ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Keyword-based multimedia data lookup in decentralized systems

Relatore:
Chiar.mo Prof.
STEFANO FERRETTI

Presentata da:
EMANUELE FAZZINI

Co-Relatore:
Chiar.mo Dott.
MIRKO ZICHICHI

II Sessione
Anno Accademico 2021/2022

Sommario

I sistemi decentralizzati hanno permesso agli utenti di condividere informazioni senza la presenza di un intermediario centralizzato che possiede la sovranità sui dati scambiati, rischi di sicurezza e la possibilità di colli di bottiglia. Tuttavia, sono rari i sistemi pratici per il recupero delle informazioni salvate su di essi che non includano una componente centralizzata. In questo lavoro di tesi viene presentato lo sviluppo di un'applicazione il cui scopo è quello di consentire agli utenti di caricare immagini in un'architettura totalmente decentralizzata, grazie ai Decentralized File Storage e alla successiva ricerca e recupero di tali oggetti attraverso una Distributed Hash Table (DHT) in cui sono memorizzati i necessari Content IDentifiers (CID).

L'obiettivo principale è stato quello di trovare una migliore allocazione delle immagini all'interno del DHT attraverso l'uso dell'International Standard Content Code (ISCC), ovvero uno standard ISO che, attraverso funzioni hash content-driven, locality-sensitive e similarity-preserving, assegna i CID IPFS delle immagini ai nodi del DHT in modo efficiente, per ridurre il più possibile i salti tra i nodi e recuperare immagini coerenti con la query eseguita. Verranno, poi, analizzati i risultati ottenuti dall'allocazione dei CID delle immagini nei nodi mettendo a confronto ISCC e hash crittografico SHA-256, per verificare se ISCC rappresenti meglio la somiglianza tra le immagini allocando le immagini simili in nodi vicini tra loro.

Abstract

Decentralized systems have allowed users to share information without the presence of a centralized intermediary who possesses sovereignty over the data exchanged, security risks, and the possibility of bottlenecks. However, practical systems for retrieving the information saved on them that do not include a centralized component are rare. In this thesis work, the development of an application is introduced whose purpose is to allow users to upload images to a totally decentralized architecture, thanks to Decentralized File Storage and the subsequent search and retrieval of such objects through a Distributed Hash Table (DHT) in which the necessary Content IDentifiers (CID) are stored.

The main aim was to find a better allocation of images within the DHT through the use of the International Standard Content Code (ISCC), i.e., an ISO standard that, through content-driven, locality-sensitive, and similarity-preserving hash functions, assigns the IPFS CIDs of the images to the nodes of the DHT in an efficient way, to reduce as much as possible the hops between nodes and retrieve images consistent with the query executed. The results obtained from allocating image CIDs in nodes by comparing ISCC and SHA-256 cryptographic hash will then be analyzed to see whether ISCC better represents the similarity between images than the hash by allocating similar images in nodes close to each other.

Contents

List of Figures

Chapter 1

Introduction

The digital transformation we are witnessing leads to a large amount of data that needs to be stored and managed as a crucial factor for economies and societies worldwide. One example is cloud platforms that allow users and companies to save their data and access it at any time if they have an internet connection.

Peer-to-peer systems are increasingly gaining a foothold in technologies used daily; think of the growing interest that cryptocurrencies, based on Distributed Ledger Technologies (DLTs), have generated within global communities. The use of decentralized systems, however, is not limited to cryptocurrencies. Such systems can also be used for different purposes, such as file sharing.

This interest in distributed software is most likely because of all the benefits they bring in terms of security, eliminating the single point of failure, censorship by those who hold the centralized servers, data reliability, absence of bottleneck, and the possibility given to the users of taking part of the network contributing with their capacities.

Based on these assumptions, the purpose of this thesis work was to present a fully decentralized architecture that would give users the ability to save their images into a Decentralized File Storage, in this case, IPFS, by being able to go out and

perform queries so that they could search for those images on a hypercube-shaped Distributed Hash Table that contains references to the objects stored with IPFS.

In order to make sure that better search performance was achieved on the research, comparison tests were carried out to choose which methodology gave optimal results in allocating images between the International Standard Content Code (ISCC) and the SHA-256 cryptographic hash to see which one performed better in allocating media objects within the DHT in terms of the minimum distance between similar images, and maximum distance between different images.

This thesis is structured into six chapters: the second chapter will present a background to the technologies used along with an introduction to some topic-related works addressed by researchers and not; the third chapter will show the architecture adopted for the system; the fourth chapter will explain the design and methodologies adopted in devising the compare tests and the query performing method; the fifth chapter will describe the implementation of the tests and queries introduced in the third chapter; the sixth chapter will compare the results obtained from the tests; and, in the end, the seventh chapter will express conclusion observations.

Chapter 2

State of the Art

This chapter is structured into two main sections. The first will introduce a background of all the technologies used to accomplish the final results, and the second will present some topic-related works of this thesis.

2.1 Background

2.1.1 Decentralized File Storage

This architecture is an alternative to classical client-server architectures, through which the user, to access a resource on the Internet, provides a URL representing the destination where the desired resource is located. The problem with client-server architectures is that being centralized, they are vulnerable to cyber attacks, data loss, and exploitation.

Here, Decentralized File Storage comes into play in which contents are not reachable based on where they are but based on `what they are`. Through InterPlanetay File System (IPFS), which is a DFS and a protocol thought for decentralized architectures [6], it is possible to generate a unique key, called Content ID (CID)

which is a digest produced from a hash function applied to the file, and users can retrieve the desired objects from the peer-to-peer network. The problem with this architecture is that it is needed the possession of the CID to get a file saved in IPFS.

So there is a need for an additional layer that saves CIDs in a structure that allows search for objects through queries. Such a layer is implemented through a Distributed Hash Table.

2.1.2 Distributed Hash Table

Distributed Hash Tables are decentralized systems that offer the possibility of exploiting the Hash Table data structure in a distributed manner, thus efficiently allocating objects to keywords. In this case, keywords represent binary identifiers by which nodes are recognized, and objects are the values contained in them.

Nodes, in addition to representing keys to which objects are assigned, and thus saved, also contain a partial view of the network so that routing mechanisms can be implemented to go from a starting node to the desired node.

The association between content and keywords, i.e., nodes, is computed using a hash function that maps the content into a keyword consisting of a sequence of n bits.

Underlying this architecture is the idea of distributing the workload over multiple nodes based on the IDs that identify them. The search for an object *i* will then become the search for the node within the DHT that manages the subset of the space of IDs containing *i*.

This type of infrastructure has been used for the implementation of many decentralized services, such as BitTorrent [7], DFS [8], and Content-Addressable Networks (CANs) [9].

2.1.3 International Standard Content Code

The International Standard Content Code (ISCC) [1] is an ISO-approved standard [10] that, given an input file, gives the ability to create a corresponding code that goes to identify the file itself.

As written in the project's vision, such standard was created to identify media files within decentralized structures, such as blockchains.

The ISCC code of a file is generated from data contained in the file itself, such as the metadata, the contents, and the raw data of the file. This mechanism allows for not having to assign a code or incorporate it into it manually; the file is the resource for generating its unique identification code based on the data from which it is formed.

Generating the code is a series of content-driven, locality-sensitive, and similarity-preserving hash functions that, unlike cryptographic hash functions, go into preserving similarity between data so that two similar contents do not have totally different codes.

ISCC Features

ISCC consists of four main components:

- Metadata Similarity, Meta-Code;
- Content Similarity, Content-Code;
- Data Similarity, Data-Code;
- Data Integrity, Instance-Code.

These components can be considered separately, all together, or used to generate a code represented by a digest derived from the four components.

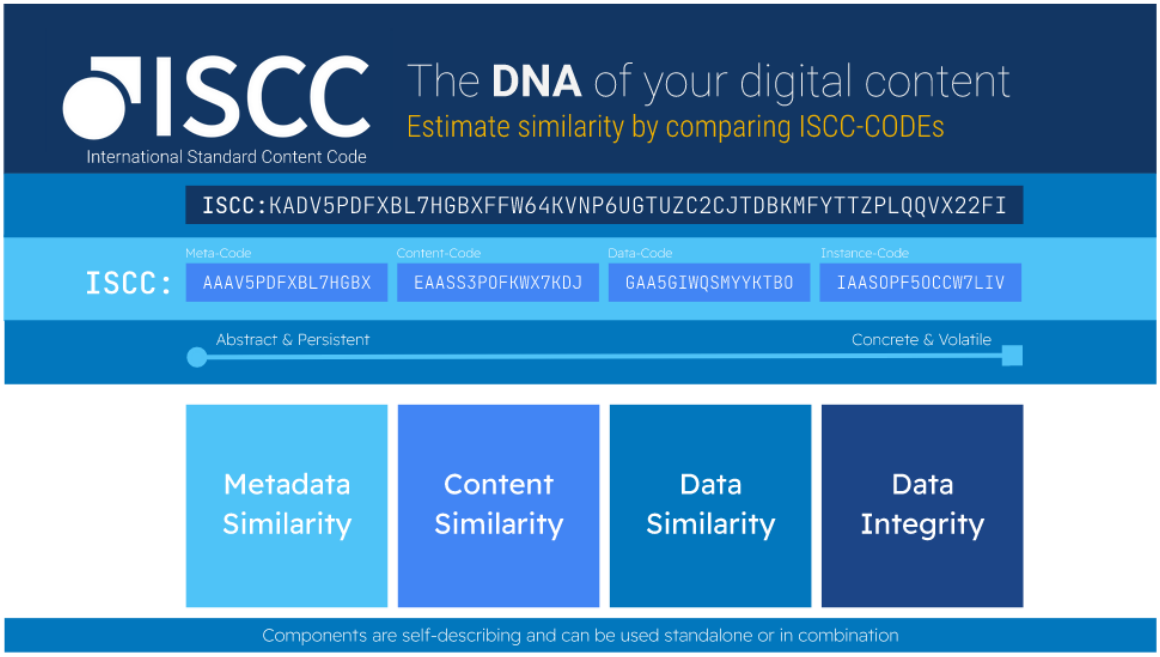

Figure 2.1: How ISCC is composed [1]

In figure 2.1, an example of ISCC code can be seen.

The ISCC code is a digest derived from the four separately computed codes. It consists of 52 characters, totaling 36 bytes (288 bits). An overview of how an ISCC code is constructed based on the data from which the file consists is shown in figure 2.2.

Each separate component consists of 72 bits, 8 bits for the header, and 64 bits for the body and has as its return value a string encoded in base58-iscc [11]. The header is intended to recognize the type of code component and, in the case of Content Similarity Code, to indicate the file type from four main choices: text, audio, video, and image. Figure 2.3 explains how the header bits are assigned for each component based on the component itself and the file type considered for the Content Similarity Code.

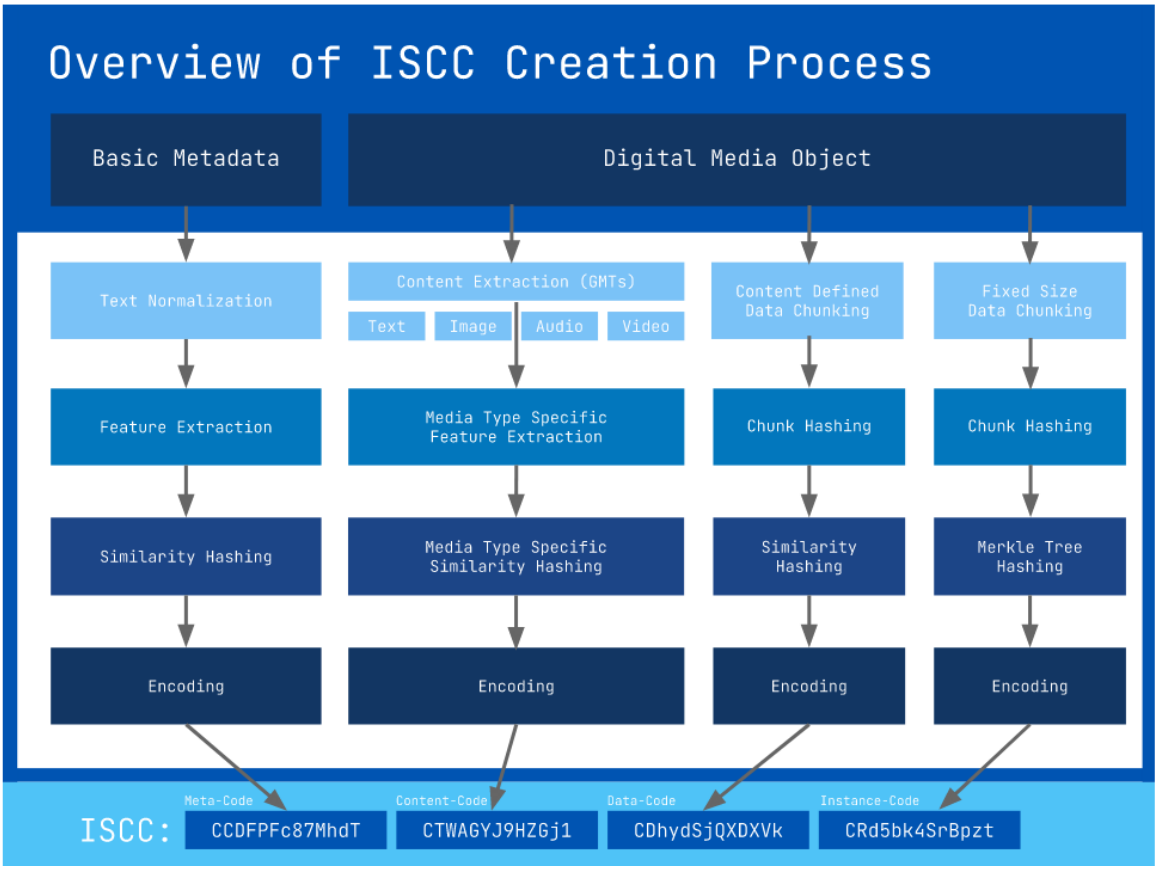

Figure 2.2: Overview of how an ISCC code is built [2]

Meta-Code

This code is generated from the file metadata passed as input to the ISCC algorithm. It takes advantage of the metadata in the file, and the ISCC-SDK [12] also gives the ability to add metadata, such as the creator's name, a description of the file, and many other contents.

The algorithm for generating the Metadata Similarity Code takes two elements as input:

- title: the file title;
- extra: a file description that gives the option of generating a unique code,

Component	Nibble-1	Nibble-2	Byte	Code
Meta-Code	0000	0000 - Reserved	0x00	CC
Content-Code-Text	0001	0000 - Content Type Text	0x10	CT
Content-Code-Text PCF	0001	0001 - Content Type Text + PCF	0x11	Ct
Content-Code-Image	0001	0010 - Content Type Image	0x12	CY
Content-Code-Image PCF	0001	0011 - Content Type Image + PCF	0x13	Ci
Content-Code-Audio	0001	0100 - Content Type Audio	0x14	CA
Content-Code-Audio PCF	0001	0101 - Content Type Audio + PCF	0x15	Ca
Content-Code-Video	0001	0110 - Content Type Video	0x16	CV
Content-Code-Video PCF	0001	0111 - Content Type Video + PCF	0x17	Cv
Content-Code-Mixed	0001	1000 - Content Type Mixed	0x18	CM
Content-Code Mixed PCF	0001	1001 - Content Type Mixed + PCF	0x19	Cm
Data-Code	0010	0000 - Reserved	0x20	CD
Instance-Code	0011	0000 - Reserved	0x30	CR

Figure 2.3: Assignment of bits in the header [3]

empty by default. In addition, other metadata can be added as Data-URL.

After a series of steps shown in this link [13], the code is generated, and the hash function to compute the digest on which base58-iscc encoding will then be applied is xxHash64 [14].

Content-Code

The second part of the header represents the file type (text, image, video, audio), as shown in figure 2.3. The generation of this code depends on the type of file being considered, and the various steps for generating the code based on the file type are shown in this link [15].

This code is necessary when, for example, the content of two files is the same, but the format is different. This specific case can be seen in image 2.4.

Figure 2.4: Content-Code features [4]

PCF stands for Partial Content Flagging and represents the last bit of the header for Content-Code. This bit means that the entire file should be considered for code generation if the bit is marked with 0 or only a specified part indicated by the user if the bit is marked with 1.

Data-Code

The Data-Code represents the code that encodes the similarity of the data contained in the file, taking advantage of the raw data without having to figure out what type of file it is. There is no need to interpret the type to determine what actions to take based on it, as with the Content-Code. This approach allows working on any file without having to process it first for the algorithm to work.

The raw data in the file is divided into chunks on which a hash will be computed. After that, a sample will be taken on which the MinHash function will be applied, and the base58-iscc encoded value will be returned as the Data-Code of the file.

The steps performed by this algorithm can be observed in this link [16].

Instance-Code

Instance-Code is used to prove the integrity of the work passed as input to the algorithm. Raw file data are split into 64 KB chunks, after which a Merkle tree will be constructed and the hash contained in the root node encoded in base58-iscc as the Instance-Code will be taken.

The steps for generating this code can be found at this link [17].

2.2 Related works

The growing interest in distributed systems leads academics and developers to search for new technologies that would give the ability to exploit decentralization for both data storage and data management.

The Graph is a platform created to provide a Decentralized Query Protocol [18], i.e., an architecture built on top of Ethereum and IPFS, which allows users to perform queries on data stored on these two technologies. Their peer-to-peer network leverages a Service Addressable Network to locate nodes that can fulfill the query. The organization of the network is similar to a Decentralized Autonomous Organization (DAO). However, the method used to store indexes does not use a DHT.

Concerning IPFS, on the other hand, to make sure that it can overcome the limitations regarding file searching due to its storage mechanism, a search engine called *ipfs-search* [19] was implemented. The problem with this solution is that it is not

based on a decentralized architecture. To make sure that such a solution was decentralized, Siva [20] was proposed, which leverages an inverted index to search for content saved on IPFS but concretely adds no improvement in terms of keyword storage optimization.

A more inherent work to the topics discussed in this paper is found in [21], in which the authors implemented a Road Hazard Detection search system for self-driving vehicles. The architecture is based on IOTA DLT to save the locations of possible road hazards and search for those locations using a hypercube-shaped DHT in which the ID of the transitions containing the information are stored. In [5], the focus is more on organizing a Decentralized Autonomous Organization for incentives users via a reward-based mechanism to keep alive their nodes, but with the same architecture as in the previously mentioned article using a DFS instead of a DLT to store the contents referenced by the CIDs saved in the DHT nodes. A similar approach to the topic addressed by this thesis can be found in [22]. The researchers developed a DHT called Hamming DHT, based on Chord architecture [23], that stores contents based on their cosine similarity, given that contents are represented as vectors, to reduce the hops between nodes when a query is performed.

Chapter 3

Architecture

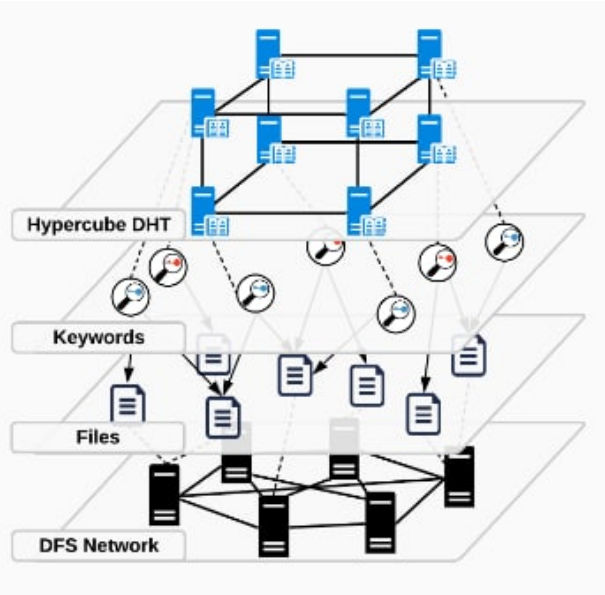

Figure 3.1: Project architecture [5]

The architecture on which the project was built is based on two main layers. The first, which constitutes the base layer, consists of a Distributed File Storage, in this case, IPFS, which is responsible for saving and sharing images, while the second consists of a hypercube-shaped Distributed Hash Table for storing the CIDs

associated with the images.

These two technologies were chosen in order to make the architecture completely decentralized. This way, avoiding all the problems related to the centralized system described before is possible.

In figure 3.1, taken from the article [5], it is possible to see the system architecture in which IPFS implements the DFS network as the first layer, where files are stored and shared between nodes, and the keyword associated to the files are the IDs of the DHT peers, in the second layer.

Each keyword that identifies a specific node within the DHT is composed of r bits, so, having the form of a hypercube, the network will always have the number of logical nodes equal to 2^r. However, there may be a different number of physical nodes than logical nodes; for example, if each logical is implemented from two physical nodes or vice versa, two logical nodes refer to only one physical node.

Regarding the routing mechanism, the term hop is introduced. This word represents the steps from one peer to another to reach the desired resource. Finding ourselves in hypercube-shaped DHT hops represents the Hamming distance between the bit-string of the starting node to the bit-string of the finishing node since each node has as neighbors those who differ from its ID by one bit. If the shortest path between two nodes is calculated, the number of hops represents the Hamming distance between their IDs.

Since the purpose of this research was to look for a more efficient way of mapping IPFS CIDs within a Distributed Hash Table to make their discovery more efficient, the hypercube form was chosen because of its feature of reducing the number of hops between slightly different nodes, thus allocating similar multimedia files within nodes that are as close as possible to each other.

Chapter 4

Methodology

This chapter will show how tests were performed to choose which candidate between ISCC and cryptographic hash is better in allocating images into the DHT nodes. The aim is that similar images would remain close together while different images stay far apart.

4.1 Images dataset

The test dataset chosen to see if ISCC is a more appropriate choice for allocating multimedia files into a hypercube-shaped DHT is composed of 30 folders, that from now on, will be called image classes, each containing 15 photos of the same subject, like a monument or a painting. Table 4.1 are reported all the image classes used as datasets and the relative queries executed on the search engine.

These images were downloaded from *bing.com* through a Python script where a dictionary stores 30 URLs representing the image search for every subject. The script iterates over the URLs and, for each one, create a folder named as the key that store the URL and downloads the first 15 images resulting from the search.

Table 4.1: Image Classes Names & Bing Query

Image Class	Bing Query
altare_della_patria	altare della patria
arco_pace	arco della pace milano
arco_trionfo	arco di trionfo
arena_verona	arena di verona
basilica_santa_croce	basilica di santa croce firenze
basilica_san_marco	basilica san marco
cinque_terre	cinque terre
colosseo	colosseo
due_torri	due torri bologna
duomo_milano	duomo di milano
fontana_trevi	fontana di trevi
hamburger	hamburger
il_bacio	il bacio klimt
lavorare_lavorare_lavorare	lavorare lavorare lavorare preferisco il rumore del mare
monalisa	monalisa
nettuno	nettuno bologna
notte_stellata_vangog	notte stellata vangog
obelisco_wash_dc	monumento a washington
pantheon	pantheon
piazza_san_marco	piazza san marco
pietà_mic	pietà michelangelo
ponte_vecchio	ponte vecchio
reggia_caserta	reggia di caserta
san_luca	san luca bologna

san_petronio	san petronio bologna
statua_libertà	statua della libertà
taj_mahal	taj mahal
torre_pisa	torre di pisa
ultima_cena	ultima cena leonardo
urlo_munch	urlo di munch

4.2 Images ID Generation

The reason ISCC was chosen to test if it is possible to obtain a more efficient allocation of the images on the hypercube DHT compared to cryptographic hash is to ensure that the hops required to reach the desired objects are as few as possible.

Cryptographic hashing was chosen as the comparison method to observe whether ISCC would perform better than a solution that generates node IDs in which to save image references more randomly.

Two main techniques were adopted regarding the generation of IDs for each image. The first concerns the generation of IDs, for both hash and ISCC, by going to choose an r, which represents the number of bits with which a node ID is composed, and a g, that represents the number of characters from which the chunks into which the code, hash, and ISCC, will be divided will be formed, which will then be used to go to calculate the position of the bits to turn on within the bit-string, initially set with all 0.

The second is similar, and instead of generating a single bit-string on which to go to compute the bit positions to be turned on, many will be created, 16 to be precise, one for each character of an ISCC code, of 16 bits each, which will later be

concatenated with each other to go to form a 256 bit-string representing the node ID in which to save the image reference.

4.2.1 Non-concatenated bit-string generation

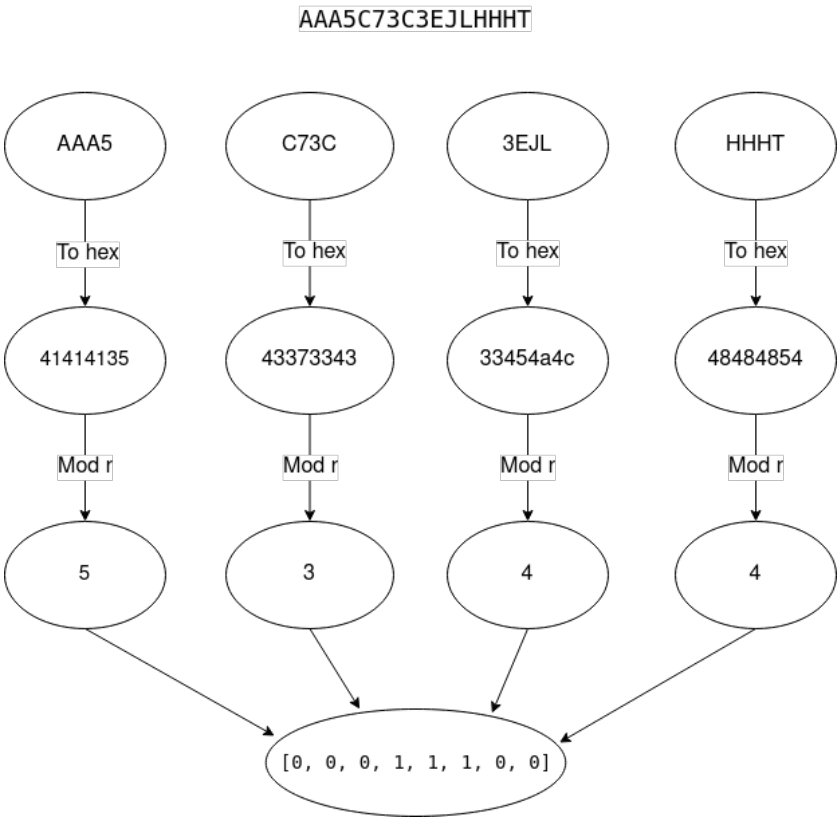

Figure 4.1: Example of ID generation from a Meta-Code

In this section, we will go into a deeper analysis of how the tests were designed about generation and comparison for the first technique, that is, the one that does not use concatenation between multiple bit-strings for the generation of the final node ID.

Regarding ISCC code, only Meta-Code and Content-Code were used, first taken individually to see how they behave in ID assignment. Then they were used to-

gether, calculating the OR between the bit-strings generated by both to obtain a final bit-string. Instead, the SHA256 function of the images was computed for the hash.

The method for generating the bit-string corresponding to the ID of a node within the hypercube will be explained below.

As was briefly introduced earlier, two main parameters will be passed:

- r: represents the number of bits that identify a node within the DHT.

- g: represents the number of characters in each chunk into which the code generated from the image will be divided. Then, from each chunk will be computed, the corresponding hexadecimal value which module with r will return the position within the bit-string, initially set with all 0, in which to go to insert 1.

In figure 4.1, it is possible to see a flowchart of how an ID is generated from the Meta-Code that it is given as input; in this case, r was set to 8 and g to 4.

For Content-Code, the same procedure is applied. The algorithm divides chunks of the generated code, transforms them into hexadecimal values, and computes the hex module r to find the bit-string position that must be set to 1.

Once both IDs generated from a single image are obtained, the OR between the Meta-Code's bit-string and the Content-Code's bit-string will be computed to obtain an additional ID representing the node chosen by the union of the two codes.

An example of how the final bit-string between Meta-Code and Content-Code is obtained can be seen below:

$$Meta - Code : AAA5C73C3GZDHDHD \rightarrow [0,0,0,1,1,1,0,0]$$

$$OR$$

$$Content - Code : EEA2T2CQVF6ZR7BU \rightarrow [0,1,1,0,0,1,0,0]$$

$$\downarrow$$

$$[0,1,1,1,1,1,0,0]$$

Given that hash generated code has four times the character that ISCC contains, when we selected a specific g for ISCC, the g that was assigned to the algorithm for generating the bit-string associated with the hash code is four times bigger than the g assigned to the ISCC node generation ID.

The selected r and g are:

- r: 8, 16, 32, 64, 128, 256, 512, 1024; for both ISCC and hash;
- g: 2 for ISCC and 8 for cryptographic hash, 4 for ISCC and 16 for cryptographic hash.

Running the tests in this way, we will have for each r two subsets: the first performs the generation of the node IDs to which to assign images considering the first two g, and the second considers the other two g.

Each run will save the results in a dataset contained in folders within a path representing the r, and g used to generate the IDs of those specific images. For example, the following path will contain all the CSVs representing the image ID generation algorithm for ISCC having $r = 8$ and $g = 2$ as parameters.: `./id_or_generated_meta/results/iscc-2_hash-8/csvs/r-8/g-2`.

These datasets, a total of 30 for each combination of r and g, one for each class of images, contain information about the Meta-Codes and Content-Codes generated by the images belonging to that specific class, with the corresponding node IDs

calculated based on the algorithm explained above, will then be used to perform the efficiency tests of ID generation by ISCC and hashes that will be explained later. A dataset for ISCC has the following column:

- Img-Name: the name of the image in the class;
- Meta: the Meta-Code generated from the image;
- R-Meta: the corresponding r-bit string generated from the Meta-Code;
- Content: the Content-Code generated from the image;
- R-Content: the corresponding r-bit string generated from the Content-Code;
- Final-R: the r-bit string obtained by calculating the OR operator between R-Meta and R-Content.

Instead, a dataset for the IDs generated by the hash has the following columns:

- Img-Name: as above, the name of the image of the class;
- SHA256: the output of a SHA256 function of the image;
- Final-R: the r-bit string obtained from the SHA256.

4.2.2 Concatenated bit-string generation

Regarding the generation of IDs through concatenating the bit-strings, turning on the bits within the r-bit string remains almost identical to that without concatenation.

This technique differs from the one explained before because multiple r's were not tested to see which performed better. A prefixed r of 16 was chosen for each bit-string to ensure that, having ISCC 16 characters, each character could fall back on a

different bit. This process would enable a similarity to be as close to the generated code within the ID as possible. Also, no multiple g's were tested since we were interested in taking one character at a time, from which we then went to see which bit the algorithm turned on.

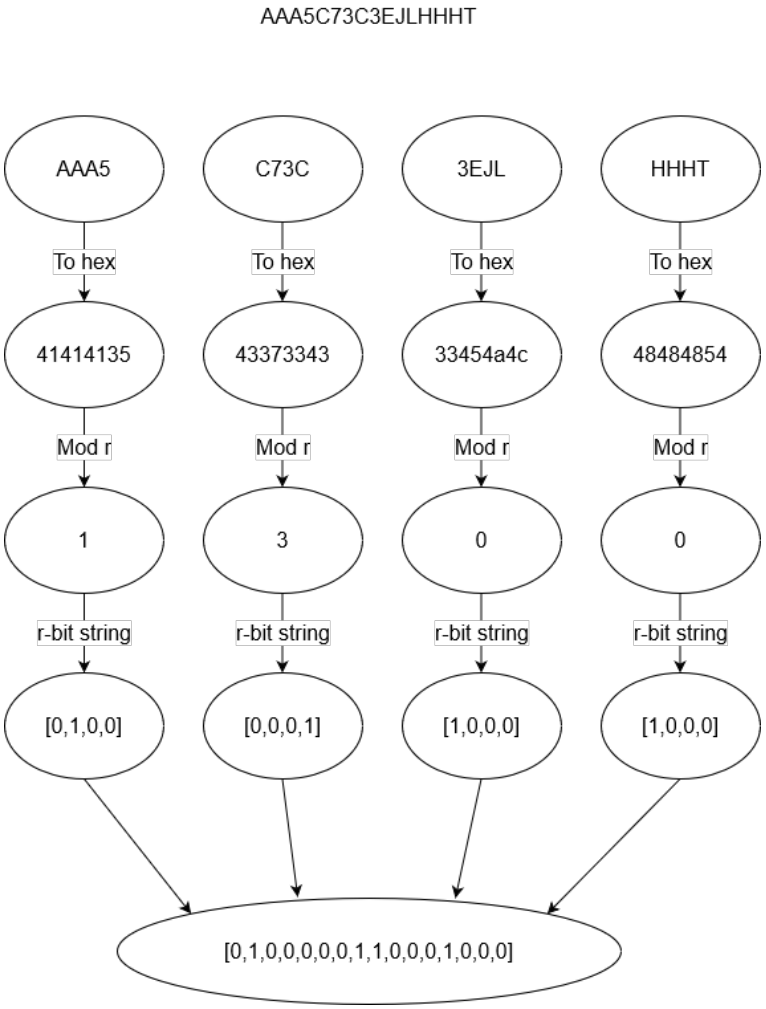

Figure 4.2: Example of concatenated bit-string generation from Meta-Code

So, going into more detail, r was set to 16 bits and g to 1 for ISCC, while for the hash, g was set to 4 to get 16 chunks.

Then, after setting these two parameters, the process is the same. Take the gen-

erated code from the image, ISCC, and hash; divide into chunks formed by g characters; for every chunk, calculate its consideration hexadecimal value and, in the end, calculate the module between the hexadecimal value and r, that is 16.

Once this procedure has been performed for each character of ISCC or every four characters of the hash, all the individual 16-bit bit-strings will be concatenated into one to form a 256-bit-string with 16 bits turned on. In the diagram 4.2, it is possible to see this mechanism for generating the bit-string starting from a Meta-Code having r and g set to 4.

As with the method that does not use concatenation, the Meta-Code and Content-Code were evaluated individually and joined to form a single node identifier using the `OR` operator between Meta-Code and Content-Code bit-strings, as in the example below.

$$Meta - Code : AAA5C73C3EJLHHHT \rightarrow [0,1,0,0,0,0,0,1,1,0,0,0,1,0,0,0]$$

$$OR$$

$$Content - Code : EEA2T2CQVF6ZR7BU \rightarrow [0,0,1,0,0,1,0,0,1,0,0,0,0,1,0,0]$$

$$\downarrow$$

$$[0,1,1,0,0,1,0,1,1,0,0,0,1,1,0,0]$$

Also, for this technique, all information computed by the algorithm collated to the images are saved to datasets placed in paths formed by folders representing the methodology used, for example, `id_concat_generated_final/results/iscc/csvs`.

As with the previous technique, a dataset was created for each class of images, containing information about the codes generated by the algorithm using bit-string concatenation. The structure of the datasets is the same as the non-concatenated method.

4.3 Internal Hamming distance

Once all the datasets diversified by the ID generation method and the r and g parameters have been obtained, we can proceed with test generation to see who performs better in allocating references to images within a hypercube-shaped Distributed Hash Table.

The first test chosen is to check which ISCC and cryptographic hash performs better in how far apart, on average, the images within the same class are.

The hamming distance between the bit-strings of the images in each dataset was calculated to measure the distance between the IDs of images belonging to the same class to figure out how many hops, on average, are needed to reach the images of a class starting from a node that contains one or more of them.

The results of the distances will be saved in a CSV representing a distance matrix between the classes; thus, the rows and columns of the matrix will be the same, that is, the names of the images belonging to that specific class. Once a matrix of distances is obtained, an average of the lower diagonal will be calculated to assign that specific class a mean distance separating images from each other within the hypercube. The standard deviation was also reported to verify that outliers do not affect the mean.

Once all the distance matrices have been created for each class whose IDs have been generated by a specific method and parameters, for example, using only Content-Code and having r=16 and g=2 for ISCC as parameters and g=8 for the hash, two JSON files will be created, one for ISCC and one for cryptographic hash, in which the average hamming distances for each class will be written with their standard deviations.

Through these averages, we can compare the generation of IDs from the two codes based on the magnitude of the hamming distance, i.e., the lower the average

internal distance, the better the allocation of images within the DHT, again taking into account the standard deviation.

4.4 Distance between image classes

However, looking at what technique performs best when placing references to images within the same class is not enough. To understand which methodology is more efficient, one must also look at how the classes behave with each other, that is, whether, within the hypercube, different classes are allocated at nodes far apart.

The method chosen to calculate the distance between classes is to assign each class a centroid and then measure the hamming distance between them. The centroids were calculated as the median node ID among the nodes to which images of a class were assigned. From bit-strings, the IDs were transformed into numbers, these numbers were inserted within an array sorted in ascending order, and the node at the center of the array, i.e., the median, was chosen as the centroid of the class.

Then, for each method combination used for ID and parameter generation, a nested JSON is created, where the nested represents the parameter combination, where its centroid and the hamming distance between that centroid and the centroids of the other 29 classes are saved for each image class.

These measurements will be used for the final part of the tests in which we are going to combine into a single metric the average internal distance from the distance to the other classes to check which combination performs best by going to see which maximizes the distance to the other classes while minimizing the distance between the node IDs assigned to the images of the same class.

4.5 Efficiency metric

The last measure used to test which combination of ISCC and hash performed best was a metric designed to test which set of parameters and id generation technique went to maximize the distance between classes while minimizing the internal distance between images belonging to the same class.

In a nutshell, an attempt was made to combine the two sections explained above into a single metric that would allow evaluating the set of factors and not the factors individually.

The main difference in devising this metric is that concerning internal distance, the average of distances between classes is not taken, but the distance between it and the centroid is calculated for each image belonging to the class; of course, not taking the image representing the centroid.
The pseudocode that leads to obtaining the metric is given in the algorithm 1.

Explained in words, the metric works like this: first, an empty array called *results* is instantiated, after which a for loop is initialized that cycles over all the image classes and two arrays are created for each class: the first containing all the hamming distances between the centroid of class c and the centroids of the other classes, and the second to save all the hamming distances between the centroid class c and the images belonging to c.

After both arrays have been created, the average of each is calculated, and the difference between them is added to the results array.

Finally, the metric is the result obtained by averaging the array results. Of course, the higher this result is, the more it stands to signify that the methodology used to generate the node id and the configuration of the r and g parameters is best.

The metrics are calculated for the same combinations of the parameters for both

Algorithm 1 Efficiency metric

$results\_array \leftarrow []$
for each class c **do**
  $distance\_between\_classes \leftarrow []$
  for each class j != c **do**
    $distance\_between\_classes.append(hamming\_distance(centroid(c), centroid(j)))$
  end for
  $internal\_distance \leftarrow []$
  for each image k != centroid(c) **do**
    $internal\_distance.append(hamming\_distance(centroid(c), k))$
  end for
  $results\_array.append(avg(distance\_between\_classes) - avg(internal\_distance))$
end for
return $abs(avg(results\_array))$

the node generation that occurred through ISCC and the cryptographic hash, after which subtraction is performed between the metrics obtained from ISCC and the metrics obtained from the hash in absolute value, and the more significant this metric is, the better ISCC proved to be for that parameter configuration; the smaller it is, the worse it proved to be compared to the hash.

The results of the metrics for generating the IDs will then be saved in a nested JSON, where the nested represents the combination of methodology and parameters used to achieve that metric, and within the object will also be the value of the comparison between ISCC and hash. Finally, another JSON will be created into which the maximum and minimum results between the comparisons will be entered so that we can then go and see whether the minimum is greater than the

maximum[1], so the hash performed better than ISCC, or whether the maximum is greater than the minimum[2], which stands for ISCC's more remarkable ability to represent images within the hypercube.

4.6 Query testing

After sifting through all the types of tests and seeing which combination performs best, a testing system will be structured in such a way that an image not belonging to any of the classes on which the tests were performed but depicts the same subject as one of them, will be chosen to generate an ID from which perform a search on the hypercube and show which images will be returned from that search.

The following were chosen as *r-values* for realizing DHT: 8, 12, 16. In this way, an attempt was made to compare how different sizes of the hypercube representing an actual situation behaved, considering that with r equal to 16, there are more than 60 thousand logic nodes.

As for g, however, it will be shown in the results phase since it was chosen based on performance. Of course, the technique that does not involve concatenation was chosen since an r equal to 256 bits has too many nodes.

To make such a simulation as truthful as possible, a JSON called `hypercube.json` will be created, within which nodes will be represented as keys of pairs, and values will be lists of CIDs, i.e., hash references to images saved on IPFS.

To implement the IPFS node through which the images were added to the network, an IPFS container on Docker was used, to which ports were opened so that it could be reached and images could be added to best simulate an actual architecture.

The implementation of the hypercube and the search using the methods that we

[1]Looking at the value as absolute value

[2]Same as the previous note

are going to describe in a moment was accomplished for both the generation of the node IDs to be associated with the images via ISCC and hash so that we can go and compare which objects result from the search, and how many of them match the images we are looking for along with the hops performed to visit the nodes that returned us the objects.

Two types of research will be carried out and are inspired by the article [21]:

- Pin-search: the mechanism of this search method is identical to the previously mentioned article. Starting from a node, which in this case is the node with all bits set to 0, we want to reach the node generated by the image, which can contain CIDs of similar images. The shortest path is calculated, and objects associated with that node are returned together with the hops to reach it;

- Superset-search: while for the previous method, the mechanism is the same as the article from which it was inspired, it is slightly different for this algorithm. In the article, a Depth First Search is applied to go for objects that can be described by keyword sets that include K, where K represents the ID of the node generated by the image with which the query is being accomplished. Instead, the implementation here involves a Breadth First Search, going to check all the neighbors of the node generated by the query, then the neighbors of the neighbors, and so on. This choice was made since it was noted that the node IDs assigned to the images belonging to the same class might not belong to the same set of keys.

Finally, the results of the three case histories will be added into a JSON for evaluation that will take place in the Results chapter.

Chapter 5

Implementation

The implementation of the tests explained earlier in the Methodology chapter was structured as follows. Six folders were created, three with the prefix `id_or_generated` and three with the prefix `id_concat_generated`.

Folders with the prefix `id_or_generated` go to represent the scripts and their results using the generation of the ID of the nodes to which the images are to be assigned without concatenating the bit-strings generated by the individual characters, as explained earlier. In contrast, folders with the prefix `id_concat_generated` contain the files, scripts, and results of the methodology using concatenation between bit-strings.

For each of the two types of generation, there are three folders since the Meta-Code was tested individually so that the prefix will end with `meta`, the Content-Code individually, so the prefix will end with `content`, and finally, the last folder representing the conjunction of the two, that terminates with the word `final`, reached by calculating the `OR` operator between the two.

The programming language chosen for writing the code with which to go about running the tests is Python. The libraries used to achieve the results are varied. The `iscc_sdk` library [12], as presented in the methodology chapter in the ISCC

section, is a library implemented by the ISCC Foundation to enable the generation and management of ISCC codes.

Other main libraries used are:

- `pandas`: for the management and creation of CSV files used to save the node IDs assigned to images and the matrices of hamming distances between images within classes;
- `ipfshttpclient`: for connecting to the IPFS node instantiated with Docker and managing objects, such as: adding and requesting images;
- `openlocationcode`: in such a way as to transform geographic coordinates into the Open Location Code format [24], to identify an area and not a precise point at which the image was taken;
- `networkx`: through the use of this library, a hypercube-shaped graph was created and used based on the number of nodes chosen for query testing in order to count the number of hops needed;
- `matplotlib`: for displaying measurements and other graph-like elements.

Some libraries already present in Python were also used, for example, `os` for using methods with which to access the files needed by the algorithms and `json` for creating and managing JSON files.

All the code and results are available on this link [25]

5.1 Non-concatenated bit-string generation

As introduced at the beginning of the chapter, the tests with their respective results are located within the following three folders:

- `id_or_generated_meta`;
- `id_or_generated_content`;
- `id_or_generated_final`;

Each folder has two main folders inside:

- script: contains all Python scripts, respectively divided into two folders, one for the scripts that generate the results for ISCC and one for the hash scripts, for producing the results from that specific methodology represented by the folder name;
- results: where all results are divided into directories based on the script that generated them.

5.1.1 ID generation from Meta-Code

Within the path `id_or_generated_meta/script/iscc`, we find all the Python files related to the generation of the results from using Meta-Code for generating node IDs to be assigned to class images.

The file called `IdGen.py` is responsible for generating IDs using ISCC's Meta-Code and works as follows.

Inside we find four methods:

- `calcISCC(path, title, metas)`: this method takes as parameters the path to the image from which we want to generate the relevant ISCC codes, a title we want to assign to the image, and additional metadata in the form of a Python dictionary, metas, which will be encoded within the Meta-Code. Returns the Meta-Code and Content-Code related to the image passed as input;

- `calc_node_id(iscc_meta, iscc_content, r, g)`: parameters are the Meta-Code and Content-Code generated by the previous method, r being the number of bits that go to represent the ID of a node, and g, the number of characters in a chunk from which to compute the positions of the bits to be turned on within an ID initially set with all bits to 0. It returns three bit-strings in the form of a list, one representing the bit-string generated by the Meta-Code, one generated by the Content-Code, and a final bit-string, which in this case is assigned equal to the bit-string generated by the Meta-Code, as we will see later;

- `create_csv(rootdir, r, csv_folder, g)`: deals with creating the CSV files to save the generated ISCC codes with their respective IDs calculated by the previous method, broken down by imaged class. `rootdir` represents the folder of photographs, r the number of bits in an ID, csv_folder the directory where to save the CSVs created, and g the number of characters in a chunk from which to go to compute the bit to be turned on;

- `code_hamming_distance(csvs, hamming_csvs)`: creates the CSV files in which the hamming distances in the form of a distance matrix will be saved for the images within the classes. The two input parameters are two paths to two folders, the first representing the directory where to fetch the CSVs with the IDs inside that the images for each class have been associated with, and the second being the path of where to go to save the CSVs generated by the method.

calcISCC

```
1 def calcISCC( path, title, metas):
2
```

```
3    meta_serialized = jcs.canonicalize(metas)
4
5    du_obj = DataURL.from_data('application/json',base64_encode=True,
     data=meta_serialized)
6
7    ic_meta = idk.IsccMeta(name=title,description="description",meta=
     du_obj.url)
8    iscc_meta = idk.embed_metadata(path,ic_meta)
9
10   iscc_meta = idk.code_meta(iscc_meta).dict()
11
12   iscc_content = idk.code_content(path).dict()
13
14   return iscc_meta['iscc'][5:], iscc_content['iscc'][5:]
```

It is possible to import several functions necessary for the canonicalization of JSON data through the JCS library. It is crucial because, as written in the documentation of JSON Canonicalization (JCS) [26], it is necessary that during serialization, transport, or parsing, data does not change. The application of JCS is essential because metadata must be in the form of DataURL to be embedded, as it is possible to see on line 5 of the code.

After that, an IsccMeta object is created to which the chosen title for the image, a standard description, and additional metadata are assigned.

With the embed_metadata() method, `iscc_sdk` gives the ability to assign metadata to the image passed as a parameter of the path variable and finally generate the Meta-Code code via the `code_meta` method that takes the return variable on line 8 as a parameter.

For Content-Code, on the other hand, pass to the `code_content` method the path to the image from which you want to compute the relevant code. At the end

of the process of generating the desired components of ISCC, the two codes are returned, and being in the form of a dictionary, the value associated with the key 'iscc' is taken, discarding the first five characters since the string is in the form ISCC:AAA5K73C3EJLHHHT.

calc_node_id

Since the code for this method is quite long, it will be divided and explained in chunks.

This first piece of code depicts the method there as the two ISCC codes were divided into chunks based on the g parameter.

First, it is checked that g is between 1 and 15, after which a loop from 1 to the length of one of the two ISCC codes, i.e., 16, takes care of creating temporary strings to which a character of the code will be appended at each step. If the number of the loop we are in is divisible by g, then the temporary string will be added to an array representing the chunks of the code, and a new string will be created.

The steps explained in the previous paragraph are done for Meta-Code and Content-Code separately but within the same loop since both have a length of 16 characters.

```
1  def calc_node_id(iscc_meta, iscc_content, r, g):
2
3      if g > 15:
4          g = 15
5      elif g < 1:
6          g = 1
7
8      iscc_decomposed_meta = []
9      iscc_decomposed_content = []
10     tmp_str_meta = ""
11     tmp_str_content = ""
```

```
12
13    for i in range(0,len(iscc_meta)): # meta and content have the same
      length
14
15        tmp_str_meta += iscc_meta[i]
16        tmp_str_content += iscc_content[i]
17
18        if (i+1) % g == 0:
19            iscc_decomposed_meta.append(tmp_str_meta)
20            iscc_decomposed_content.append(tmp_str_content)
21            tmp_str_meta = ""
22            tmp_str_content = ""
23
24        if g % 2 != 0 and i == len(iscc_meta)-1:
25            iscc_decomposed_meta[len(iscc_decomposed_meta)-1] +=
      iscc_meta[i]
26            iscc_decomposed_content[len(iscc_decomposed_content)-1] +=
      iscc_content[i]
```

The following code shows how the node to which the image given as input to the method is assigned is calculated.

The following steps are accomplished in a for loop executed for the previously calculated number of chunks into which the two codes have been divided.

- for the i-th chunk, the corresponding hexadecimal value is calculated using the `hexlify` method;
- after that, this hex value is transformed into an integer, passing the value 16 as the basis for casting;
- finally the modulus between the integer value and *r*, i.e., the number of bits in an ID, is computed;

- this result represents the position of the bit to be turned on and is added to the corresponding list, representing the Meta-Code or Content-Code.

Once the two lists containing the bit positions to be set to 1 have been constructed, two for loops, one for Meta-Code, and one for Content-Code, will go to turn on the bits at the corresponding positions.

The variable final_r, which stands for the ID to which that image is to be assigned, is given the value corresponding to the bit-string generated by the Meta-Code, and all three IDs are returned by the method.

```
    r_bit_posisitons_meta = []
    r_bit_posisitons_content = []

    for i in range(0, len(iscc_decomposed_meta)): # meta and content have
    the same length

        deco_hex_meta = hexlify(iscc_decomposed_meta[i].encode()).decode
    ()
        deco_hex_content = hexlify(iscc_decomposed_content[i].encode()).
    decode()
        val_int_meta = int(deco_hex_meta,16)
        val_int_content = int(deco_hex_content,16)
        mod_meta = val_int_meta % r
        mod_content = val_int_content % r
        r_bit_posisitons_meta.append(mod_meta)
        r_bit_posisitons_content.append(mod_content)

    print(r_bit_posisitons_content)
    print(r_bit_posisitons_meta)
    r_string_meta = [0 for i in range(0,r)]
    r_string_content = [0 for i in range(0,r)]

    for el in r_bit_posisitons_meta:
```

```
21          r_string_meta[el] = 1
22
23      for el in r_bit_posisitons_content:
24          r_string_content[el] = 1
25
26      final_r = r_string_meta
27
28      return r_string_meta, r_string_content, final_r
```

create_csv

This method takes all the folders within the directory containing the images and creates a CSV for each folder, i.e., image class, which reports all the images for that class with their generated codes and IDs.

An Open Location Code is generated for each image, via the appropriate library, from geographic coordinates taken randomly. Of course, images belonging to the same class have identical locations.

After that, the relevant Meta-Code and Content-Code of the image are generated, and then compute the node ID assigned to that picture using the methods described above.

Once all the necessary informations are obtained, a list is created to which they are added, and this list is appended to the CSV. Once all the images belonging to a class have been processed, the generated CSV will be saved in the results folder based on the g parameter used to calculate the IDs.

```
1 def create_csv(self, rootdir, r, csv_folder, g):
2
3     for root, subFolders, files in os.walk(rootdir):
4
5         for dir in subFolders:
```

```
6
7           i = 0
8           dir_path = os.path.join(rootdir, dir)
9           dir_df = pd.DataFrame(columns = ['Img-Name','Meta','R-Meta','
    Content', 'R-Content', 'Final-R'])
10
11          latlong = {
12                      "lat": round(random.uniform(-190,190), 6),
13                      "long": round(random.uniform(-190,190), 6),
14                  }
15
16          _olc = olc.encode(latlong['lat'], latlong['long'], 8)
17          meta = {
18                  "@type":"Places",
19                  "olc": _olc
20              }
21
22          for file in os.listdir(dir_path):
23
24              file_path = os.path.join(dir_path, file)
25              file_title = "Img " + str(file)
26
27              iscc_meta, iscc_content = self.calcISCC(file_path,
    file_title,meta)
28              r_string_m, r_string_c, final_r = self.calc_node_id(
    iscc_meta, iscc_content, r, g)
29
30              file_codes = [str(file),iscc_meta,str(r_string_m),
    iscc_content,str(r_string_c),str(final_r)]
31
32              dir_df.loc[i] = file_codes
33              i += 1
34
```

```
35            dir_df.to_csv(csv_folder+"/"+str(dir)+".csv", index=False)
```

code_hamming_distance

This method aims to create a distance matrix reporting the distance between nodes saved in the CSVs created with the method above.

A CSV is read at a time via a for loop to accomplish this calculation. After that, as many distance matrices as there are columns in the CSV are created, leaving out the column representing the image names. For example, if the columns are: Meta, R-Meta, Content, R-Conten, and Final-R, five distance matrices will be created for each values contained within the column, separated by a line called `Type` within the CSV.

```
1  def code_hamming_distance(self, csvs, hamming_csvs):
2
3      for file in os.listdir(csvs):
4
5          if "hamming" not in str(file):
6
7              hamming_file = str(file)[:-4] + "_hamming.csv"
8
9              df_codes = pd.read_csv(os.path.join(csvs, file))
10             idx_h_df = list(df_codes['Img-Name'])
11             idx_h_df.insert(0,'Type')
12             cols = idx_h_df[1:]
13             df_codes = df_codes.set_index('Img-Name')
14
15             concat_df = pd.DataFrame()
16
17             for col in df_codes.keys():
18
```

```
19                sep = [col for i in range(0,len(cols))]
20
21                h_df = pd.DataFrame(columns=cols, index=idx_h_df)
22                h_df.loc['Type'] = sep
23
24                for idx, row in df_codes.iterrows():
25                        img = idx
26                        str1 = str(df_codes.loc[img,col]).replace(' ','')
    .replace('[','').replace(']','').replace(',','')
27
28                        for idx, row in df_codes.iterrows():
29
30                            str2=str(df_codes.loc[idx,col]).replace(' ','
    ').replace('[','').replace(']','').replace(',','')
31
32                            h_dist = hamming(list(str1),list(str2)) * len
    (list(str1))
33                            h_df.loc[img,idx] = round(h_dist)
34
35                concat_df = pd.concat([concat_df,h_df], ignore_index=
    False)
36
37            concat_df.to_csv(os.path.join(hamming_csvs, hamming_file))
```

5.1.2 ID generation from Content-Code

The script implementation mechanism regarding generating results resulting only from assigning images to nodes via Content-Code is purportedly identical to the previous section's.

The only line that changes is the assignment of the final_r a to the ID generated by the Meta-Code. In this case, the variable is assigned to the node ID generated

by the Content-Code, as shown in the code block below. As for the other three methods, they remain unchanged.

```
    final_r = r_string_content
```

As with Meta-Code, scripts and results are saved in the `id_or_generated_content` folder.

5.1.3 ID generation from Meta-Code & Content-Code

The implementation remains the same for the methodology that uses both codes to generate the ID assigned to the images. However, a third bit-string is created in which for each i-th position of the string, the result between the `OR` of the bit at the i-th position of the bit-string generated by the Meta-Code and the i-th position of the bit-string generated by the Content-Code is saved, as can be seen in the following piece of code.

```
    final_r = [0 for i in range(0,r)]

    for i in range(0,r):
        bool_m = bool(r_string_meta[i])
        bool_c = bool(r_string_content[i])
        if bool_c or bool_m:
            final_r[i] = 1
```

All scripts and the results derived from them for this methodology are saved in the folder `id_or_generated_final`.

5.2 Concatenated bit-string generation

As per the design, in addition to the generation of node IDs to be assigned to images described above, the type that exploits the concatenation of bit-strings generated by the characters of ISCC codes taken individually and then concatenated can be found in the following three folders:

- `id_concat_generated_meta`;
- `id_concat_generated_content`;
- `id_concat_generated_final`;

As with the implementations described in the previous section, within these three folders, we find two sub-folders called scripts and results that function in the same way as the sub-folders within the directories concerning implementations that do not use concatenation between bit-strings.

Scripts for generating IDs using ISCC codes are found in the path `folder_name/script/iscc`, where folder_name must be replaced with each of the folder names in the bulleted list.

The following sections will explain only how bit-string concatenation was implemented since the other three methods remained unchanged.

5.2.1 ID generation from Meta-Code

The bit-strings are created within the `calc_node_id` method. The initial part remains unchanged from the methodology without concatenation, and then for each character is computed its hexadecimal number, run the module with r, and a 16-bit bit-string is created in which the bit with the position resulting from the modulus is turned on. That bit-string is appended to a list that will result in the final node ID with a total of 256 bits.

```
def calc_node_id(iscc_meta, r, g):

    iscc_decomposed_meta = []
    tmp_str_meta = ""

    for i in range(0,len(iscc_meta)):

        tmp_str_meta += iscc_meta[i]

        if (i+1) % g == 0:
            iscc_decomposed_meta.append(tmp_str_meta)
            tmp_str_meta = ""

    final_r = []

    for i in range(0, len(iscc_decomposed_meta)):

        deco_hex_meta = hexlify(iscc_decomposed_meta[i].encode()).decode
    ()
        val_int_meta = int(deco_hex_meta,16) # base 16 for hexa string
        mod_meta = val_int_meta % r
        r_string_chunk = [0 for i in range(0,r)]
        r_string_chunk[mod_meta] = 1
        final_r.append(r_string_chunk)

    final_r_ = []
    for el in final_r:
        final_r_.extend(el)

    return final_r_
```

5.2.2 ID generation from Content-Code

The same is for generation from Content-Code, so in this case, the characters taken to calculate the positions of the bits to be turned on are those in the Content-Code generated from the image.

The code can be found in the block below.

```
def calc_node_id(iscc_meta, iscc_content, r, g):

    iscc_decomposed_content = []
    tmp_str_content = ""

    for i in range(0,len(iscc_meta)):

        tmp_str_content += iscc_content[i]

        if (i+1) % g == 0:
            iscc_decomposed_content.append(tmp_str_content)
            tmp_str_content = ""

    final_r = []

    for i in range(0, len(iscc_decomposed_content)):

        deco_hex_content = hexlify(iscc_decomposed_content[i].encode()).
    decode()
        val_int_content = int(deco_hex_content,16)
        mod_content = val_int_content % r
        r_string_chunk = [0 for i in range(0,r)]
        r_string_chunk[mod_content] = 1
        final_r.append(r_string_chunk)

    final_r_ = []
```

```
    for el in final_r:
        final_r_.extend(el)

    return final_r_
```

5.2.3 ID generation from Meta-Code & Content-Code

Whereas, as for generation by combining both codes, the two codes within the method are combined, as shown by the following code.

```
def calc_node_id(iscc_meta, iscc_content, r, g):

    iscc_decomposed_meta = []
    iscc_decomposed_content = []
    tmp_str_meta = ""
    tmp_str_content = ""

    for i in range(0,len(iscc_meta)):

        tmp_str_meta += iscc_meta[i]
        tmp_str_content += iscc_content[i]

        if (i+1) % g == 0:
            iscc_decomposed_meta.append(tmp_str_meta)
            iscc_decomposed_content.append(tmp_str_content)
            tmp_str_meta = ""
            tmp_str_content = ""

    final_r = []

    for i in range(0, len(iscc_decomposed_meta)):

```

```
23          deco_hex_meta = hexlify(iscc_decomposed_meta[i].encode()).decode
    ()
24          deco_hex_content = hexlify(iscc_decomposed_content[i].encode()).
    decode()
25          val_int_meta = int(deco_hex_meta,16)
26          val_int_content = int(deco_hex_content,16)
27          mod_meta = val_int_meta % r
28          mod_content = val_int_content % r
29          r_string_chunk = [0 for i in range(0,r)]
30          r_string_chunk[mod_meta] = 1
31          r_string_chunk[mod_content] = 1
32          final_r.append(r_string_chunk)
33
34      final_r_ = []
35      for el in final_r:
36          final_r_.extend(el)
37
38      return final_r_
```

5.3 ID generation from cryptographic Hash

As the last section is devoted to implementing the scripts for generating the IDs associated with the images, the cryptographic hash, i.e., the yardstick by which ISSC was tested, is presented.

The mechanism remains the same as the methods presented above, but with the difference that the script has one less method, `calcISCC`, and the hash generation is accomplished directly within the `calc_node_id` method.

In the code snippet presented below, the steps are as follows:

- first, the image is read from which then, via the `hashlib` library, the respec-

tive SHA256 will be calculated;

- after which the list containing the hash code chunks divided into g characters is created;

- Finally, the positions of the bits to be turned on within the bit-string are calculated by the method with the form explained above.

```
def calc_node_id(file_path, r, g):

    with open(file_path,"rb") as f:
        bytes = f.read() # read entire file as bytes
        readable_hash = hl.sha256(bytes).hexdigest()

    k = 0
    j = 1
    decomposed_hash = []
    while k < len(readable_hash):
        decomposed_hash.append(readable_hash[k:(g*j)])
        k = g*j
        j += 1

    r_bit = [0 for i in range(0,r)]

    for el in decomposed_hash:
        mod = int(el, 16) % r
        r_bit[mod] = 1

    return readable_hash, r_bit
```

The `create_csv` and `code_hamming_distance` are almost identical to the ISCC generation node ID method.

As presented in the Methodology chapter, such scripts are executed for different configurations of r and g, more precisely for r ranging from 8 to 1024 with steps of the power of 2, and for each value that r can take, the combinations with g can be 2 and 4 for ISCC, while 8 and 16 for hash, so that for both codes we have the same number of chunks from which we then go on to derive the positions of bits to be turned on, since they have different lengths, the SHA256 hash function outputs a string four times as long as the string produced by a single component of ISCC. Instead, to compare it to the generation of IDs from bit-string concatenation, g is set to 4, so there are 16 bits turned on out of 256.

5.4 Internal Hamming distance

In this section and the next two, no distinction will be made in the implementation between ID generation by concatenation and not since the mechanism is the same for both.

Once all the CSVs containing the matrices of the distances between the images within the classes are generated, it is possible to calculate an average internal distance present within a class, flanked by the standard deviation, so that it is possible to see if the values deviate much from the calculated average.

The code is developed in this way. There is a script called HammingMean.py in which only one function reads the CSV file, calculates the various metrics, and writes them to a JSON file.

The code will be presented in several blocks, given its length.

First, the path to CSV files is created based on r, and g that are passed as a parameter since these CSVs are located in a path representing r and g used to generate them, as specified in the methodology.

```
1 def calc_hamming_mean(self, r, g, r_dim_csvs):
```

```
    r_dir = "r-"+str(r)
    new_r_dir = os.path.join(r_dim_csvs, r_dir)

    hamming_mean_dict = {}

    g_dir = "g-"+str(g)
    g_path = os.path.join(new_r_dir, g_dir)
    hamming_mean_dict[g_dir] = {}

    iscc_dir = "iscc-csvs"
    iscc_path = os.path.join(g_path, iscc_dir)
```

After that, for each file that contains the word `hamming` in the name, since the CSVs carrying the IDs associated with the images for each class and those with the distance matrices are in the same folder named the same way but with the word hamming differentiating them, we go on to create a DataFrame pandas in which to save the values with which we will later work.

```
    for file in os.listdir(iscc_path):

        if "hamming" in str(file):

            csv = os.path.join(iscc_path, file)

            hamming_mean_dict[g_dir][str(file)] = {}
            df = pd.read_csv(csv)
```

Finally, since the CSV representing the distance matrices has as many of them as there are columns within the CSV in which the generated IDs were saved, a for loop will split those matrices into separate DataFrames on which to go and calculate the

mean and standard deviation with the `numpy` library, which will be added to the dictionary that is returned by the function and finally, written to a JSON file.

```
        for idx, row in df.iterrows():

            if idx % len(df.keys()) == 0:

                tmp_df = pd.DataFrame(columns=df.keys())
                arr = df[idx:(idx+len(df.keys()))][:]
                tmp_df = pd.DataFrame(arr)
                tmp_df = tmp_df.set_index('Unnamed: 0')
                code = tmp_df.loc['Type'][0] # take the type of the code
                tmp_df = tmp_df.drop('Type',axis=0)
                tmp_df = tmp_df.astype(np.float64)

                #tmp_df = tmp_df.mask(np.equal(*np.indices(tmp_df.shape))
    )
                df_lower = np.tril(tmp_df)
                df_lower[df_lower == 0] = np.nan
                mean = np.nanmean(df_lower)
                std = np.nanstd(df_lower)
                if isnan(mean):
                    hamming_mean_dict[g_dir][str(file)][code+"-mean"] = 0
                else:
                    hamming_mean_dict[g_dir][str(file)][code+"-mean"] =
    mean
                if isnan(std):
                    hamming_mean_dict[g_dir][str(file)][code+"-std"] = 0
                else:
                    hamming_mean_dict[g_dir][str(file)][code+"-std"] =
    std

     return hamming_mean_dict
```

The resulting JSON file from this code will include all the values that r can take, but only one value that g can take on. In this way, it is possible to divide the combinations of g into several files since the folder structure for saving the results separates files based on the values that g takes. The results will be better analyzed in the Results chapter.

Regarding calculating the average internal distances between images within the DHT generated by the concatenated generation ID typology and the hash, the methodology remains the same; what changes is the path to the CSVs.

5.5 Distance between image classes

As explained above, more than the internal distance is needed to understand the efficiency of image allocation within the hypercube. What is additionally needed is to understand how the classes behave with each other, whether they remain close together or whether they are spaced apart.

The script named Centroids.py contains two functions:

- centroids_median: It deals with calculating the centroid taken as the median between the IDs assigned to the images in the class. Its parameters are: r, g, r_dim_csvs, which is the folder where the CSV files are contained;

- hamming_centroids_median: for each class, calculates the hamming distance between the centroid and the centroids belonging to the other classes. Its parameters are g and `save_json`, the folder in which it saves the created JSON file. In this case, r is not passed as a parameter since this method is not called inside a loop that passes it r as a parameter but has it implemented inside.

centroids_median

As in the previous case, the path to fetch the CSVs containing the node IDs associated with the images must be computed.

```
def centroids_median(r, g, r_dim_csvs):

    centroids = {}

    r_dir = "r-"+str(r)
    new_r_dir = os.path.join(r_dim_csvs, r_dir)

    g_dir = "g-"+str(g)
    g_path = os.path.join(new_r_dir, g_dir)
    centroids[g_dir] = {}

    iscc_dir = "iscc-csvs"
    iscc_path = os.path.join(g_path, iscc_dir)
```

For each CSV file, the relative binary values of the IDs are transformed into decimals, from which an ordered list will be derived to fetch the element in the median position of the list and set it as a centroid.

```
    for file in os.listdir(iscc_path):

        if "hamming" not in file:

            csv_path=os.path.join(iscc_path,file)

            df = pd.read_csv(csv_path)

            final_r = df['Final-R']
            ids_to_num = []

```

```
            for el in final_r:
                bit_n = ""
                for j in el:
                    if j in ['0','1']:
                        bit_n += str(j)
                int_n = int(bit_n,2)
                ids_to_num.append(int_n)

            sorted = np.sort(ids_to_num)

            median_idx = int(len(sorted)/2)
            median = sorted[median_idx]

            getbinary = lambda x, n: format(x, 'b').zfill(n)

            centroid_node = getbinary(median, r)

            centroids[g_dir][str(file)] = {
                "centroid":str(centroid_node),
                }

    return centroids
```

Once the centroids of each class for different combinations of the two parameters are obtained, a nested JSON file is written, where the nested represents the combination, associating each class with its own centroid.

hamming_centroids_median

Once the JSON file containing the class centroids for different combinations of r and g is created, this method takes care of updating that JSON such that in the object containing the class centroid, there is another one inside which there are 29

objects of type key-value where the key is a class `i` represented by the object and the value is the hamming distance from class `j` different from the class `i`.

```
def hamming_centroids_median(g, save_json):

        centroids = json.load(open(os.path.join(save_json,'centroids-median-iscc.json'), 'r'))
        i = 8
        while i<=1024:

            r_dir = "r-"+str(i)
            g_dir = "g-"+str(g)
            keys = centroids[r_dir][g_dir].keys()

            for k in keys:

                k_centr = centroids[r_dir][g_dir][k]['centroid']
                centroids[r_dir][g_dir][k]['hamming_cents'] = {}
                for kj in keys:
                    if kj != k:
                        kj_centr = centroids[r_dir][g_dir][kj]['centroid']
                        ham_k_kj = hamming(list(k_centr), list(kj_centr)) * len(list(k_centr))
                        centroids[r_dir][g_dir][k]['hamming_cents'][kj] = ham_k_kj

            i *= 2

        with open(os.path.join(save_json,'centroids-median-iscc.json'), 'w') as fp:
            json.dump(centroids, fp)
```

Once this calculation is performed for all classes, the JSON file of centroids is updated with the hamming distances from the other classes, which will then be used in calculating the efficiency metric.

5.6 Efficency metric

The purpose of implementing this metric was to find which combination could maximize the distance between the classes, keeping the distance between the images of the classes from each other as small as possible.

The code and results can be found in the `compare_new_metric` folder, within which there are two main scripts:

- final_new_metric.py: it deals with the generation of a JSON file in which the metrics, calculated as described in the previous chapter, are saved, broken down by generation type. Thus, there are six main objects: three that involve bit-string concatenation and three that do not, in which other objects are nested, which we will go into later, based on the generation type;
- highlights.py: once the JSON with all the metrics has been created, this script generates another one in which it saves the minimum and maximum values obtained from the various combinations by going to analyze the JSON created by the previous script.

5.6.1 final_new_metric.py

This script is very long, so it will be analyzed in blocks. It reads the centroids calculated in the previous section, calculates the internal distance between the centroid and the other nodes to which the class images have been assigned, and finally, performs the subtraction between the two after calculating their averages.

It is composed of three main methods:

- `new_metric_concat(centroid_json, csv_dir)`: goes to compute metrics only for explained types that involve concatenation between bit-strings. As parameters, it provides the folder where the JSON files of the centroids are saved according to which type of ISCC code we are analyzing, whether Meta-Code, Content-Code, or the combination of the two called Final, and the folder where the CSVs are saved in which the image-ID association is written;

- `new_metric_or(r,g,rootdir_cents,csv_dir,check_iscc)`: deals with the calculation of the efficiency metric and the subsequent comparison between the metrics generated by ISCC and the hash for generation types that do not involve bit-string concatenation. Among the parameters, we find two of them the same as the previous method for centroids and node IDs to which the images have been associated, with the addition of r and g, since their combinations are multiple for these types of node ID generation, and the last parameter that takes care of checking whether we are analyzing data generated by ISCC or by hash, to change g accordingly.

- `compare_new_metric()`: The latter parameter-free method creates a JSON file in which all the results derived from the metrics for the different methodologies are encapsulated.

new_metric_concat

At the beginning of this method, the JSON file of centroids is read in which the hamming distances between one class and all others for each are stored. After that, all those data structures later used for the final metric calculation are instantiated.

```
1 def new_metric_concat(centroid_json, csv_dir):
2
```

```
    result_json = json.load(open(centroid_json, 'r'))

    dictj = result_json

    img_cls = dictj.keys()
    hamming_dict = {}

    hamming_mean_cents_array = []
    avg_internal_dist_array = []
    new_metric_array = []
```

Next, we enter a for loop that iterates over all the classes within the JSON and divides into two main sections: the first that deals with calculating an average of the hamming distances between the i-th class and all the other classes; the second that goes to retrieve from the CSV file all the IDs of the images associated with the i-th class and calculates their average distance from its centroid.

We will subtract these two measures from each other, and the resulting value will be added to a list whose average will return the efficiency metric.

It was decided to add two values representing, respectively, the average of the averages of the hamming distances between classes and the averages of the internal distances between classes with their standard deviations to go deeper into the result received from the efficiency metric.

```
    for k in list(img_cls):

        """  MEAN DISTANCE FOR CENTROIDS  """
        hamming_dict = dictj[k]['hamming_cents']
        hamming_mean_cents = np.mean(list(hamming_dict.values()))

        """  MEAN DISTANCE FROM CLASS CENTROID TO THE IMAGES  """
        k_csv = csv_dir+"/"+k
```

```
        df = pd.read_csv(k_csv)

        final_r = df['Final-R']
        k_centr = dictj[k]['centroid']
        sum_hamming = 0
        for el in final_r:

            el1 = str(el).replace(' ','').replace('[','').replace(']','').replace(',','')
            sum_hamming += hamming(list(k_centr), list(el1)) * len(list(k_centr))

        avg_internal_dist = sum_hamming / len(final_r)
        hamming_mean_cents_array.append(hamming_mean_cents)
        avg_internal_dist_array.append(avg_internal_dist)
        new_metric_array.append(hamming_mean_cents - avg_internal_dist)

    hamming_mean_cents = np.mean(hamming_mean_cents_array)
    hamming_sd_cents = np.std(hamming_mean_cents_array)
    avg_internal_dist = np.mean(avg_internal_dist_array)
    std_internal_dist = np.std(avg_internal_dist_array)
    new_metric = np.mean(new_metric_array)

    return hamming_mean_cents, hamming_sd_cents, avg_internal_dist, std_internal_dist, new_metric
```

new_metric_or

The code contained in this method is much the same as presented in the previous method. What changes is that here r and g are passed in such a way as to go to the folders and construct a nested JSON representing the combination of the two

values.

As can be seen below, the parameter `check_iscc` serves as a control to go in and modify g in case we are analyzing the results from applying the cryptographic hash to the images and then go in and quadruple the value of g.

```
def new_metric_or(r,g,rootdir_cents,csv_dir,check_iscc):

    if check_iscc == "iscc":
        centroid_json = os.path.join(rootdir_cents, "centroids-median-
    iscc.json")
    else:
        g = g*4
        centroid_json = os.path.join(rootdir_cents, "centroids-hash-
    median.json")

    r_dir = "r-"+str(r)
    g_dir = "g-"+str(g)

    result_json = json.load(open(centroid_json, 'r'))
    dictj = result_json[r_dir][g_dir]
    img_cls = dictj.keys()

    hamming_dict = {}
    hamming_mean_cents_array = []
    avg_internal_dist_array = []
    new_metric_array = []
```

From here to the end of the method, the mechanism and implemented code are almost to the code presented in the previous method. Then we iterate over the classes, calculate the average hamming distance between them and the average hamming distance within the class, and then subtract them and derive the efficiency metric.

Here, too, the averages of the two values that generated the efficiency metric are saved so that we can go and do a more in-depth analysis of the results.

compare_new_metric

This method reads all the folders in the project's root folder to check their prefix.

```
def compare_new_metric():

    root = ".."
    final_compare_json = {}

    for dir in os.listdir(root):
```

In the case where the prefix starts with `id_concat`, then we are going to build that part of the dictionary instantiated at the beginning, called `final_compare_json`, with the `new_metric_concat` method, separating the metric computed from the files generated with ISCC from the metric computed from the files generated with the hash. Finally, a subtraction between the two efficiency metrics derived from ISCC and hash is performed, and the result will or will not represent a better performance of ISCC than the hash.

```
        if "id_concat" in dir:

            centroids_json_iscc = root+"/"+dir+"/results/iscc/centroids-median-iscc.json"
            centroids_json_hash = root+"/"+dir+"/results/hash/centroids-hash-median.json"
            csvs_dir_iscc = root+"/"+dir+"/results/iscc/csvs"
            csvs_dir_hash = root+"/"+dir+"/results/hash/csvs"

            hamming_mean_cents_hash, hamming_sd_cents_hash, avg_internal_dist_hash, std_internal_dist_hash, new_metric_hash  =
```

```
    new_metric_concat(centroids_json_hash, csvs_dir_hash)
9           hamming_mean_cents_iscc, hamming_sd_cents_iscc,
    avg_internal_dist_iscc, std_internal_dist_iscc, new_metric_iscc  =
    new_metric_concat(centroids_json_iscc, csvs_dir_iscc)
10
11          compare_metric = new_metric_iscc - abs(new_metric_hash)
```

In the case where, on the other hand, the prefix starts with `id_concat`, we will use the `new_metric_or` method to go and compute the metrics on the JSON and CSV files, which, however, in addition to representing the methodology will also represent the combination of r and g as per the test design.

```
1  elif "id_or" in dir:
2
3           final_compare_json[dir] = {}
4
5           for g in [2,4]:
6
7               centroids_dir = root+"/"+dir+"/results/iscc-"+str(g)+"
    _hash-"+str(g*4)+"/centroids"
8               csv_dir = root+"/"+dir+"/results/iscc-"+str(g)+"_hash-"+
    str(g*4)+"/csvs/"
9
10              final_compare_json[dir]["iscc-"+str(g)+"_hash-"+str(g*4)]
     = {}
11
12              r = 8
13
14              while r<=1024:
15
16                  final_compare_json[dir]["iscc-"+str(g)+"_hash-"+str(g
    *4)]['r-'+str(r)] = {}
```

```
17
18                  hamming_mean_cents_hash, hamming_sd_cents_hash,
    avg_internal_dist_hash, std_internal_dist_hash, new_metric_hash  =
    new_metric_or(r,g,centroids_dir, csv_dir, 'hash')
19                  hamming_mean_cents_iscc, hamming_sd_cents_iscc,
    avg_internal_dist_iscc, std_internal_dist_iscc, new_metric_iscc  =
    new_metric_or(r,g,centroids_dir, csv_dir, 'iscc')
20
21                  compare_metric = new_metric_iscc - abs(
    new_metric_hash)
```

Once the dictionary, including all metrics for all types of ID generation, has been concluded, the JSON in which all values have been saved is created.

```
1    with open(os.path.join('./','compare_new_metric.json'), 'w') as fp:
2        json.dump(final_compare_json, fp)
```

5.6.2 highlights.py

Without showing the code, this script fetches for both the ID generation typologies and different combinations of the r and g parameters, the minimum value and the maximum value saved as `compare_metric`, that is, the value obtained by subtracting from the metric calculated for ISCC, the metric calculated for the cryptographic hash.

In this way, only the relevant results given by executing the efficiency metric will be analyzed.

5.7 Query Testing

This last section will present how a query system was implemented on a simulated hypercube to see how an actual system would behave in searching for images based on the chosen type.

The simulation was performed for an ISCC-based and hash-based architecture, so the two methodologies could be compared once the results were obtained.

The generation of the ID to be associated with the images is done by considering the Meta-Code of ISCC, and the reasons for this choice will be shown in the next chapter. As r values, 8, 12, and 16 were taken. While, for g, values 2 and 4 were chosen for ISCC, and consequently, values 8 and 16 were chosen for hash to compare which combination reacted best to image allocation within the DHT.

This section will not go deeper into the code, given that most of it have already been seen in previous sections.

There are two scripts:

- query-iscc.py: which deals with the generation of results based on the parameters r and g given as input regarding the allocation of images in the decentralized architecture given the generation of IDs via the Meta-Code;

- query-hash.py: creates results regarding the use of hash as a methodology for generating IDs from images.

The image class chosen for the evaluation of the goodness of the methodology is `san_petronio`, in which the images of the Basilica of San Petronio in Bologna are encapsulated. Thus, a picture that is not among those in the class was taken and evaluated by the query on the hypercube.

In addition to the images of the san_petronio class, five other classes were added to see how many resulting images were related to the image-query class.

The two scripts' mechanisms are the same, so no distinction will be made in explaining the implementation. The only thing that changes is how the ID was generated based on the chosen encoding.

5.7.1 Query script

First, the variables are initialized that we will later use to: access the picture with which run the query, access the picture classes, save the values of r and g, instantiate the connection to the IPFS node on Docker, and the dictionary that will contain the results of the query.

For writing the code, inspiration was taken from the implementation of a hypercube using Docker from this GitHub repository [27].

```
if __name__ == "__main__":

    query_image = "san_petronio.jpg"
    rootdir = '../photos/'
    r_list = [8, 12, 16]
    g = 2

    addr = '/ip4/0.0.0.0/tcp/5001'
    ipfs_client = ipfshttpclient.connect(addr)


    results = {}
```

After that, we enter a for loop that iterates over the values assigned to r, and the variables we will need to visit the nodes in the hypercube are instantiated, such as:

- HOPS: the counter of hops that are executed to search for the requested objects;

- NODES: The number of nodes present in the hypercube, that are 2^r;
- LABELS: all node IDs generated based on the NODES variable;
- graph: the hypercube-shaped graph generated via the `networkx` library;
- initial_id: The ID from which to start the hops for searching nodes in the hypercube.

```
for r in r_list:

    HOPS = 0

    NODES = 2 ** r

    LABELS = {tuple(int(j) for j in create_binary_id(i, r))
            : create_binary_id(i, r) for i in range(0, NODES)}

    graph = nx.relabel_nodes(
            nx.generators.lattice.hypercube_graph(r), LABELS)

    initial_id = ''
    for i in range(0,r):
        initial_id += '0'
```

Next, three methods are called that we will need to go and create all those files needed for the simulation:

- create_csv(rootdir, r, g): method previously seen, goes to create the CSV files for each class, in which the image-ID associations are saved;
- create_json_hypercube(LABELS, r): which creates a JSON file that will represent the simulated hypercube where key-value pairs correspond to ID-list of objects contained in the node with that ID;

- add_obj_ipfs(rootdir, ipfs_client): For each image saved in the CSVs of the image classes chosen for the test, it saves a copy on the IPFS node returning the relevant CIDs with which to download them.

```
        create_csv(rootdir, r, g)

        create_json_hypercube(LABELS, r)

        add_obj_ipfs(rootdir, ipfs_client)
```

Finally, the methods for creating and executing the architecture query are called.

First, the JSON representing the hypercube for the *r* that we are evaluating in the loop is read; second, the node in which the image with which the query is being run should fall is computed via the `calc_query_image_node` method; third, the query on the graph is run via the search method, which returns the number of hops run by the query and the objects that were found by the query.

```
        hypercube = json.load(open('r-'+str(r)+'/iscc/hypercube.json', 'r
    '))

        query_image_node = str(calc_query_image_node(query_image,r)).
    replace("[","").replace("]","").replace(",","").replace(" ","")

        hops_count_search, objects = search(graph, initial_id,
    query_image_node, hypercube, NODES, 15)
```

In the end, a variable called `true_obj` is created in which the CIDs corresponding to the images of the san_petronio class are stored, and through a for loop, how many of the objects returned by the query belonging to the class are evaluated to determine how many of them are correct.

These values are entered into the dictionary instantiated at the beginning of the

script, which will then be saved as a JSON file.

```
        true_obj = list(json.load(open('r-'+str(r)+'/hash/
    san_petronio_hashs.json','r')).values())

        positive = 0

        for el in objects:
            if el in true_obj:
                positive += 1
                print(positive)
```

Being the core method of the querying part, the `search` method that goes to perform the research within the hypercube will be briefly presented.

search

This function provides six parameters:

- graph: the graph created by `networkx` in the shape of a hypercube;
- initial_id: the node ID of the node from which to start the hops;
- keyword: the keyword generated by the image with which the query is being run;
- hypercube: the JSON file containing all the objects assigned to the nodes;
- nodes: the number of nodes present in the hypercube;
- threshold: the number of objects for which the function should stop searching.

The goal of the threshold parameter and to distinguish a Pin-search from a Superset-search. In the case where the threshold is set to -1, the former is executed, while for numbers greater than 0, the latter is executed.

As seen in the code presented below, in the case where the threshold value is -1, the if condition is met, and we enter another if in which we check whether the keyword of the node we are looking for matches the starting node. If it does, the hops are 0, and the objects belonging to that node are returned; otherwise, the shortest path between the starting node and the node we want to get to is calculated, and the hops are given by the number of nodes in the path minus one, i.e., the starting node.

```
def search(graph, initial_id, keyword, hypercube, nodes, threshold=-1):

    hops = 0
    if threshold == -1:

        if keyword == initial_id:
            return 0, hypercube[initial_id]
        else:
            path_to_node = nx.shortest_path(graph, initial_id, keyword)
            hops = len(path_to_node)-1
            return hops, hypercube[path_to_node[-1:][0]]
```

If the threshold has a value greater than 0, the Superset-search algorithm is applied, which works as follows.

An empty list is instantiated in which to go and save the objects resulting from the search. Secondly, the path to the node corresponding to the keyword generated by the image is computed, and the objects corresponding to that node are taken. Next, that node is set as the root of the search from which to search for all the neighbors to examine.

```
    elif threshold > 0:

        results_object = []
```

```
        path_to_node = nx.shortest_path(graph, initial_id, keyword)
        hops += len(path_to_node)-1
        root = path_to_node[-1:][0]
        results_object.extend(hypercube[root])
        connected_nodes = graph.neighbors(root)
```

From here, we enter a while loop that ends when the threshold is reached or if all nodes have been visited, and thus there are no more objects to search.

The while loop continues until the threshold has been filled or all nodes have been visited, and if one of these conditions is reached, the number of hops and the objects resulting from the search are returned. On the other hand, if the conditions are not reached, we continue to search the neighboring nodes until the condition is fulfilled. The script continues to cycle over the nodes near the root, and for each visited node, we add its ID to the list of checked nodes, append the objects contained within it to the results list, increase the hops and decrease the threshold based on the number of objects found. This operation will then be computed for all the neighboring nodes until one of the two conditions becomes false.

```
        visited = []
        while threshold > 0 and len(visited) < nodes:

            connected_nodes = list(connected_nodes)
            for node in connected_nodes:

                if node not in visited:
                    visited.append(node)
                    print(visited)
                    results_object.extend(hypercube[node])
                    hops += 1
                    threshold-=len(hypercube[node])
                    if threshold <= 0 or len(visited) >= nodes:
```

```
                        break

            if threshold >= 0 or len(visited) <= nodes:
                children = []
                for node in connected_nodes:
                    children.extend(graph.neighbors(node))
                connected_nodes = children

        return hops, results_object
```

Chapter 6

Results

This chapter will present the results obtained from the execution of the scripts presented in the previous chapter.

Results from internal hamming distance and distances between classes will not be shown, given that they are part of the Efficiency Metric algorithm.

6.1 Efficiency Metric

The analysis of the results regarding efficiency metrics will be carried out in this way. As the methodologies without concatenation have multiple combinations of r and g, as was explained in this 4.2.1, the output values given by the different combinations of g's will be examined individually. Two graphs will be shown: one for the combination of $g = 2$ for ISCC and $g = 8$ for the hash, and the other having the two values doubled. While for methods with concatenation, a graph will be displayed for each methodology, and considerations will be made in the final subsection.

The values reported in green in the images represent the result between the subtraction of the metric derived from ID generation by ISCC, blue color, minus the same one derived by hashing, orange color.

What is desired is for the metric derived from ISCC to be as much greater than the metric derived from using the cryptographic hash. This result suggests that ISCC is a better option for image allocation within a DHT. Thus if the absolute maximum value obtained from the subtraction of the two metrics absolute value is greater than the absolute minimum value, we would understand that the ISCC performs better than the hash. The opposite otherwise.

6.1.1 Non-concatenated bit-string generation

The results regarding the methods that do not involve concatenation will be formed by two graphs, more precisely two bar-plots: the first one represents the use of g equal to 2 for ISCC and equal to 8 for the hash, while in the second g was set to 4 for ISCC and 16 for hash.

This distinction can be noticed in the title of the subplots in the images.

Meta-Code

In the image 6.1, it is possible to see the minimum and maximum values generated by comparing the ISCC and hash metrics for the results from creating the IDs associated with the images using only the meta code.

Regarding g set to 2 for ISCC and 8 for the hash, we find that the number of bits present in a string that resulted in the minimum value was 8, meaning that with 8 identifying bits for a node and g set equal to 2, the metric of ISCC did not prove to be much better than the hash, as the difference is just over 1.

While the maximum for the combination of the different r's with g set equal to 2 for ISCC and 8 for hash was reached by r = 1024, with a value given by the subtraction of the two metrics of just over 6.

Concerning the combination of the different r's with g = 4 for ISCC and g = 16

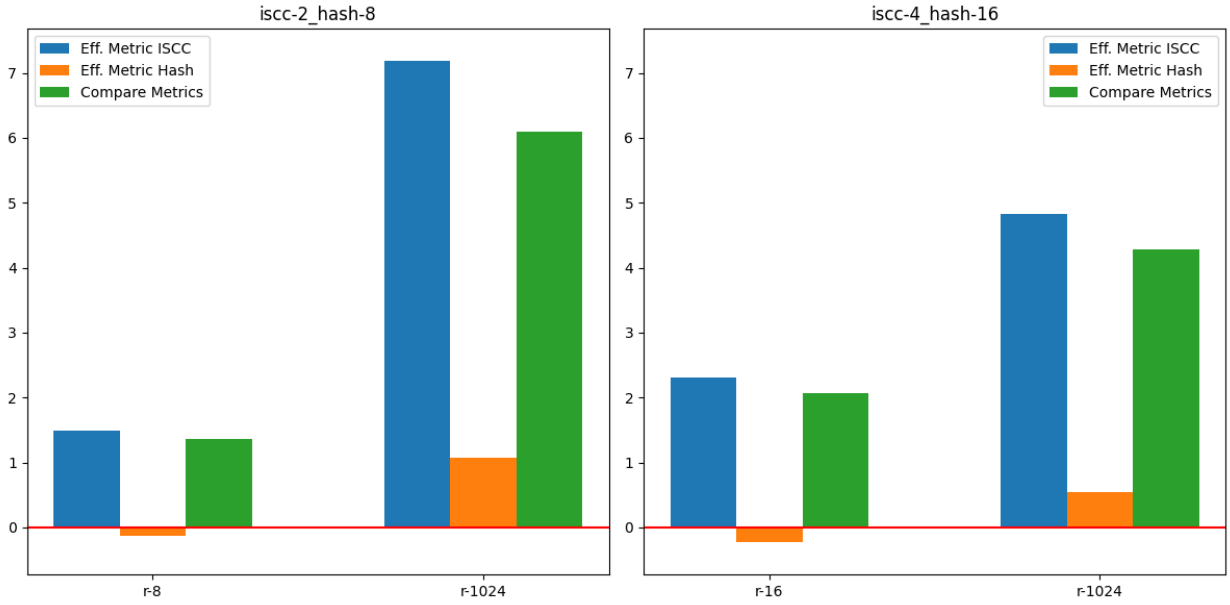

Figure 6.1: Efficiency Metric results from non-concatenated Meta-Code

for the hash, we find a better minimum value than the minimum of the previous g setting, given by $r = 16$. At the same time, the maximum is slightly lower, with a value just above four which, as in the previous case, was given by $r = 1024$.

So, having made these considerations, regarding the generation of IDs starting only from the Meta-Code, we have that $r = 1024$ and $g = 2$ are the best combination for the generation of IDs to be associated with images. It remains to be taken into account, however, that this type is affected by the image metadata, which can change and go to improve or worsen the performance. Nevertheless, in general, from the data resulting from such metrics, it was noted that the larger r is, the better Meta-Code performs than the hash.

Content-Code

Compared to the Meta-Code, the content code behaved slightly differently. In the image 6.2, you can see the two subplots related to the different values assigned

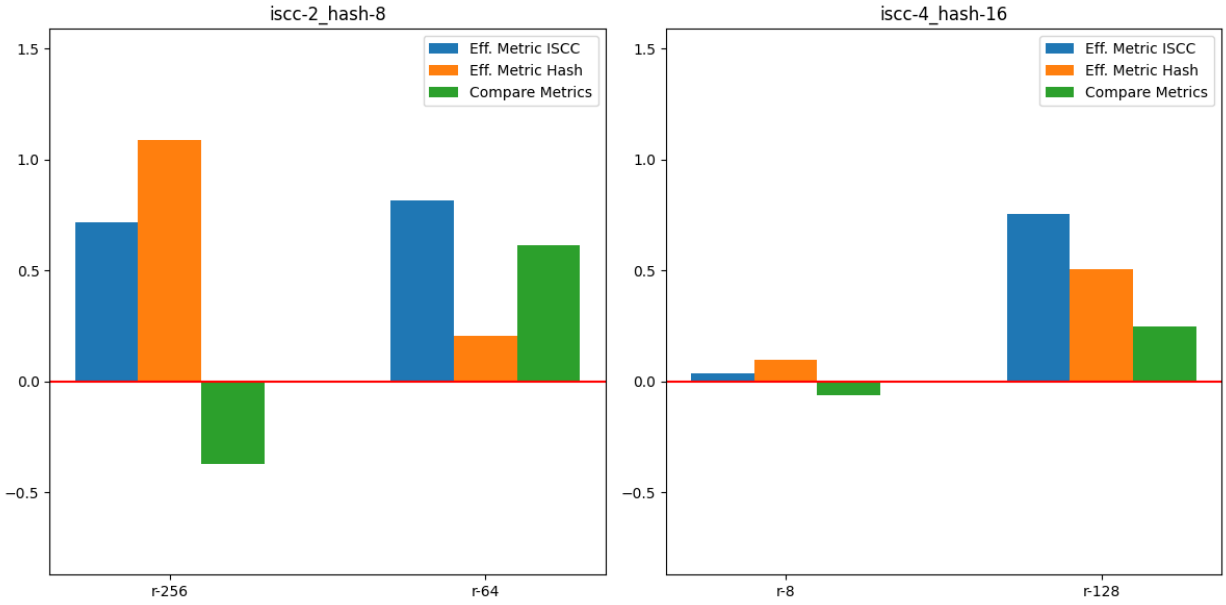

Figure 6.2: Efficiency Metric results from non-concatenated Content-Code

to g.

Here we do not have that in all cases the comparison of metrics was always better for ISCC, indeed, in the two minimal cases, we have that for $r = 256$, and $g = 2$ for ISCC and hash $g = 8$, and for $r = 8$ and $g = 4$ for ISCC and $g = 16$ for the hash, the generation of IDs from the decomposition of the cryptographic hash was better in image allocation than using Content-Code.

This phenomenon could be attributed to the fact that Content-Code uses the image's content, like its pixels. Since the class images represent the same subject in the picture but are very different from each other, the algorithm generates IDs from the Content-Code could vary a lot, so the hash has an efficiency metric greater than ISCC. However, the difference remains small since, for g set to 2, the comparison results in just under 0.4, and for g equal to 4, the subtraction is 0.05.

Analyzing the maximum values of both subplots, we can see that ISCC again has a more significant efficiency metric than the hash. However, only slightly, as

for $g = 2$, we have a result just above 0.6, given by $r = 64$, while for $g = 4$, we have a result equal to 0.25.

We can see that the behavior is more random than Meta-Code. We do not have an improvement as r rises, but it is a behavior that improves more based on the similarity of the images, regardless of r and g.

Meta-Code and Content-Code

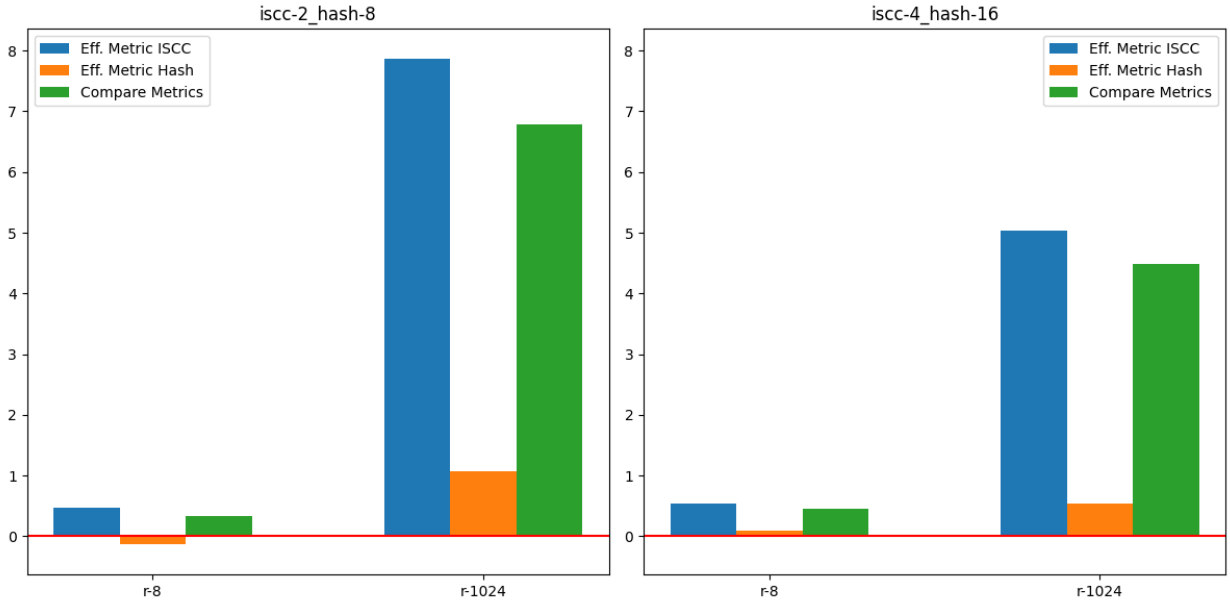

Figure 6.3: Efficiency Metric results from non-concatenated Meta-Code & Content-Code

From the generation of IDs using Meta-Code and Content-Code jointly, we see that the behavior of image allocation is very similar to that of Meta-Code used individually.

As seen in the image 6.3, we observe that from very small r's, the minimum value resulting from the subtraction of the two metrics from ISCC and hash is generated. In contrast, for very large r's, it can obtain a greater detachment between

ISCC and hash at the performance level, with ISCC performing better.

Another interesting observation is that the maximum values obtained from this type of ID generation are higher than the maximums obtained using Meta-Code alone but by very little. This behavior could be due to the use of Content-Code providing both an advantage and a disadvantage simultaneously. An advantage because it helps to put more bits in the strings and thus makes the IDs of similar images more similar. At the same time, the disadvantage is its poor performance in recognizing similar images.

Final consideration

In the previous subsections, we have seen how the various types of ID generation behave, starting from the components of ISCC taken separately and then joined.

Comparing them, the result is that using the Meta-Code and Content-Code joined to form an ID is the best option since they far outperform the hash.

Nevertheless, one essential detail emerges if we examine the data more closely. Looking at the averages of the hamming distances between the classes and the averages of the internal distances to the images of a class of the metric obtained by Meta-Code and the metric obtained by the conjunction of Meta-Code and Content-Code, it comes out that, in fact, the average distance between the centroids of the various classes resulting, from the technique involving the union of the two ISCC components, is almost twice as high as the same one generated by Meta-Code. However, regarding the distance within a class, we have that the average value resulting from using the two codes together is three times larger than that calculated from using the Meta-Code individually.

This detail goes to indicate to us that, in terms of performance, among which falls the number of hops to search for objects within the DHT inherent to the request

we are making, the use of Meta-Code alone is better suited for the generation of IDs to be associated with images in a hypercube architecture, because the number of hops will be fewer than if we had generated the IDs with the two components together.

6.1.2 Concatenated bit-string generation

This subsection will analyze the results by comparing the efficiency metric calculated on ID generation from ISCC and hashes. Then, we will compare how it differs between ID generation that does not involve bit-string concatenation.

Meta-Code

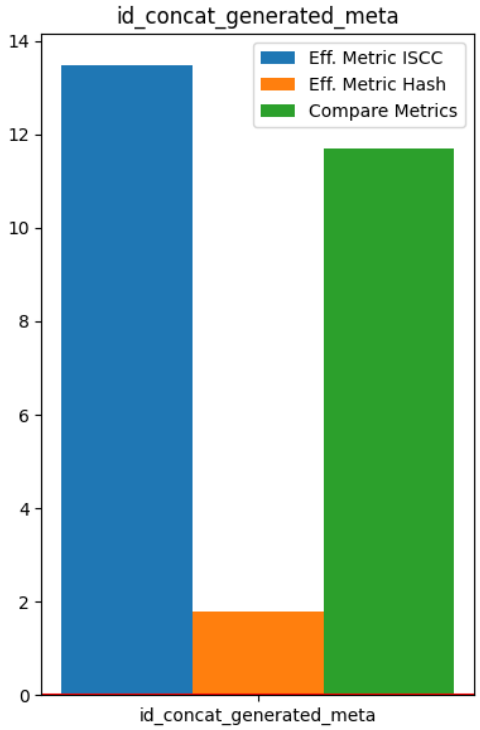

Figure 6.4: Efficiency Metric results from concatenated Meta-Code

Figure 6.4 shows the metrics and their comparison regarding the calculation of results with the IDs generated by the Meta-Code alone.

As can be seen, in this case, ISCC performs much better than the hash, returning a subtraction between the two of just under 12.

Content-Code

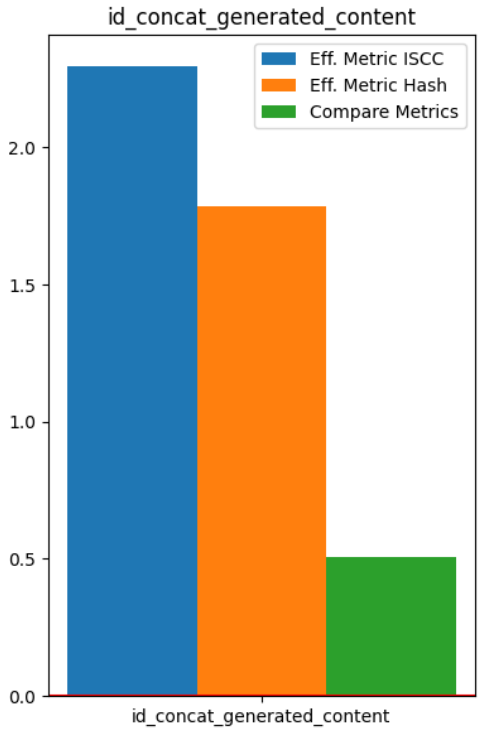

Figure 6.5: Efficiency Metric results from concatenated Content-Code

Content-Code also got the better of the hash, but less than Meta-Code, precisely because, as is written in the previous subsection, the behavior of Content-Code is more random, relying on the content of the images.

Figure 6.5 shows the bar plot in which the metrics and their comparison are shown.

Meta-Code & Content-Code

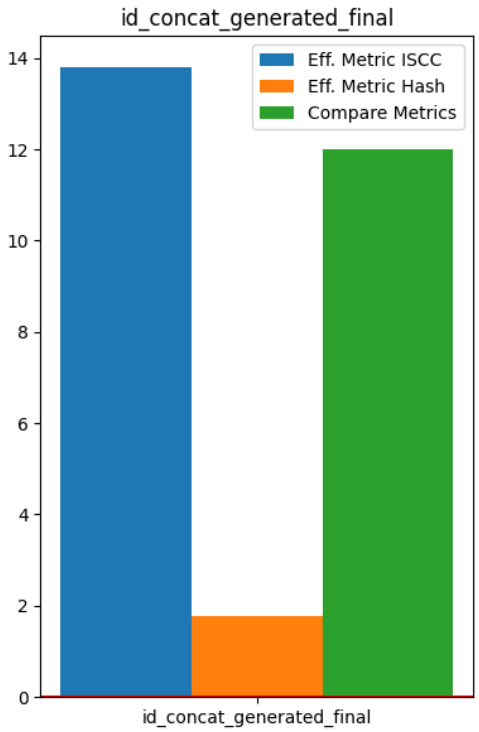

Figure 6.6: Efficiency Metric results from concatenated Meta-Code & Content-Code

Combining the two ISCC components results in behavior similar to the non-concatenation of bit-strings. As we can see from the graph in figure 6.6, which represents the metrics obtained from the conjunction of Meta-Code and Content-Code, similar results are obtained to the use of Meta-Code alone, 6.4, with a slight improvement, probably made by Content-Code for the reasons explained above, in the previous section.

Final consideration

Given the results of all three types of generating node IDs from the images using a technique of concatenating the bit-strings generated by the individual characters

of ISCC, the conclusion we tend toward is that the combination of the two components is the better choice.

Even for these methodologies, the situation arose where the values that generated the ISCC metrics by combining the two components performed worse in terms of the internal distance between classes, a factor probably given by the use of Content-Code. Therefore, considering the number of hops to be performed as a performance indicator, the generation of IDs via Meta-Code is the better choice.

6.1.3 Non-concatenation VS Concatenation

As a final comparison, we are left to examine which of the two methodologies, between using concatenation to form a bit-string in which each r bit represented the mapping of an ISCC character to the ID or going to set a predefined number of r bits and map a set of characters within that ID; performed better.

Keeping in mind the final considerations written for both methodologies, only the Meta-Code will be considered as the starting code to generate the hypercube keywords.

In the type that does not involve concatenation, the number of bits required to make ISCC perform better than the hash was 1024, either for g set equal to 2 or equal to 4, but of the two g's, the former performed better; it is possible to see the results in this graphic 6.1.

Comparing the value just mentioned, which corresponds to just over 6, generated by an efficiency metric on ISCC of 7, with the comparison of the two metrics derived from ISCC and hash in the case where concatenation is applied, it turns out that the latter performs much better than the former.

6.2 Query testing

The results for query testing on the simulated hypercube will be presented in this way. Three subsections will be created: the first one that will compare the behavior of ISCC with the hash having g set to 2 for the former and 8 for the latter; The second subsection will be presented in the same way but with the results generated with g doubled compared to the first one; inside the third subsection, final considerations will be made about what was obtained.

As mentioned in previous chapters, the r's used for hypercube creations are 8, 12, and 16.

Regarding the limit set to Superset-search, since there are 15 images in a class, this value was entered as the number to stop the search.

The bar graphs shown in the following images have three columns for each r. The first column represents how many of the images returned by the search match the images of the relevant photo class; The second column reports the total number of objects returned by the visited nodes, and the third and final column is the number of hops computed to arrive at returning the limiting number of objects.

6.2.1 Searching with ID generating from g equals 2 and 8

Pin-search

Figure 6.7 shows the query results for all three r with Pin-search performed on the node generation from the Meta-Code of the images.

While in figure 6.8, the results of the query performed on the simulated hypercube by basing the generation of IDs on cryptographic hashes are shown.

Neither configuration was able to find a node among those to which the images related to the san_petronio class were assigned, and the number of hops represents the hamming distance between the node with all bits set to 0 and the node that was

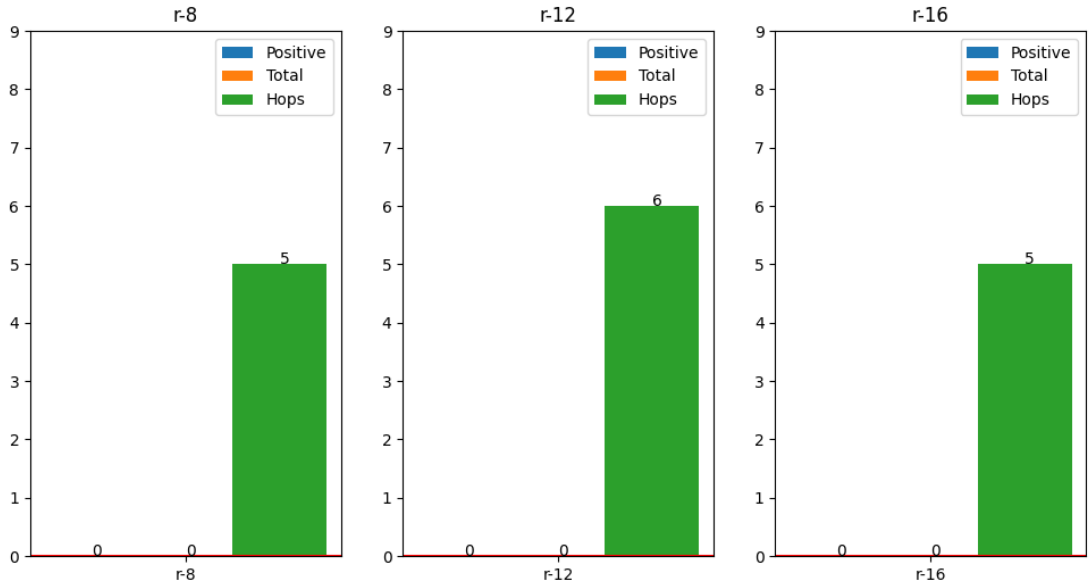

Figure 6.7: Pin-search results from ISCC ID generation with g=2

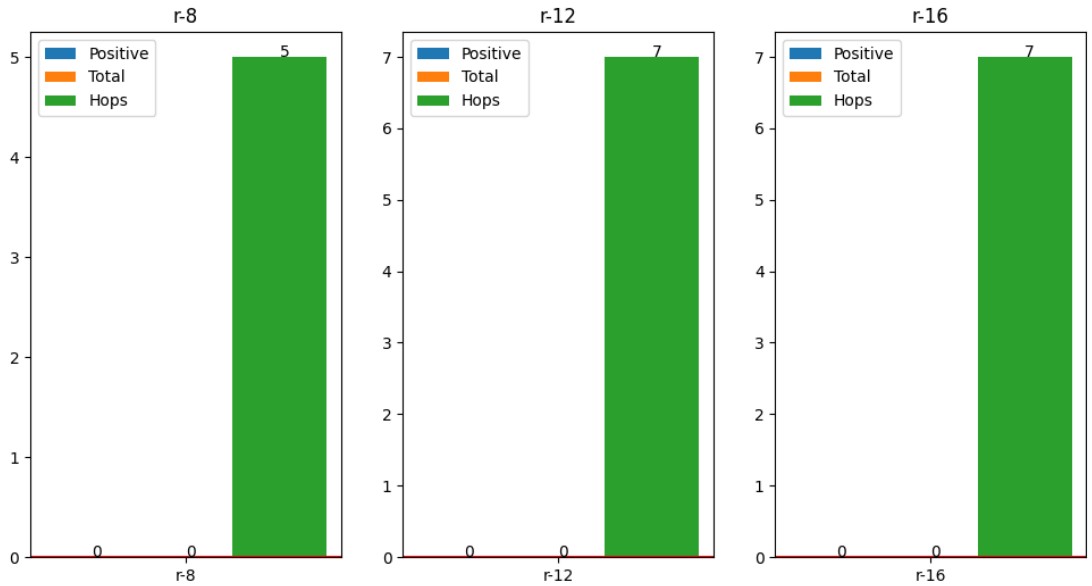

Figure 6.8: Pin-search results from Hash ID generation with g=8

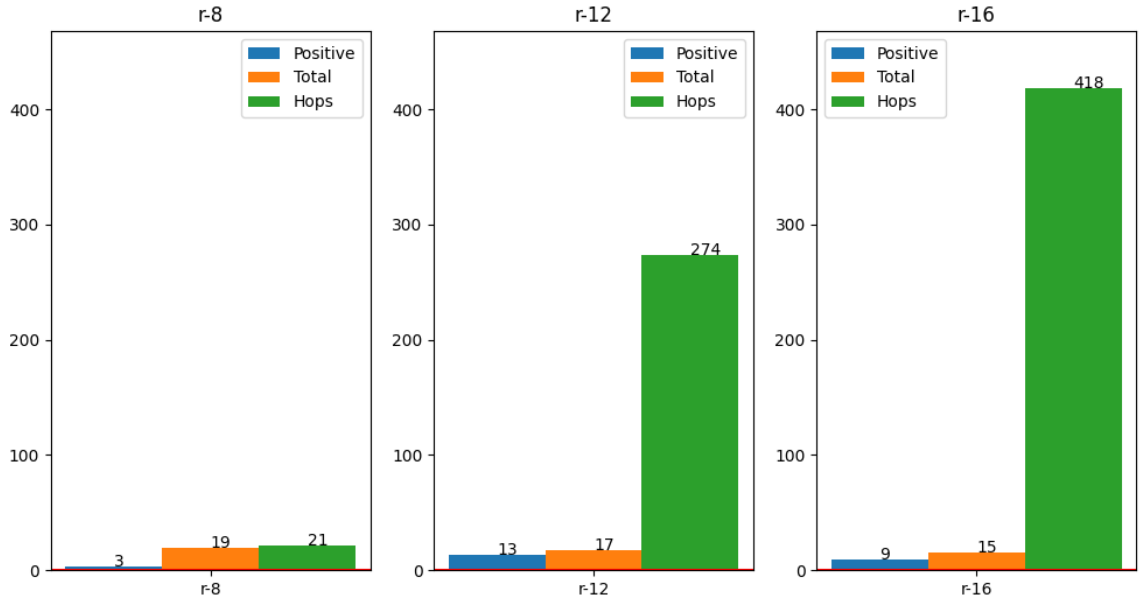

Figure 6.9: Superset-set search results from ISCC ID generation with g=2

generated by the algorithm to which the image was given as input to perform the query.

Superset-search

In contrast, as for Superset-search, the results are more attractive.

In Figures 6.9 and 6.10, we can find the query results for ISCC and Hash, respectively. The number of hops has increased by far, but so has the number of images related to the class we are searching for.

The r that performed best concerning ISCC was 12, with a total of 17 objects returned, 13 belonging to the class we are looking for, which is two less than the total number of images contained within it. There is also to take into account the hops, which are far more, 253 to be precise, than those obtained by the hypercube

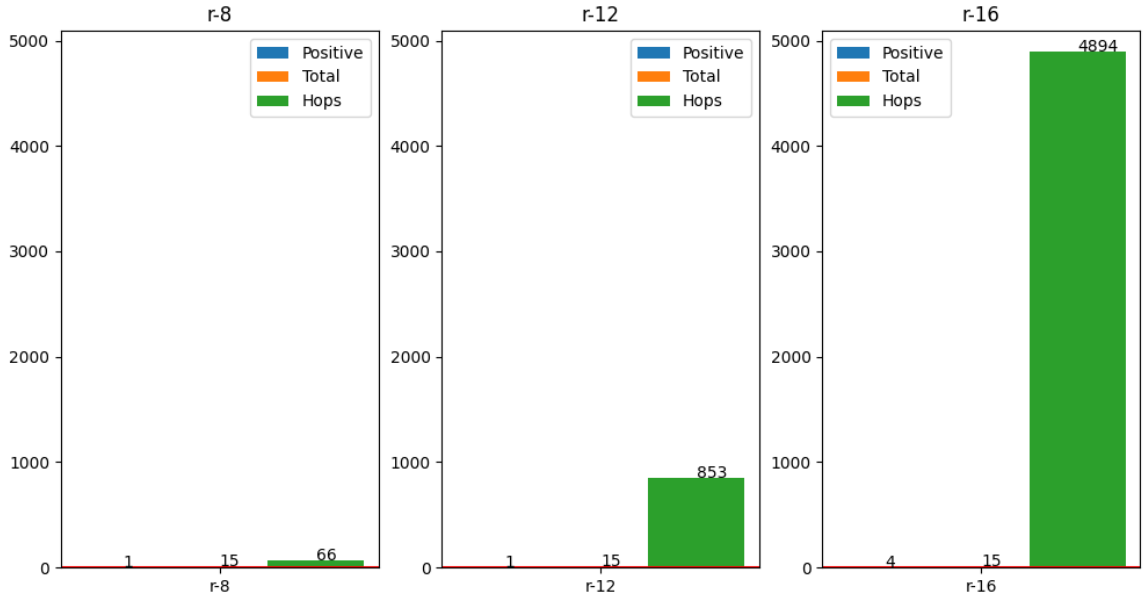

Figure 6.10: Superset-set search results from Hash ID generation with g=8

whose nodes are identified by 8 bits, which, however, found only three images inherent to the class out of 19.

On the hash side, however, we see much worse performances, with a maximum of 4 out of 15 images found belonging to the searched class. Moreover, the number of hops accomplished is ten times larger than the maximum number of hops achieved by ISCC with r equal to 16.

6.2.2 Searching with ID generating from g equals 4 and 16

Pin-search

Doubling the g, we see a substantial change as far as ISCC is concerned. In figure 6.11, it is possible to see that there were matches for r equals 8 and 16. However, only in the hypercube with eight identifying bits does the searched node possess an

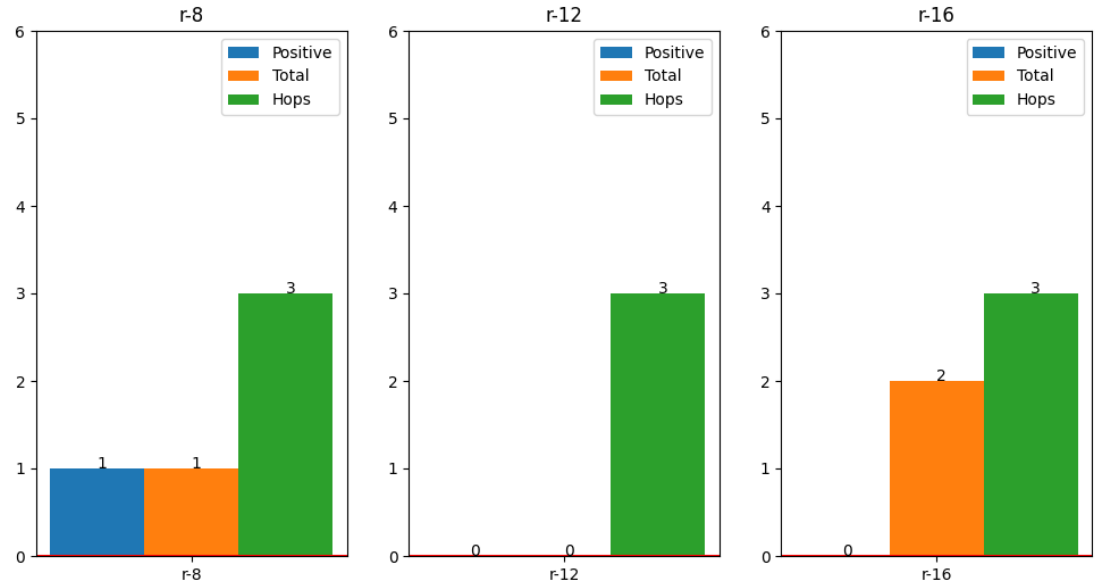

Figure 6.11: Pin-search results from ISCC ID generation with g=4

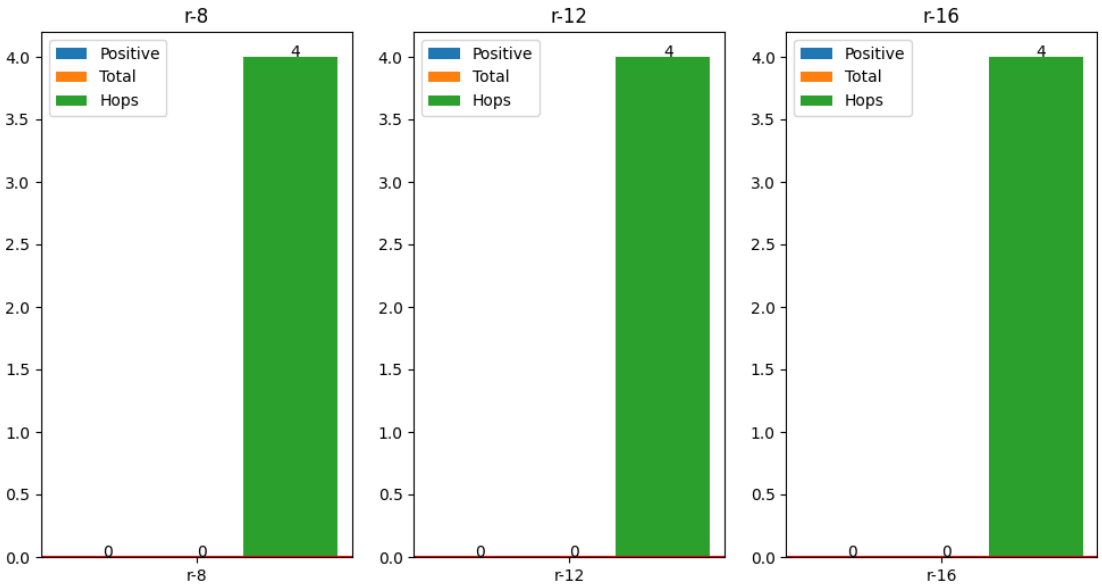

Figure 6.12: Pin-search results from Hash ID generation with g=16

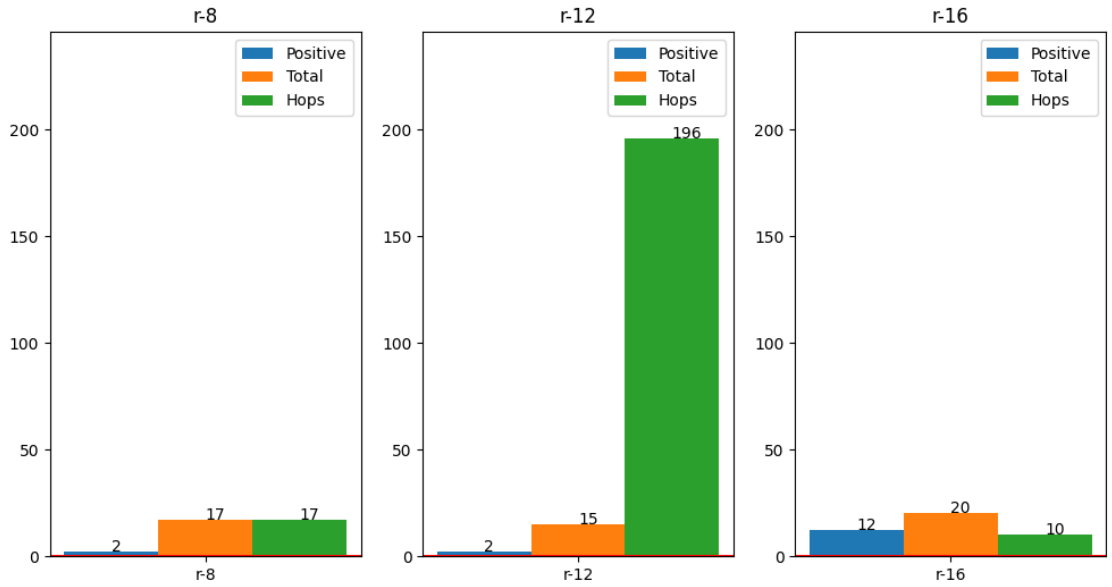

Figure 6.13: Superset-set search results from ISCC ID generation with g=4

image belonging to the class san_petronio.

In contrast, as far as the cryptographic hash is concerned, the behavior remains identical to the previous subsection. In the nodes found, no images or images belonging to the wished class existed. The results can be seen in figure 6.12.

Superset-search

Superset-search also showed exciting results, especially regarding the application of ISCC to a hypercube with 16 identification bits. As shown in figure 6.13, we note how with only ten hops, 20 objects were retrieved, 10 of which belong to the desired class.

On the other hand, the hash did not prove capable of holding its own, as the best results are obtained from using 8 and 16-bit identifiers, having both retrieved 15 images, 4 of them belonging to the san_petronio class. However, as shown in

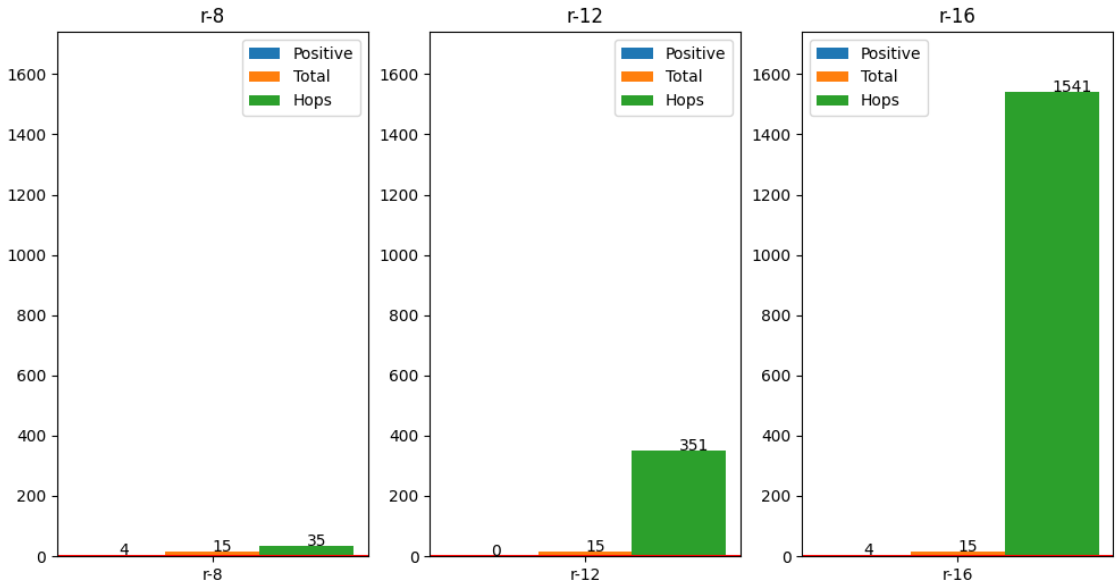

Figure 6.14: Superset-set search results from Hash ID generation with g=16

figure 6.14, the number of hops performed is much greater than the number of hops of the hypercube that uses ISCC's Meta-Code to generate the node IDs to be assigned to the images.

6.2.3 Final consideration

After analyzing the two research techniques on the two types of ISCC and cryptographic hash code usage based on theoretical bounds tested through efficiency metrics and a simulated hypercube query system, it is possible to say that ISCC performs better in allocating media files within a hypercube-shaped Distributed Hash Table.

In particular, the Meta-Code component of ISCC has enabled promising results on such an architecture, capturing, through the metadata inserted in the images, the

similarity between images and placing them close together within the DHT. The allocation allowed us to obtain a discrete number of images related to the query being accomplished while containing the number of hops within a limited value, considering that the smallest hypercube on which the query system was tested includes 256 nodes, and the largest one contains more than 60 thousand nodes.

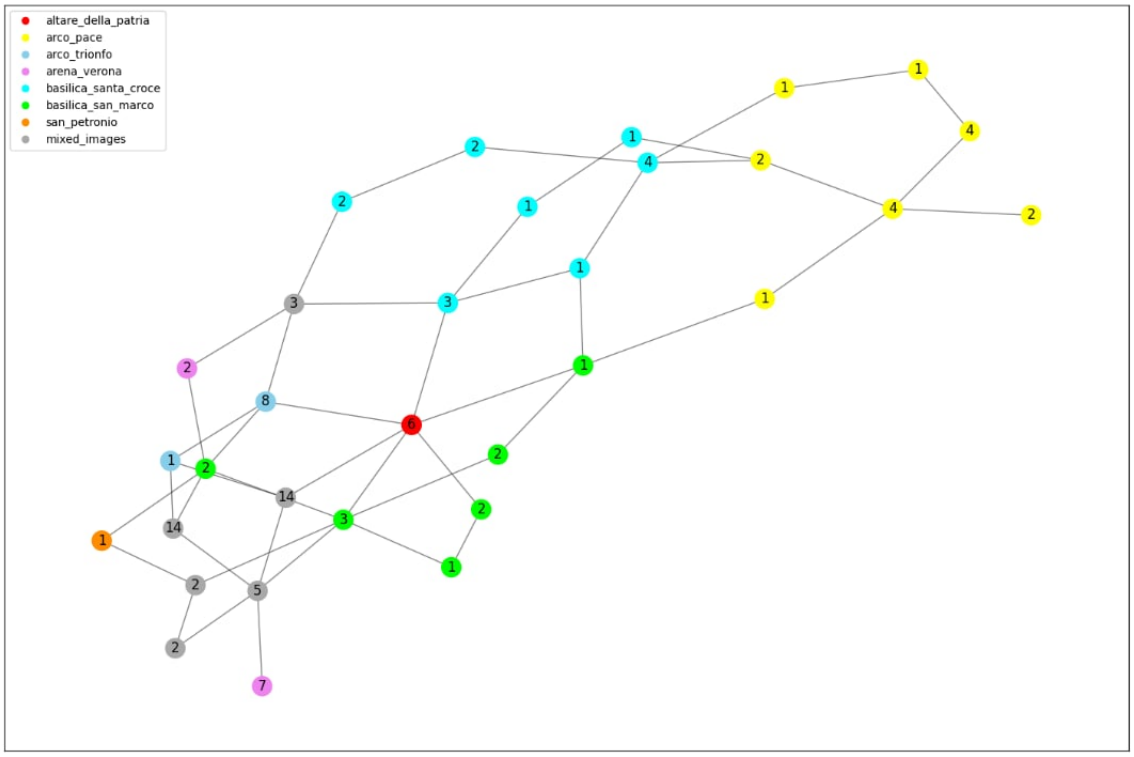

Figure 6.15: Sub-graph of nodes containing images with ISCC allocation

Figure 6.15 and 6.16 show how the images were allocated over the hypercube nodes with eight identifying bits, respectively, for ISCC and cryptographic hash.

We can notice that ISCC obtains a more effective clustering for images of the same class; the nodes containing the images of the same class stay closer, and the ones containing images of different classes stay far apart. Instead, the image allocation with hash is more spread over the hypercube, causing more hops to reach the object's threshold and performing worse in terms of returned images coherent with

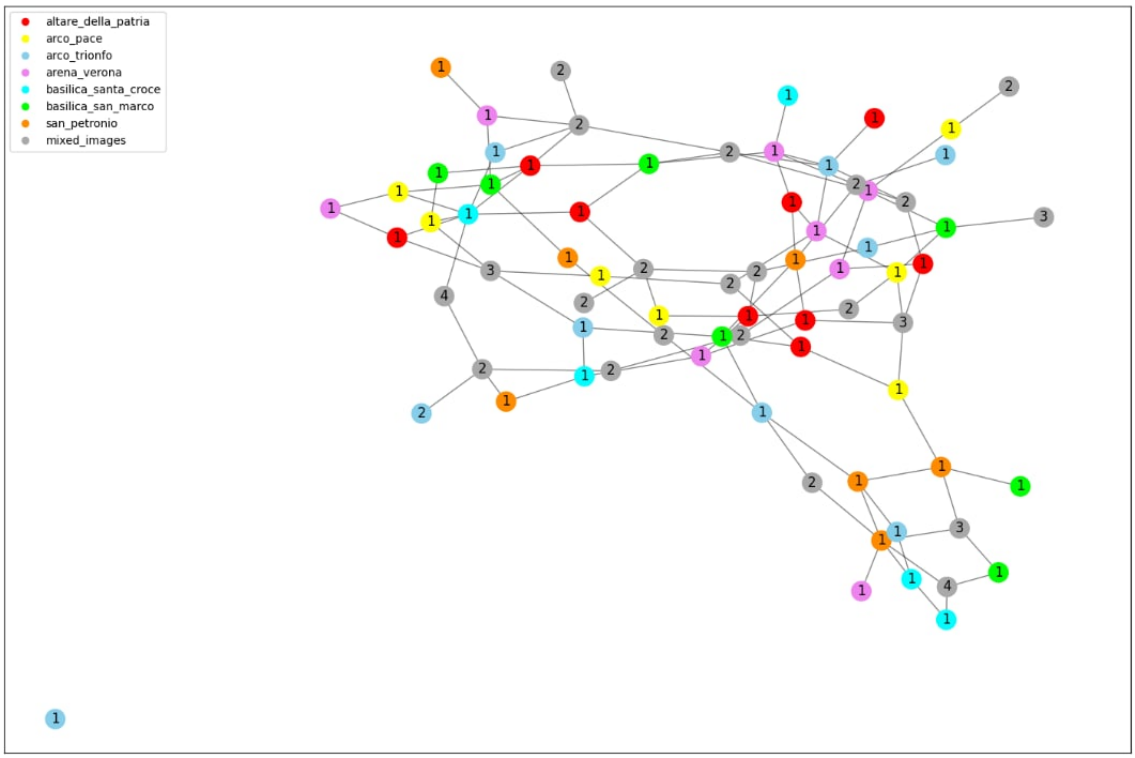

Figure 6.16: Sub-graph of nodes containing images with Hash allocation

the query.

In both subgraphs, nodes colored in gray represent them which contain images among more classes than one. In the ISCC subgraph, 6.15, the san_petronio, arco_trionfo, and arena_verona classes have only one or two nodes associated and stay closer to the nodes that contain mixed images. This means that most images were mixed in some nodes, leading to the Pin search and Super-set search results.

The spread in the hypercube created with the hash technique, 6.16, is due to the few images allocated in each node. Indeed, more have only one image associated, as reported from the node labels, and this leads to a significant number of hops and few images returned correlated with the query.

Chapter 7

Conclusions

In this thesis, a decentralized architecture based on two technologies was presented: Distributed Hash Tables (DHT), in this case in the form of a hypercube, and Decentralized File Storage (DFS), i.e., IPFS.

The purpose of this work was to look for a more efficient methodology of assigning objects, saved as hashes of them, to the nodes of the DHT in such a way as to reduce as far as possible the number of hops, i.e., jumps between peers, and make sure that the objects returned belonged to the search that is being performed.

Image type files were chosen as the use case for this objective to try to place similar images close together and different images far apart between nodes.

To this end, a new technology called the International Standard Content Code (ISCC) has been introduced to identify digital files within decentralized systems. This standard uses content-driven, locality-sensitive, and similarity-preserving hashing functions. It is composed of four components that go to identify a media file, of which only two have been chosen for the generation of the node IDs to which the images within the DHT should be assigned, namely Meta-Code and Content-Code. The former bases the generation of the file ID on the metadata it contains, the latter on the file's actual content, for example, the pixels in an image.

After that, the test design and implementation were presented to check how ISCC performed compared to a system that assigns IPFS CIDs to the nodes in a more random manner by exploiting the SHA-256 of the images.

The results showed that ISCC's Meta-Code best fits the purpose of the thesis by obtaining discrete values regarding the efficiency metric designed to measure the performance of various architecture implementations. Later, that system was used to develop a simulation to perform a query on the hypercube and see how the search using the Pin-search and Superset-search systems was performed.

The theoretical tests proved correct in identifying Meta-Code as a better tool than hash for a complex query system based on a hypercube-shaped distributed hash table. The hops accomplished within the architecture are smaller than that based on ID assignment from the image hash and with better results from the point of view of the returned objects matching the searched objects.

7.1 Future developments

Future developments that can be performed to try to improve the use of ISCC in image allocation within a hypercube-shaped Decentralized Hash Table architecture are:

- run new tests with new combinations of r and g to see if there are better ones;
- see if using different components of ISCC improves or worsens performance, e.g., taking Meta-Code and Data-Code;
- to see if new types of bit-string generation other than the one implemented in this thesis work, e.g., going for the base58-iscc characters based on their position and meaning shown in figure 2.3, can improve the allocation of images within nodes;

- tests new operators in addition to OR to see how image allocation behaves.

Chapter 8

Bibliography

[1] Iscc. https://iscc.codes.

[2] Iscc structure. https://iscc.codes/specification/#iscc-structure.

[3] Iscc component headers. https://iscc.codes/specification/#list-of-component-headers.

[4] Iscc content identification. https://iscc.codes/features/#related-product-identification.

[5] Mirko Zichichi, Luca Serena, Stefano Ferretti, and Gabriele D'Angelo. Governing decentralized complex queries through a dao. In *Proceedings of the Conference on Information Technology for Social Good*, pages 121–126, 2021.

[6] Barbara Guidi, Andrea Michienzi, and Laura Ricci. Data persistence in decentralized social applications: The ipfs approach. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4. IEEE, 2021.

[7] Aaron Grunthal. Efficient indexing of the bittorrent distributed hash table. *arXiv preprint arXiv:1009.3681*, 2010.

[8] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

[10] Iscc iso. https://www.iso.org/standard/77899.html.

[11] Base58-iscc. https://iscc.codes/specification/#base58-iscc.

[12] Iscc sdk. https://sdk.iscc.codes/.

[13] Iscc: Meta-code. https://iscc.codes/specification/#generate-meta-code.

[14] xxhash64. http://cyan4973.github.io/xxHash/.

[15] Iscc: Content-code. https://iscc.codes/specification/#content-code-text.

[16] Iscc: Data-code. https://iscc.codes/specification/#content-code-text.

[17] Iscc: Instance-code. https://iscc.codes/specification/#content-code-text.

[18] The graph. https://thegraph.com/.

[19] Interplanetary file system search engine. https://github.com/ipfssearch/ipfssearch.

[20] Nawras Khudhur and Satoshi Fujita. Siva-the ipfs search engine. In *2019 Seventh International Symposium on Computing and Networking (CANDAR)*, pages 150–156. IEEE, 2019.

[21] Mirko Zichichi, Luca Serena, Stefano Ferretti, and Gabriele D'Angelo. Complex queries over decentralised systems for geodata retrieval. *IET Networks*, 2022.

[22] Rodolfo da Silva Villaca, Luciano Bernardes de Paula, Rafael Pasquini, and Maurício Ferreira Magalhaes. Hamming dht: Taming the similarity search. In *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*, pages 7–12. IEEE, 2013.

[23] Robert Morris, M Frans Kaashoek, David Karger, Hari Balakrishnan, Ion Stoica, David Liben-Nowell, and Frank Dabek. Chord: A scalable peer-to-peer look-up protocol for internet applications. *IEEE/ACM Transactions On Networking*, 11(1):17–32, 2003.

[24] Open location code. https://maps.google.com/pluscodes/.

[25] Git repository implementation. https://gitlab.com/ema.fazz/tesi_lm_1.

[26] Json canonicalization scheme. https://www.rfc-editor.org/rfc/rfc8785.

[27] Hypercube on ipfs. https://github.com/anansi-research/hypfs.

Ringraziamenti

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito alla realizzazione di questo elaborato.

Un ringraziamento particolare va al mio relatore, il Professor Stefano Ferretti e il correlatore, il Dottor Mirko Zichichi, per il supporto datomi durante il periodo di tirocinio e tesi.

Ringrazio i miei genitori per il loro sostegno non solo economico, ma soprattutto emotivo, fornendomi gli strumenti per raggiungere questo traguardo, spronandomi a dare il meglio e incoraggiando le mie decisioni. Insieme a loro voglio ringraziare le mie sorelle, Valeria, Ilaria e Rebecca, sempre presenti nello svago e nell'ascolto, consigliandomi nelle scelte.

Grazie ai miei amici, nonché quella che definirei una seconda famiglia, Gianmarco, Giacomo e Stefano. Con loro ho condiviso le gioie e le difficoltà del nostro percorso universitario, aiutandoci l'un l'altro, senza mai privarci dei momenti di divertimento.

Ringrazio tutti i miei amici d'infanzia e adolescenza, pronti ad accogliermi durante le mie permanenze nella mia città natale, nonostante la lontananza.

Ringrazio Michele, per il suo sostegno morale durante gli studi, date le sue pressioni e costanti domande sugli esami da affrontare, anche ad orari improponibili.

Ringrazio la mia numerosa famiglia, in particolare i cugini con cui sono cresci-

uto: Carlotta, Veronica, Luca e Matteo. Con loro non è mai mancata una pausa caffè e sigaretta.

Voglio ringraziare Lisa e Stefano, per essere sempre stati presenti e avermi per primi accolto in questa magnifica città, lasciandomi, in alcune occasioni, accudire Ernesto con il quale ricordo con affetto le lunghe passeggiate.

Grazie alle mie attuali coinquiline, Stefania ed Alessia, per la loro disponibilità e comprensione durante i momenti più impegnativi nella preparazione di esami e tesi.

Infine, ultima, ma non per importanza, ringrazio Elisabetta, sempre pronta a starmi accanto, sia nei momenti di gioia, sia in quelli più duri, senza mai lamentarsi e con affetto accettando quelli che sono miei difetti.