standard dynamic array, so the assessment step could be skipped, resulting in a faster indexation step.

Bibliography

- AGUERREBERE, C.; BHATI, I.; HILDEBRAND, M.; TEPPER, M.; WILLKE, T. L. Similarity search in the blink of an eye with compressed indices. **Proc. VLDB Endow.**, v. 16, n. 11, p. 3433–3446, 2023. doi:10.14778/3611479.3611537. Disponível em: <<https://www.vldb.org/pvldb/vol16/p3433-aguerrebere.pdf>>.
- BOLETTIERI, P.; ESULI, A.; FALCHI, F.; LUCCHESI, C.; PEREGO, R.; PICCIOLI, T.; RABITTI, F. CoPhIR: a test collection for content-based image retrieval. **CoRR**, abs/0905.4627v2, 2009. Disponível em: <<http://cophir.isti.cnr.it>>.
- BOZKAYA, T.; ÖZSOYOĞLU, Z. M. Indexing large metric spaces for similarity search queries. **ACM Trans. Database Syst.**, v. 24, n. 3, p. 361–404, 1999. doi:10.1145/328939.328959. Disponível em: <<https://doi.org/10.1145/328939.328959>>.
- CHEN, L.; GAO, Y.; SONG, X.; LI, Z.; ZHU, Y.; MIAO, X.; JENSEN, C. S. Indexing metric spaces for exact similarity search. **ACM Comput. Surv.**, ACM, v. 55, n. 6, dec 2022. ISSN 0360-0300. doi:10.1145/3534963. Disponível em: <<https://doi.org/10.1145/3534963>>.
- CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **International Conference on Very Large Data Bases (VLDB)**. Athens, Greece: [s.n.], 1997. p. 426–435.
- FIGUEROA, K.; PAREDES, R.; CAMARENA-IBARROLA, A.; VILLELA, H. T. Improving the permutation-based proximity searching algorithm using zones and partial information. **Pattern Recognit. Lett.**, v. 95, p. 29–36, 2017. Disponível em: <<https://doi.org/10.1016/j.patrec.2017.04.012>>.
- FIGUEROA, K.; PAREDES, R.; REYES, N. New permutation dissimilarity measures for proximity searching. In: **11th International Conference on Similarity Search and Applications (SISAP)**. Lima, Peru: Springer, 2018. (LNCS, v. 11223), p. 122–133. Disponível em: <https://doi.org/10.1007/978-3-030-02224-2_10>.
- GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. In: **International Conference on Management of Data (SIGMOD)**. Boston, MA: [s.n.], 1984. p. 47–57. Disponível em: <<https://doi.org/10.1145/971697.602266>>.
- HJALTASON, G. R.; SAMET, H. Distance browsing in spatial databases. **ACM Transactions on Database Systems (TODS)**, ACM, New York, NY, v. 24, n. 2, p. 265–318, 1999. ISSN 0362-5915. doi:10.1145/320248.320255.

_____. Index-driven similarity search in metric spaces. **ACM Trans. Database Syst.**, v. 28, n. 4, p. 517–580, 2003. doi:10.1145/958942.958948. Disponível em: <<https://doi.org/10.1145/958942.958948>>.

JÉGOU, H.; DOUZE, M.; SCHMID, C. Product quantization for nearest neighbor search. **IEEE Trans. Pattern Anal. Mach. Intell.**, v. 33, n. 1, p. 117–128, 2011. Disponível em: <<https://doi.org/10.1109/TPAMI.2010.57>>.

KORN, F.; PAGEL, B.; FALOUTSOS, C. On the 'dimensionality curse' and the 'self-similarity blessing'. **IEEE Trans. Knowl. Data Eng.**, v. 13, n. 1, p. 96–111, 2001. doi:10.1109/69.908983. Disponível em: <<https://doi.org/10.1109/69.908983>>.

MANBER, U.; MYERS, E. W. Suffix arrays: A new method for on-line string searches. **SIAM J. Comput.**, v. 22, n. 5, p. 935–948, 1993. doi:10.1137/0222058.

MOHAMED, H.; MARCHAND-MAILLET, S. Metric suffix array for large-scale similarity search. In: **ACM WSDM 2013 Workshop on Large Scale and Distributed Systems for Information Retrieval, Rome, IT**. [S.l.: s.n.], 2013. p. 1–6.

MOHAMED, H.; MARCHAND-MAILLET, S. Permutation-based pruning for approximate K-NN search. In: **International Conference on Database and Expert Systems Applications (DEXA)**. Prague, Czech Republic: Springer, 2013. (Lecture Notes in Computer Science, v. 8055), p. 40–47. Doi:10.1007/978-3-642-40285-2_6.

NAVARRO, G. **Compact Data Structures - A Practical Approach**. [S.l.]: Cambridge University Press, 2016. ISBN 978-1-10-715238-0.

ROSA, F. R.; LOUZA, F. A.; RAZENTE, H. L. Vetor de sufixos métrico compacto. In: **Proceedings of the 38th Brazilian Symposium on Databases, SBDD 2023, Belo Horizonte, MG, Brazil, September 25-29, 2023**. SBC, 2023. p. 414–419. Disponível em: <<https://doi.org/10.5753/sbdd.2023.233440>>.

ROUSSOPOULOS, N.; KELLEY, S.; VINCENT, F. Nearest neighbor queries. In: **International Conference on Management of Data (SIGMOD)**. San Jose, CA: [s.n.], 1995. p. 71–79. doi:10.1145/223784.223794.

SAMET, H. **Foundations of Multidimensional and Metric Data Structures**. San Francisco: Morgan Kaufmann, 2006.

VADICAMO, L.; AMATO, G.; GENNARO, C. Induced permutations for approximate metric search. **Information Systems**, Elsevier BV, v. 119, p. 102286, out. 2023. ISSN 0306-4379. Disponível em: <<http://dx.doi.org/10.1016/j.is.2023.102286>>.

YIANILOS, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In: RAMACHANDRAN, V. (Ed.). **ACM/SIGACT-SIAM Symposium on Discrete Algorithms**. Austin, Texas: ACM/SIAM, 1993. p. 311–321. <http://dl.acm.org/citation.cfm?id=313559.313789>. Disponível em: <<http://dl.acm.org/citation.cfm?id=313559.313789>>.

ZEZULA, P.; AMATO, G.; DOHNAL, V.; BATKO, M. **Similarity search**. 2006. ed. New York, NY: Springer, 2005. (Advances in Database Systems).