Succinct and Self-Indexed Data Structures for the Exploitation and Representation of Moving Objects

Autor: Adrián Gómez Brandón

Tesis doctoral UDC / 2020

Directores:
Gonzalo Navarro
Nieves Rodríguez Brisaboa

UNIVERSIDADE DA CORUÑA

Succinct and Self-Indexed Data Structures for the Exploitation and Representation of Moving Objects

Autor: Adrián Gómez Brandón

Tesis doctoral UDC / 2020

Directores:
Gonzalo Navarro
Nieves Rodríguez Brisaboa

Programa Oficial de Doutoramento en Computación

UNIVERSIDADE DA CORUÑA

PhD thesis supervised by
Tesis doctoral dirigida por

Gonzalo Navarro
Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
851 Santiago (Chile)
Tel: +56 2 29784952
Fax: +56 2 26895531
`gnavarro@dcc.uchile.cl`

Nieves Rodríguez Brisaboa
Departamento de Computación y Tecnologías de la Información
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
`brisaboa@udc.es`

Gonzalo Navarro y Nieves Rodríguez Brisaboa, como directores, acreditamos que esta tesis cumple los requisitos para optar al título de doctor internacional y autorizamos su depósito y defensa por parte de Adrián Gómez Brandón cuya firma también se incluye.

Á miña familia

Acknowledgements

Finally! This stage as Ph.D. student comes to an end. A stage full of lessons, deadlines, travels, ups and downs. I must start thanking my advisors, without them this thesis will not be possible: Nieves and Gonzalo. Nieves supported me from the beginning to the end of this long way. Thanks for the help, confidence, advice, and making everything easier. I have to thank Gonzalo for all his confidence, ability to work and patience, but especially, his great humility, despite being the Messi of Compact Data Structures.

I must also thank Jose and Fari for so much help, collaboration, and motivational talks. Thank you to Ana and Mon for their help in teaching and carrying out this thesis. Thanks to Carmen for all the help with the forms, boarding passes etc. Thanks to all the members of the Database Lab, especially to my fellow adventurers: Alex, Daniil, Fernando and Tirso.

Thank you to all those people I have met throughout my travels abroad, who have made me feel at home: Ariel, Bonney, Diego Díaz, Diego Seco, Dominik, Giulio, Joanna and Pepe. Thanks to Travis Gagie and Nicola Prezza for everything I learned during the Summer School.

Many thanks to my friends, especially the old ones: Christian, Juan, Julio, Noelia, Rebeca and Tania; all the parties, games and walks, were essential for the development of this thesis.

Finally, fundamental was my family, especially my parents, my grandmother and my sister, who are the ones who see my bitterest face. Thank you for all the teachings, efforts, advice and laughter. I will never have enough time to thank you for everything you do for me.

Agradecimientos

¡Finalmente! Esta etapa de doctorado está llegando a su fin. Una etapa llena de lecciones, plazos, viajes, altibajos. Tengo que comenzar agradeciendo a mis directores de tesis, sin los cuales esto no hubiera sido posible: Nieves y Gonzalo. Nieves me apoyó desde el principio hasta el final de este largo viaje. Muchas gracias por toda la ayuda, confianza, consejos y por hacerme todo más fácil. Tengo que agradecer a Gonzalo por toda su confianza, capacidad de trabajo y paciencia, pero sobre todo, su gran humildad, a pesar de ser el Messi de las Compact Data Structures.

También debo agradecer a José y Fari por tanta ayuda, colaboración y charlas motivadoras. Gracias a Ana y Mon por su ayuda en la enseñanza y realización de esta tesis. Gracias a Carmen por toda la ayuda con los formularios, tarjetas de embarque, etc. Gracias a todos los miembros del Database Lab, especialmente a mis compañeros de aventuras: Alex, Daniil, Fernando y Tirso.

Gracias a todas aquellas personas que he conocido durante mis viajes al extranjero, que me han hecho sentir como en casa: Ariel, Bonney, Diego Díaz, Diego Seco, Dominik, Giulio, Joanna y Pepe. Gracias a Travis Gagie y Nicola Prezza por todo lo que aprendido durante la Summer School.

Muchas gracias a mis amigos, especialmente a los de siempre: Christian, Juan, Julio, Noelia, Rebeca y Tania; todas las fiestas, partidas y paseos fueron esenciales para el desarrollo de esta tesis.

Finalmente, fundamental fue mi familia, especialmente mis padres, mi abuela y mi hermana, quienes son los que ven mi cara más amarga. Gracias por todas las enseñanzas, esfuerzos, consejos y risas. Nunca tendré tiempo suficiente para agradeceros por todo lo que haceis por mí.

Agradecementos

Por fin! Esta etapa de doutorando chega ao seu fin. Unha etapa chea de leccións, prazos, viaxes, subidas e baixadas. Teño que comezar dándolle grazas aos meus directores de tese, sen os cales isto non sería posible: Nieves e Gonzalo. Nieves apoioume dende o principio até a fin deste longo camiño. Moitas grazas por toda a axuda, confianza, consellos e facerme todo máis fácil. A Gonzalo téñolle que agradecer toda a súa confianza, capacidade de traballo e paciencia, pero sobre todo, a súa gran humildade, a pesar de ser o Messi das Compact Data Structures.

Tamén debo darlle as grazas a Jose e Fari por tanta axuda, colaboración e charlas motivacionais. Agradecerlle a Ana e Mon a axuda prestada na docencia e realización desta tese. Grazas a Carmen por toda a axuda cos formularios, tarxetas de embarque etc. Grazas a todos os membros do Laboratorio de Bases de Datos, especialmente aos meus compañeiros de aventuras: Alex, Daniil, Fernando e Tirso.

Agradecerlle a toda aquela xente que me fun atopando ao longo das miñas viaxes no estranxeiro, os cales me fixeron sentir como na casa: Ariel, Bonney, Diego Díaz, Diego Seco, Dominik, Giulio, Joanna e Pepe. Agradecer a Travis Gagie e a Nicola Prezza todo o aprendido durante a Summer School.

Moitas grazas aos meus amigos, en especial, aos de sempre: Christian, Juan, Julio, Noelia, Rebeca e Tania; todas as festas, partidas e paseos, foron imprescindibles para o desenvolvemento desta tese.

Finalmente, fundamental foi a miña familia, especialmente meus pais, miña avoa e miña irmá, que son os que ven a miña cara máis amarga. Grazas por todas as ensinanzas, esforzos, consellos e risas. Nunca terei o suficiente tempo para agradecervos todo o que facedes por min.

Abstract

This thesis deals with the efficient representation and exploitation of trajectories of objects that move in space without any type of restriction (airplanes, birds, boats, etc.). Currently, this is a very relevant problem due to the proliferation of GPS devices, which makes it possible to collect a large number of trajectories. However, until now there is no efficient way to properly store and exploit them.

In this thesis, we propose eight structures that meet two fundamental objectives. First, they are capable of storing space-time data, describing the trajectories, in a reduced space, so that their exploitation takes advantage of the memory hierarchy.

Second, those structures allow exploiting the information by object queries, given an object, they retrieve the position or trajectory of that object along that time; or space-time range queries, given a region of space and a time interval, the objects that are within the region at that time are obtained. It should be noted that state-of-the-art solutions are only capable of efficiently answering one of the two types of queries.

All of these data structures have a common nexus, they all use two elements: snapshots and logs. Each snapshot works as a spatial index that periodically indexes the absolute position of each object or the Minimum Bounding Rectangle (MBR) of its trajectory. They serve to speed up the spatio-temporal range queries. We have implemented two types of snapshots: based on k^2-trees or R-trees.

With respect to the log, it represents the trajectory (sequence of movements) of each object. It is the main element of the structures, and facilitates the resolution of object and spatio-temporal range queries. Four strategies have been implemented to represent the log in a compressed form: ScdcCT, GraCT, ContaCT and RCT.

With the combination of these two elements we build eight different structures for the representation of trajectories. All of them have been implemented and evaluated experimentally, showing that they reduce the space required by traditional methods by up to two orders of magnitude. Furthermore, they are all competitive in solving object queries as well as spatial-temporal ones.

Resumen

Esta tesis aborda la representación y explotación eficiente de trayectorias de objetos que se mueven en el espacio sin ningún tipo de restricción (aviones, pájaros, barcos, etc.). En la actualidad, este es un problema muy relevante debido a la proliferación de dispositivos GPS, lo que permite coleccionar una gran cantidad de trayectorias. Sin embargo, hasta ahora no existe un modo eficiente para almacenarlas y explotarlas adecuadamente.

Esta tesis propone ocho estructuras que cumplen con dos objetivos fundamentales. En primer lugar, son capaces de almacenar en espacio reducido los datos espacio-temporales, que describen las trayectorias, de modo que su explotación saque partido a la jerarquía de memoria.

En segundo lugar, las estructuras permiten explotar la información realizando consultas sobre objetos, dado el objeto se calcula su posición o trayectoria durante un intervalo de tiempo; o consultas de rango espacio-temporal, dada una región del espacio y un intervalo de tiempo se obtienen los objetos que estaban dentro de la región en ese tiempo. Hay que destacar que las soluciones del estado del arte solo son capaces de responder eficientemente uno de los dos tipos de consultas.

Todas estas estructuras de datos tienen un nexo común, todas ellas usan dos elementos: snapshots y logs. Cada snapshot funciona como un índice espacial que periódicamente indexa la posición absoluta de cada objeto o el Minimum Bounding Rectangle (MBR) de su trayectoria. Sirven para agilizar las consultas de rango espacio-temporal. Hemos implementado dos tipos de snapshot: basadas en k^2-trees o en R-trees.

Con respecto al log, éste representa la trayectoria (secuencia de movimientos) de cada objeto. Es el principal elemento de nuestras estructuras, y facilita la resolución de consultas de objeto y de rango espacio-temporal. Se han implementado cuatro estrategias para representar el log de forma comprimida: ScdcCT, GraCT, ContaCT y RCT.

Con la combinación de estos dos elementos construimos ocho estructuras diferentes para la representación de trayectorias. Todas ellas han sido implementadas y evaluadas experimentalmente, donde reducen hasta dos órdenes de magnitud el espacio que requieren los métodos tradicionales. Además, todas ellas son competitivas

resolviendo tanto consultas de objeto como de rango espacio-temporal.

Resumo

Esta tese trata sobre a representación e explotación eficiente de traxectorias de obxectos que se moven no espazo sen ningún tipo de restrición (avións, paxaros, buques, etc.). Na actualidade, este é un problema moi relevante debido á proliferación de dispositivos GPS, o que fai posible a recollida dun gran número de traxectorias. Non obstante, ata o de agora non existe un xeito eficiente de almacenalos e explotalos.

Esta tese propón oito estruturas que cumpren dous obxectivos fundamentais. En primeiro lugar, son capaces de almacenar datos espazo-temporais, que describen as traxectorias, nun espazo reducido, de xeito que a súa explotación aproveita a xerarquía da memoria.

En segundo lugar, as estruturas permiten explotar a información realizando consultas de obxectos, dado o obxecto calcúlase a súa posición ou traxectoria nun período de tempo; ou consultas de rango espazo-temporal, dada unha rexión de espazo e un intervalo de tempo, obténense os obxectos que estaban dentro da rexión nese momento. Cómpre salientar que as solucións do estado do arte só son capaces de responder eficientemente a un dos dous tipos de consultas.

Todas estas estruturas de datos teñen unha ligazón común, empregan dous elementos: snapshots e logs. Cada snapshot funciona como un índice espacial que indexa periodicamente a posición absoluta de cada obxecto ou o Minimum Bounding Rectangle (MBR) da súa traxectoria. Serven para acelerar as consultas de rango espazo-temporal. Implementamos dous tipos de snapshot: baseadas en k^2-trees ou en R-trees.

Con respecto ao log, este representa a traxectoria (secuencia de movementos) de cada obxecto. É o principal elemento das nosas estruturas, e facilita a resolución de consultas sobre obxectos e de rango espacio-temporal. Implementáronse catro estratexias para representar o log nunha forma comprimida: ScdcCT, GraCT, ContaCT e RCT.

Coa combinación destes dous elementos construímos oito estruturas diferentes para a representación de traxectorias. Todas elas foron implementadas e avaliadas experimentalmente, onde reducen ata dúas ordes de magnitude o espazo requirido polos métodos tradicionais. Ademais, todas elas son competitivas para resolver tanto consultas de obxectos como espazo-temporais.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter summarizes the contents of this thesis, and we introduce the motivation of the structures designed for representing moving object trajectories. Section 1.1 gives the motivation and a brief introduction to state of the art. Section 1.2 introduces our method to represent trajectories and the common elements between our structures. Besides, it briefly explains the queries that we are interested in solving and their classification in two types. Finally, Section 1.3 presents the organization of this thesis in the different chapters.

1.1 Motivation

More than two decades after it emerged, the field of moving object databases is still an active area of research. During the last years, the number of devices that track information about the position of different kinds of objects has increased considerably. For example, nowadays, we can collect a large amount of data from the GPS positions of large sets of cars, ships, planes, smartphones, and wearable devices. Consequently, in the last years, the size of the datasets of moving object trajectories has sharply increased.

Those datasets open up a wealth of new possibilities to obtain knowledge from moving object trajectories, which can be useful in different types of applications like traffic management, analyzing human movement, tracking animal behavior, security and surveillance, military battlefield, and others [GLW08]. Due to the sharply increasing sizes of these datasets, the treatment and storing of moving object data becomes a challenge.

A trajectory, which does not consider a road network, is a path followed by a moving object through space as a function of time. Due to storage requirements and the limitations of the devices used to acquire the object positions, the continuous movement of an object is usually approximated with discrete samples of spatio-temporal location points: the more samples taken, the more accurate the trajectory. However, high sampling rates result in large amounts of data, which increases storage, transmission, and processing needs. Even when storage, network, and processing capacity grows rapidly, the collected data grows even faster, and thus it is necessary to aim for reduced trajectory representations [ZZ11].

One traditional way to store trajectories is to use some disk storage, such as conventional record-based files. However, accessing to disk is a costly operation that difficulties querying and handling the data in an efficient way. The performance of accessing the data can be improved with one or more indexes to speed up queries over the stored data. This approach holds the bulk of the data on disk, while the index structures reside in main memory, at least partially. Other methods combine the data and the index in a single structure, though part still resides on disk. Even then, the handling and querying the data is not efficient. Thus new techniques for storage and efficient processing are necessary [ZZ11]. For this reason this thesis proposes new data structures that compress the trajectory representation and avoids access to disk.

With the increasing gap in the access time of main memory versus disk, compressing the trajectories in order to query them in main memory is an attractive option. Traditional methods for compressing trajectories include line generalization (or simplification) techniques, keeping only some of the trajectory points, and discarding the rest. This approach results in some loss of information on the real trajectory. A lossless strategy to obtain compression is the use of *delta compression*, where each new position is stored as the difference with the previous one. This idea exploits the fact that consecutive positions are expected to be closer to each other, and that smaller numbers can be stored using fewer bits. Extracting a whole trajectory with this arrangement is easy. Efficiently accessing the position of an object at a given time, instead, requires sampling some absolute positions at regular time intervals, which introduces a space/time tradeoff. Some recent proposals following this trend [CMWM10, WZX$^+$14] build on delta compression, coupled with an encoding that favors small numbers. The optimal codes for delta compression can be obtained with a statistical encoder that exploits frequency bias (typically, smaller numbers are more frequent).

Therefore, the underlying queries that a system managing collections of trajectories should answer are: recovering the position of an object at a specific time instant and recovering information about a part of its path during some time (*object queries*). They are useful to obtain the trajectory of a taxi along the time or its position at a desired time instant. However, some applications need to support other kind of queries. The most classical queries are *range spatio-temporal queries*, which return those objects that hold some spatio-temporal constraints (e.g., objects within a region during a period o time, objects closest to a point at a specific time instant). Following with the example of taxis, they are useful to locate the closest taxi to a position or identify the taxis within a spatial area.

Consequently, another issue is how to index the trajectory data to answer range spatio-temporal queries, which are not just retrieving a whole trajectory or finding the position of an object at a given time instant. Many indexes have been proposed since the 90's to handle a rich set of queries on trajectory data. Most indexes were modifications of the R-tree [Gut84], which augmented another dimension to deal with the time. None of those works, however, compresses the data. Instead, they are designed to work on disk, which is much slower than the main memory.

A new family of data structures called compact data structures combines, in a single representation, a compressed representation of the data with the mechanisms that provide direct access to any given datum, or even complex queries [Nav16]. These structures keep the data compressed all the time, without ever needing to decompress it. In addition to the obvious space savings, compact data structures allow more massive datasets to be managed in main memory, much faster processing of datasets that can fit entirely in main memory thanks to compression, and improved performance of distributed deployments.

In many cases, the compact data structure is coupled with indexes that speed up the retrieval of information, enabling query times comparable to, and often better than, traditional setups. The mechanism by which data is simultaneously compressed and indexed is commonly known as *self-indexing* and is particularly useful in situations where storage space is a problem.

For this reasons, this thesis aims to study and design new compact data structures and algorithms to represent collections of trajectories of objects that are moving in the space without any constraint and without assuming the existence of an underlying network. Our methods show excellent performance in space/time in comparison with classical spatio-temporal indexes.

1.2 Contributions

1.2.1 Compact representation of trajectories

This thesis focuses on a central problem, storing trajectories of objects moving freely in the space in a compact representation, and efficiently retrieving and querying their data. Therefore, our contribution consists of the design, analysis, implementation, and experimental evaluation of different compact data structures. All of them have particular properties, thus it makes necessary to design different algorithms to solve the proposed query types (object and range).

All of our structures are composed of two elements: *snapshots* and *logs*. The snapshots store spatial information of the objects at regular time instants and, the log stores the relative movements of each object, where each relative movement corresponds with the displacement of the object from one time instant to the next one.

We propose different data structures for snapshots and logs. For each combination of those elements, we implement the corresponding algorithms. In the case of the snapshots we propose two structures:

- *Snapshots based on k^2-trees*, which represents the areas where there are objects by using a k^2-tree. With the help of an additional array, we can discern the objects within each area. The compression of that kind of snapshot exploits the clustering and empty areas of objects.
- *Snapshots based on R-trees*, each snapshot uses a compressed version of an R-tree, a classical spatial index. It stores, for each individual object, the rectangular area that contains the trajectory of the object along a specific interval of time.

Concerning the log, we design four different techniques:

- *ScdcCT* exploits the fact that short movements are more frequent than large displacements by compressing the log with (s, c)-Dense Codes [BFNP07], that has low redundancy over the zero-order empirical entropy of the sequence.
- *GraCT* considers the log as a sequence of integers, and it is compressed with a grammar-based compressor called Re-Pair [LM00] that exploits the repetitiveness of patterns between all the objects.
- *ContaCT* is based on a structure for *partial-sums*, and its primary goal is to compute the position at a time instant in constant time at the cost of using additional space with respect to the previous log structures.
- *RCT* was designed for the compression of highly repetitive trajectories and tries to represent all of them with relative compression, that is, all the trajectories are composed of parts from an artificial trajectory, which contains the most common movements of the objects.

As we developed two data structures for snapshots and four for logs, we finally have eight different techniques. For each of them, we evaluate its compression effectiveness. All our structures obtain a compression ratio of around 5%–25%, with respect to to the minimum binary representation of the trajectory data, as we describe in Chapter 9.

1.2.2 Solving queries efficiently

The eight structures presented in this thesis can solve different queries efficiently. As we explain above, we can distinguish two kinds of queries: *object queries* and *spatio-temporal range queries*. The first type of queries is focused on retrieving information about the location or trajectory of a specific object. Instead, the second type of queries compute the objects within a region during an interval of time. Below, we present more details about the implemented queries.

1.2.2.1 Object queries

There are three different ways to obtain information about the location or trajectory of an object:

- **Search Object**: given an object identifier *id* and a time instant t_q, this query computes the position of that object at the queried time instant t_q.

- **Search Trajectory**: like the previous query, it calculates the consecutive positions of an object during an interval of time $[t_s, t_e]$. That is, it produces the sequence of positions traversed by the object during the queried interval of time.

- **Minimum Bounding Rectangle (MBR)**: given a range of time $[t_s, t_e]$ and an object, it computes the smallest rectangle that contains the trajectory of the object from t_s to t_e.

1.2.2.2 Spatio-temporal range queries

The result of this type of queries is a list of objects. Unlike in object queries, where the object is always a parameter of the query. The first two spatio-temporal range queries obtain the objects within a region, and the last one identifies those that are the closest ones with respect to a point.

- **Time Slice**: this query returns those objects within a given region r_q at a given time instant t_q.

- **Time Interval**: it is an extension of *time slice* that expands t_q to an interval of time $[t_s, t_e]$. Hence, it returns those objects within r_q in any time instant belonging to $[t_s, t_e]$.

- **K-Nearest Neighbors**: given a point p_q in the space and a time instant t_q, it returns the k closest objects to p_q at t_q.

1.3 Structure of the Thesis

The structure of the thesis is as follows. First, in Chapter 2, we present some basic concepts about data compression and compact data structures. In Chapter 3, we show the previous work in the field of moving objects. After that we explain our contributions with the following chapters:

- Chapter 4 introduces the general idea of our contributions, that is, the method common to all of our structures and its elements: snapshots and logs.

- Chapter 5 defines the components of our structures: snapshots and logs; and a set of operations that can be solved in each one of those components. Several algorithms to solve queries about trajectories, spatio-temporal queries, and retrieving the closest objects to a point are described.
- Chapter 6 presents the two different kinds of snapshots and the algorithms to retrieve the information stored within them, which are the basis to solve some queries.
- Chapter 7 describes the four different structures for the representation. This chapter also presents the algorithms that the logs need to support the queries.
- Chapter 8 explains how to represent real trajectories by using our structures. It also describes the necessary modifications of our structures in order to represent the missed information that suffer real datasets.
- Chapter 9 presents the experimental evaluation of our eight structures over different datasets, varying different parameters. Besides, their scalability is studied, and they are compared with a classical spatio-temporal index.
- Chapter 10 discusses the conclusions and some future works for our contribution.

Chapter 2

Basic Concepts

This thesis proposes new compact data structures for the representation of extensive collections of trajectories of objects that are moving freely in the space. In this chapter, we introduce concepts of different fields for a better understanding of our contributions. In Section 2.1, we present several basic notions of information theory and data compression. Section 2.2 introduces several compact data structures used in this thesis.

2.1 Information Theory and Data Compression

2.1.1 Basic concepts on Information Theory

Information Theory is a field of Computer Science that focuses on studying the quantification of information to transmit messages through communication channels efficiently. The bases of Information Theory were proposed by Shannon [Sha48], providing many useful concepts. In that work, one of the most relevant ideas for this thesis is how to compute the minimum amount of space required to encode a message. This allows us to determine the repetitiveness of the message and discern which techniques can be applied over these data.

Assume that we have an infinite source of information that emits symbols $x \in \mathcal{X}$ with a probability $p(x)$. This can be mathematically modeled as a discrete random variable X that takes values in $\mathcal{X}$ with probability mass function $p(x) = Pr\{X = x\}$. The amount of information associated with an outcome $x \in \mathcal{X}$ is defined by the formula $I_X(x) = \lg \frac{1}{p(x)}$.[1] In other words, an outcome offers more information than one with a higher probability. For example, whether $p(x) = 1$ there is no information, because the source is emitting x continuously as it is expected by its probability.

[1] Note that we denote $\log_2$ as lg

Related to this concept of information is the *entropy*. Shannon[Sha48] defined it as a function $\mathcal{H}(X)$ or $\mathcal{H}$ that measures the amount of information or uncertainty that is expected from a random variable X, and can be computed as:

$$\mathcal{H}(X) = \sum_{x \in \mathcal{X}} p(x) \lg \frac{1}{p(x)} \tag{2.1}$$

Whether the information is not proportioned by an infinite source, Shannon defines a notion of *entropy* for finite sequences called *zero-order empirical entropy*. Let us define a sequence $S[1,n]$ over an alphabet $\Sigma = [1 \ldots \sigma]$, where each symbol s appears n_s times in S. The *zero-order empirical entropy* of S is computed as:

$$\mathcal{H}_0(S) = \sum_{1 \leq s \leq \sigma} \frac{n_s}{n} \lg \frac{n}{n_s}. \tag{2.2}$$

It measures the uncertainty about S by considering only the probability of occurrence of each symbol. In most of the cases, encoding S with $\mathcal{H}_0$ bits per symbol is good enough.

An encoding function C for a random variable X maps every symbol in $\mathcal{X}$ to $\mathcal{D}^*$, where $\mathcal{D}$ is an alphabet of cardinality D and $\mathcal{D}^*$ is the set of finite-length strings composed by symbols from $\mathcal{D}$. Therefore, any symbol $x \in \mathcal{X}$ can be encoded by the *encoding* or *code* C and the result *codeword* is $C(x)$, which is composed by *target symbols* from the *target alphabet* $\mathcal{D}$. We can distinguish two types of encoding depending on the lengths of the *codewords*: *fixed-length* and *variable-length*. In the first case, every codeword has the same length: $|C(x)| = |C(y)|\ \forall\ x, y \in X$, where $|x|$ denotes the length of x. Instead, in variable-length codes, each symbol can be encoded with different lengths. Notice that two different symbols $x \neq y\ x, y \in X$ are univocally decodable when $C(x) \neq C(y)$, otherwise decoding a codeword could be ambiguous.

A direct extension of C is C^*, which transforms a finite string of symbols *message* into a finite string of target symbols. The encoded string can be computed by appending the individual codewords of each source symbol: $C^*(x_1, x_2, \ldots, x_n) = C(x_1)C(x_2)\ldots C(x_n)$. The original message can be recovered by decoding each codeword. However, for some *encoding schemes*, detecting each codeword's end can be difficult and need to read a large portion of the message. Those encodings which allow decoding a codeword $C(x)$ after reading its last bit are known as *instantaneous* or *prefix-free* encodings. Formally, an encoding scheme is *prefix-free* if there is no code $C(x)$ that is a prefix of other code $C(y)$. It is important to notice that if C is *prefix-free*, $C^*(x_1, x_2, \ldots, x_n)$ is *univocally decodable*. Also, for all *univocally decodable* encoding, we can find a *prefix-free* code with the same average length, hence both occupies the same but the *prefix-free* is easier to decode. Those codes which are prefix-free and get the minimum average length are known as *optimal* codes. A lower bound on average length can be computed by the entropy, given the source symbols and their probabilities.

High-order models take into account a context of a fixed-size k, which is the k preceding values of a symbol x. Those models are known as *k-order models*, and they measure the information of a symbol by considering the k preceding symbols. For example, in natural language, if one knows the previous words, it is easier to guess the next word. Based on this idea, Shannon proposes the *k-order empirical entropy*, as:

$$\mathcal{H}_k(S) = \sum_{C=s_1\ldots s_k} \frac{|S_c|}{n}\mathcal{H}_0(S_c). \tag{2.3}$$

being S_C a string composed by joining the symbols that follows each occurrence of the context $C = s_1 \ldots s_k$ in S.

2.1.2 Data Compression: basic concepts

Facing the necessity to represent large datasets in less space, emerges the *data compression*, which tries to improve their manipulation, storage, and transmission.

2.1.2.1 Classification of compression techniques

Compression techniques transform an input message into a compressed version by a phase of *encoding*. The original message can be recovered from the encoded version by a stage of *decoding*. Depending on the result of decoding the encoded message, we can classify compression techniques into two categories.

- **Lossy compression techniques**, after performing the encoding process, the encoded message is not able to retrieve the original message. In that phase, some information of the input message is lost, which implies that the decoded result will be very similar to the original message but not the same. Lossy compression is advantageous in areas where an approximate version of the original message is enough. For example, there are widely used to compress video or images, where human eyes cannot detect those small differences.

- **Lossless compression techniques**, from the encoded message, we can retrieve the original one. Some fields that do not allow the loss of any information. Hence lossy compression techniques cannot be used. For example, in text compression, these techniques are largely used, because if the message undergoes any modification may become meaningless. In this thesis, we focus only on this kind of techniques.

Another way of categorizing the compression techniques is according to how the encoding process is realized. We can distinguish two families:

- **Statistical techniques** assign codewords to the source symbols according to their frequency. Shorter codewords will correspond to the more frequent source symbols. Some well-known statistical techniques are Huffman codes [Huf52], arithmetics codes [Abr63, WNC87, MNW95], or the family of Dense Codes [dMNZBY00, BFNP05, BFNP10].

- **Dictionary techniques**, by considering the input as a string of source symbols, these techniques create a dictionary of substrings and replace their appearances in the source file with pointers to their corresponding entry in the dictionary. Those techniques reduce the space by representing several symbols by one codeword. The *Lempel-Ziv* family[ZL77, ZL78, KPZ10] are the most famous dictionary techniques. By using a sliding window of fixed-size, they replace substrings by pointers to previous occurrences of identical substrings. Also, there is a more structured way of compression based on dictionaries, namely grammar compression, which is more suitable for random access, pattern matching, etc. Those techniques compress a sequence S into a single sequence C and a context-free grammar G. With C and G, the original sequence S can be obtained without losing information. One of the most well-known grammar-based compressors is Re-Pair [LM00].

2.1.3 Encoding Integer Numbers

We can classify the different methods for encoding integer numbers into two groups: small and large integers. The most well-known techniques for representing small integers are unary-codes, λ-codes, and δ-codes:

- **Unary codes** are a variable-length encoding for extremely small integers. Basically, the encoding represents an integer x as 0-bits repeated $x-1$ times followed by a 1-bit, that is, $unary(x) = 0^{x-1}1$. Therefore, $|unary(x)| = x$, and as a consequence the number 0 cannot be represented.

- **Gamma codes** (γ) are only convenient when x is small. γ-code encodes the length of x in unary code followed by the number x without its most significant bit. That is, $\gamma(x) = unary(|x|)[x]_{|x|-1}$, where $[x]_{|x|-1}$ is the binary representation without the highest bit.

- **Delta-codes** (δ) are useful when x is too large for being represented with γ-codes. It is very similar to γ-codes, and it is defined as: $\delta(x) = \gamma(|x|)[x]_{|x|-1}$. That is, the length of x is encoded with γ-codes instead of unary codes.

On the other hand, when the integer is too large, there are some more efficient techniques than δ-codes. These techniques aim to improve space efficiency and fast decoding. An example of these techniques is VByte-codes [WZ99].

The aim of VByte-codes is not only to be space-efficient but also to obtain fast decoding. In this case, it speeds up the decoding phase by obtaining a byte-aligned variable-length solution. It means that each value x is split into byte-length chunks. Therefore, VByte-codes divide x into chunks of 7 bits. Each value is stored in the lowest bits of a chunk. The highest bit of the byte specifies with 1-bit or 0-bit, when the byte stores the binary representation of the last chunk or does not, respectively. That is, $VByte(x) = b_1b_2 \dots b_k$ where $k = \lceil |x|/7 \rceil$ and b_i stores the bits of x at positions $[7 \times i, \dots 7 \times (i+1) - 1]$ padding to the left with 0-bit or 1-bit if $i < k$ or $i = k$, respectively. This code can be extended to a string of integers $X = x_1x_2 \dots x_n$ as $VByte^*(X) = VByte(x_1)VByte(x_2) \dots VByte(x_n)$.

2.1.4 Statistical compressors

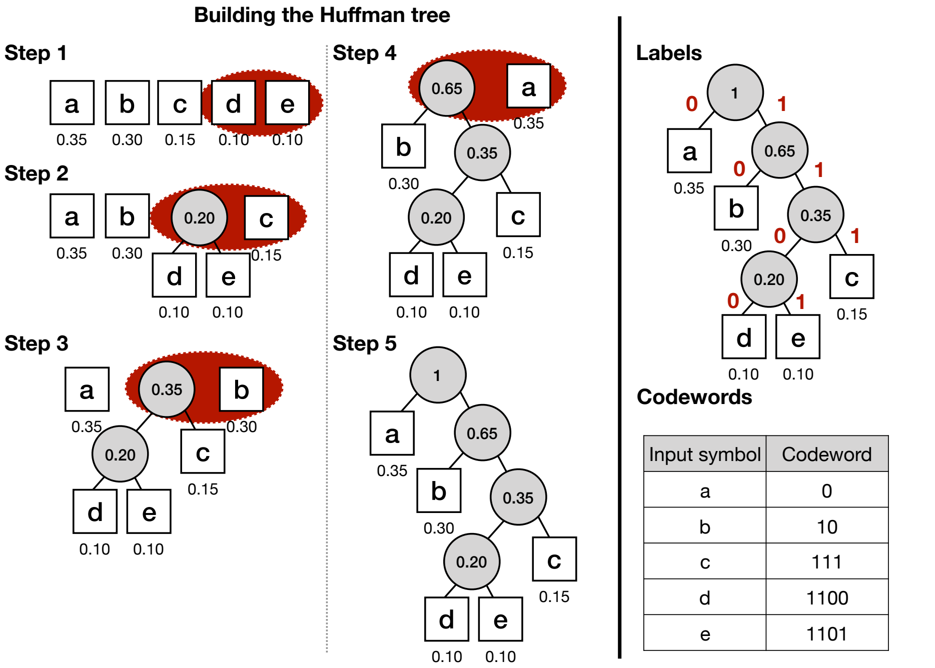
2.1.4.1 Huffman codes

Huffman proposed an algorithm [Huf52] that builds a prefix-free code of minimum average length. These codes are known as *Huffman codes*. Its main idea is to assign codewords whose length is proportional to each symbol's frequency by associating short codewords to symbols with high probability and large codewords to those with less probability. With this approach, the length of the output stream of bits for a random variable X is between $n\mathcal{H}(S)$ and $n\mathcal{H}(S) + 1$, that is, it requires at most one extra bit per symbol with respect to the entropy.

To obtain the Huffman codes, during the encoding, the algorithm builds a tree that contains prefix-free codes for each symbol. Classical Huffman tree is a full binary tree, where each node can contain zero or two child nodes, and each leaf corresponds to a codeword. Every node is labeled with a weight that represents the sum of the probabilities of its children leaves. Their position in the tree depends on that label; a higher level node is heavier than one node at a lower level.

The Huffman algorithm starts with a list containing n leaf nodes, one per source symbol, whose labels correspond to the probability of the symbol. That list is sorted by probability. The algorithm takes the two nodes with the smallest probability and creates their parent node. The parent stores the sum of the probabilities of its children. Then, the two smallest nodes are removed from the list, and their parent is added. The process is repeated until there is just one node in the list. This last node is the root of the Huffman tree, and thus, its label is 1. Whether the nodes are sorted by probability, building a Huffman tree takes $O(n)$ time [MNW95]. After building the tree, each codeword is obtained with a top down traversal from the root until the leaf that contains the encoded symbol. The binary representation of the codeword depends on the path from the root to the leaf. Each branch on the left corresponds with a 0-bit, otherwise adds a 1-bit.

For example, in Figure 2.1, the left half shows the construction of the Classical Huffman tree for an alphabet $\{a, b, c, d, e\}$, and the right half illustrates the labeling and assignment of the codewords for each symbol. During the construction of the tree, the list of symbols is kept sorted by their probabilities. In Step 3, we could choose to join a and b in a subtree with frequency 0.65. However, we have chosen b and its right subtree. Notice that, depending on the selected alternative, the Huffman code changes, which is not a rare case. Usually, several Huffman trees can



Input symbol	Codeword
a	0
b	10
c	111
d	1100
e	1101

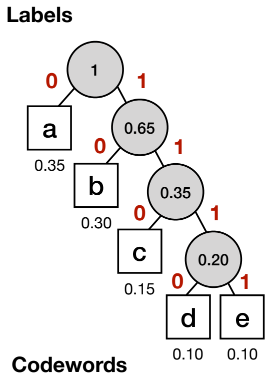
Figure 2.1: Example of Huffman code.

be built over the same sequence. Consequently, to the codewords of a message, the compressed file needs to include information about the alphabet and the shape of the Huffman tree; otherwise, the message could not be decompressed. During the decompression stage, the algorithm reads each bit and traverses the Huffman tree until reaching a leaf. Whether the algorithm is in an internal node and the read bit is a 0-bit, the algorithm follows the left branch, otherwise, the right branch. When the traversal reaches a leaf, a source symbol is obtained, and it is output. Then, the traversal starts again from the root.

2.1.4.2 Canonical Huffman

Given a set of source symbols and their probabilities, different Huffman trees can be built, and thus different codes can be generated. Although Huffman's algorithm computes the codewords for each source symbol, only their lengths are relevant. This means that once those lengths are known, codewords can be assigned in several ways. Among all of them, the canonical Huffman code [SK64] is the most used since its shape requires less space to be stored.

The canonical Huffman tree is built from left to right in increasing order. There



Codewords

Input symbol	Codeword
a	0
b	10
c	110
d	1110
e	1111

Figure 2.2: Example of Canonical Huffman.

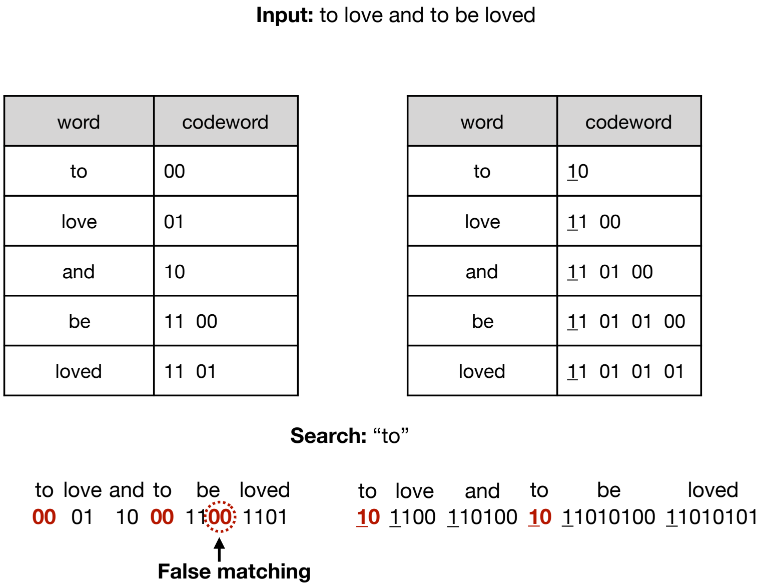
is at least one leaf per level, and they are placed in the first position available from left to right. The following properties hold:

- Codewords are assigned in increasing length order with the lengths of Huffman's algorithm.
- Codewords of a given length are consecutive binary numbers.
- The first codeword c_l of length l is related to the last codeword of length $l - 1$ by $c_l = 2(c_l - 1 + 1)$.

The information about the shape of the canonical Huffman tree can be compactly represented by storing only the lengths of the codewords. Therefore, the compressed file requires $O(h)$ integers, where h is the height, to represent the shape of the tree. Additional space is necessary to store the source alphabet, sorted by frequency. Figure 2.2 shows the Canonical Huffman codes for the example of Figure 2.1.

2.1.4.3 Plain Huffman and Tagged Huffman Codes

When using Huffman, if the source alphabet is composed of characters and the target alphabet are bits, the compression ratio and the compression/decompression speed



Input: to love and to be loved

word	codeword
to	00
love	01
and	10
be	11 00
loved	11 01

word	codeword
to	10
love	11 00
and	11 01 00
be	11 01 01 00
loved	11 01 01 01

Search: “to”

Figure 2.3: Comparison of Plain and Tagged Huffman Codes. For legibility we assume each byte is composed by two bits.

are poor. *Plain Huffman* and *Tagged Huffman* are the word-based byte-oriented variants of the Huffman code [dMNZBY00]. By using bytes instead of bits as target alphabet, since it avoids bit manipulations, the algorithm provides faster decompression but pays more space with respect to a bit-oriented approach. Another feature of these variants is that they allow searching for a pattern directly in the compressed text faster than searching the uncompressed text. Plain Huffman Code obtains better compression ratios than Tagged Huffman, but the Plain approach does not provide random access, that is, Tagged Huffman can decompress any portion of the text and start a search at any position [BM77, NR02]. If we use Plain Huffman and start a search in a position different from the beginning of the text, a false match can occur, as shown in Figure 2.3. However, Tagged Huffman Codes avoid that problem by marking the first byte of a codeword: the first bit of each byte is a flag, set to 1 when it corresponds with the first bit of the codeword. The remaining 7 bits are used for the Huffman code. Since only 7 bits are dedicated to coding, Tagged Huffman needs more space than Plain Huffman to encode a given message. However searches are faster, and it also allows random decompression.

2.1.4.4 End-Tagged Dense Code and (s,c)-Dense Code

In state of the art, some proposals improve the performance of Tagged Huffman. The first work proposed was End-Tagged Dense Code [BINP03, BFNP07] that achieves similar compression ratios to Plain Huffman but keeps the performance capabilities of Tagged Huffman.

The main difference between ETDC and Tagged Huffman is that, instead of marking the first byte of the codeword, it marks the codeword's last byte. Hence, ETDC reserves the first bit of each byte as a flag that indicates whether the byte is the last one of its codeword. Although the difference is quite simple, there is a positive implication: the code is a prefix code regardless of the content of the other 7 bits. Since it does not need to use Huffman code for the remaining bits, ETDC can code all possible combinations of those 7 bits, thus producing a dense encoding. That is the key to obtain compression ratios close to Plain Huffman and improve those obtained by Tagged Huffman.

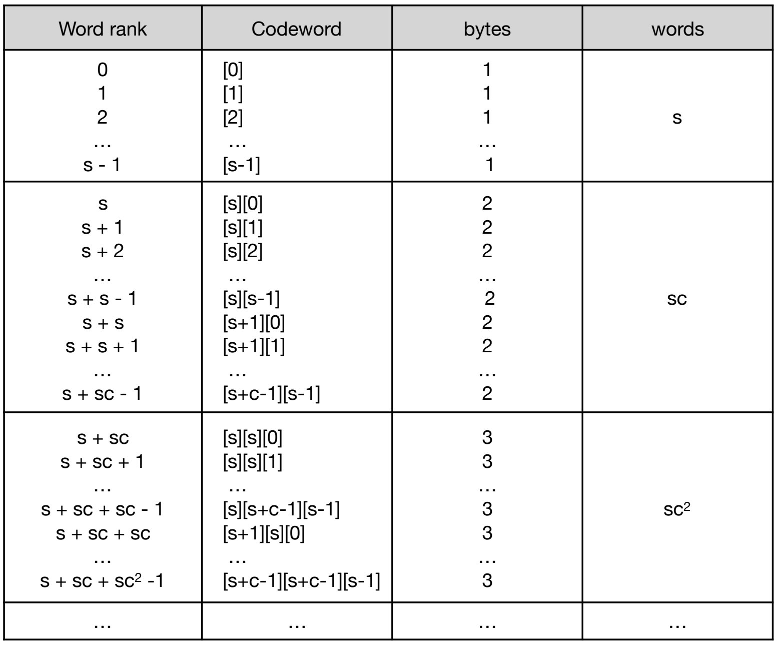
Assuming our target symbols require b bits (b = 8 in the byte-oriented version) and given source symbols sorted by decreasing frequencies, each codeword is a sequence of target symbols representing digits in the range $[0, 2^{b-1} - 1]$ except the last symbol whose value is in the range $[2^{b-1}, 2^b - 1]$. The process of assigning these codewords can be run sequentially, making the computation of the codewords simpler and faster than Huffman. It is important to notice that the codewords are assigned depending on the rank in the sorted vocabulary. Therefore, the decompressor only needs the non-decreasingly sorted vocabulary to obtain the original message.

(s,c)-Dense Code is a generalization of ETDC. By using codewords with $b = 8$, ETDC uses the values in the range $[0, 127]$ for those bytes that are not the end of a codeword, called continuers (c), and the values in $[128, 255]$ for the last symbol of the codewords, called stoppers (s). Notice that the number of stoppers and continuers are identical, this proportion could not be optimal for a given word frequency distribution. In (s,c)-Dense Code, any $s + c = 2^b$ can be used. Thus the values in $[0, s - 1]$ are the stoppers and those bytes in $[s, 2^b - 1]$ are used as continuers. The assignment of the codewords and their distribution in bytes are shown in Figure 2.4. For a given word frequency distribution, the optimal s and c values can be computed [BFNP07] to maximize compression ratios. Given a sorted word vocabulary in decreasing frequency, we can describe the encoding process as follows:

- One-byte codewords $[0, s - 1]$ are given to the first s words in the vocabulary.

- Two-byte codewords are assigned to the words in the sorted vocabulary in the range $[s, s + sc - 1]$. The first byte has a continuer value $[s, s + c - 1]$ and the last a stopper value in the range $[0, s - 1]$.

- By the function $\mathcal{F}(i) = \sum_{j=0}^{i} sc^{j-1}$ for any $i > 0$ and assuming $\mathcal{F}(0) = 0$. Any k-byte codeword is assigned to the range of the vocabulary $[\mathcal{F}(k-1), \mathcal{F}(k) - 1]$,



Word rank	Codeword	bytes	words
0	[0]	1	s
1	[1]	1	
2	[2]	1	
...	...	...	
s - 1	[s-1]	1	
s	[s][0]	2	sc
s + 1	[s][1]	2	
s + 2	[s][2]	2	
...	...	...	
s + s - 1	[s][s-1]	2	
s + s	[s+1][0]	2	
s + s + 1	[s+1][1]	2	
...	...	...	
s + sc - 1	[s+c-1][s-1]	2	
s + sc	[s][s][0]	3	sc^2
s + sc + 1	[s][s][1]	3	
...	...	...	
s + sc + sc - 1	[s][s+c-1][s-1]	3	
s + sc + sc	[s+1][s][0]	3	
...	...	...	
s + sc + sc^2 -1	[s+c-1][s+c-1][s-1]	3	
...	...	...	...

Figure 2.4: Distribution of (s,c)-Dense Code words.

by using $k-1$ continuers and one stopper.

In that work, the authors propose an algorithm to encode and decode a word, given its position in the sorted vocabulary. For instance, given the i-th ranked word $x = i - \frac{sc^{k-1}-s}{c-1}$, the first $k-1$ values of the codeword are the representation of number $\lfloor x/s \rfloor$ in base c, adding then s to each digit, and the last digit is $x \mod s$.

2.1.5 Dictionary-based compressors

2.1.5.1 Lempel-Ziv family

The Lempel-Ziv family includes the most well-known dictionary-based techniques. Compressors as *p7zip*, *gzip* and *compressor* are implementations based on variants of this family. All of them are derived from the first basic methods: *LZ77* and *LZ78*.

LZ77. LZ77 [ZL77] was the first proposed method in the Lempel-Ziv family. The main idea of LZ77 is to build a dictionary on a sequence S of an alphabet σ from the previously processed substring. For this purpose, the LZ77 has a *fixed-size sliding window* holding the m last processed symbols. The algorithm traverses the sequence

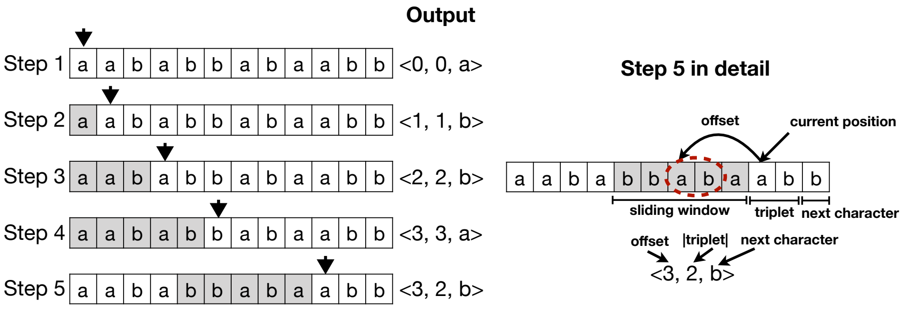

Output
Step 1 a a b a b b a b a a b b <0, 0, a>
Step 2 a a b a b b a b a a b b <1, 1, b>
Step 3 a a b a b b a b a a b b <2, 2, b>
Step 4 a a b a b b a b a a b b <3, 3, a>
Step 5 a a b a b b a b a a b b <3, 2, b>
Step 5 in detail
offset
current position
a a b a b b a b a a b b
sliding window
triplet
next character
offset
|triplet|
next character
<3, 2, b>

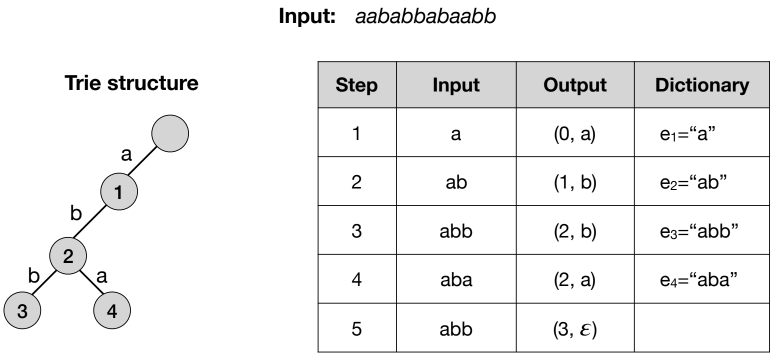
Figure 2.5: Example of LZ77.

and starts with an empty window. In each step, the algorithm looks for the maximal subsequence contained in the window that matches with the next input symbols. Note that the next symbol starts one position after the end of the window. Assuming that the matched subsequence is $s = s_1 s_2 \dots s_l$ and the following symbol is c, LZ77 encodes that substring as a triple $\langle p, l, c \rangle$, where the p value denotes the position of the occurrence of s in the window as a backward offset, and l is the length of s. Once the triplet is computed, the window moves $l + 1$ positions forward. Whether there is no match, that is the subsequence is empty $s = \mathcal{E}$, the encoded triplet is $\langle 0, 0, c \rangle$ and the sliding window moves one position forward. Figure 2.5 shows an example of LZ77 compression for *aababbabaabb*. Notice that the fixed-size sliding window is colored in gray.

During the decompression, the window keeps the last decoded symbols. Hence for a given triplet $\langle p, l, c \rangle$, the decoder only needs to copy the l symbols starting at position p before the last decoded symbol, and append to that sequence the symbol c. As a consequence, decompression turns out very fast.

The compression of LZ77 depends on the size of the window. The greater is that window, the higher the probability to encode larger substrings. Since the bits needed to represent p grows as the size of the window increases, in most of the implementations the window size is set to $4,096$ bytes. Usually, a triplet for compressing text can be encoded in 8 bytes: 12 bits for p, 4 bits for l, and 8 bits for the character c. Furthermore, the minimum size of the window must be considered, in order to avoid cases where the triplet occupies more that the substring.

There are other variants of this family based on LZ77, for instance, LZMA (Lempel-Ziv-Markov Chain Algorithm) is one of them. Usually, it builds a dictionary of size 1GB, although it can be limited up to 4GB. As a consequence of this huge dictionary, implementations of LZMA like *p7zip* can obtain better compression ratios than those based on LZ77 like *gzip*. Instead, compression and decompression require more memory and time.



Input: *aababbabaabb*

Trie structure

Step	Input	Output	Dictionary
1	a	(0, a)	e_1="a"
2	ab	(1, b)	e_2="ab"
3	abb	(2, b)	e_3="abb"
4	aba	(2, a)	e_4="aba"
5	abb	(3, ε)	

Figure 2.6: Example of LZ78.

LZ78. The LZ78 [ZL78] compressor replaces the *sliding window* by a dictionary that stores all the processed subsequences. The algorithm reads one symbol at a time and locates it in the dictionary. If the symbol is stored in the dictionary, the algorithm reads the next symbol and concatenates it with the previous one, creating a subsequence of size two. Again, the algorithm continues appending the read symbols until the subsequence is not in the dictionary. In that case, we have found the longest matching entry (e_k). The subsequence is encoded as the pair $\langle k, c \rangle$, where k is the index of the dictionary entry, and c is the symbol that follows e_k in the input. A new entry that corresponds with $e_k \cdot c$ is added to the dictionary. Those steps are repeated until processing the whole sequence.

To locate the dictionary's entries efficiently, the algorithm builds a *trie* on the dictionary. That is, there is a tree where each node points to a dictionary entry e_i, representing the subsequence obtained by appending the symbols in the path from the root to its corresponding node n_i. Processing the text, for each read symbol, we traverse the tree downwards. We found the longest match (e_k), when no edge allows moving to the next symbol of the sequence. After finding e_k, the dictionary adds a new entry and updates the trie. An example of LZ78 and its trie, with the previous string used in LZ77 (*aababbabaabb*), is shown in Figure 2.6.

Although LZ78 compression is faster than LZ77, its decompression speed is slower than LZ77. A variant of LZ78 is LZZW [Wel84], which the base of GIF image format and the Unix *compress* program. The main difference is that LZW only points to entries in the dictionary, it does not add the extra symbol. For that, LZW initializes the dictionary and trie with the alphabet symbols. As a consequence, LZW gets better compression ratios than LZ78.

RLZ. Known as Relative Lempel-Ziv [KPZ10], this technique is largely used for compression of highly-repetitive sequences. The main difference with the previous approaches is that it builds a *reference*, in other words, it creates a static window

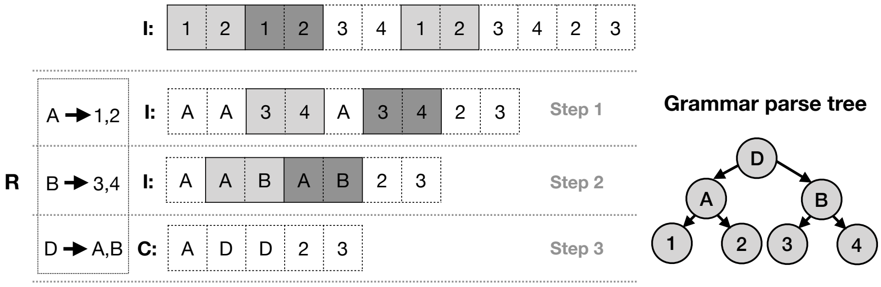


Figure 2.7: Example of Re-Pair over a sequence of integers. The most frequent pairs in each step are colored in gray and the rules created in each step are stored in R.

whose information is highly representative of the sequence to compress. Considering that the sequence S can be composed of several sub-sequences, that reference can be *real* or *artificial.* Whether one of those sub-sequences is chosen as reference, the reference is *real.* On the other hand, when the reference is built by joining parts of those sub-sequences, it is called *artificial.* One of the most powerful methods in DNA for building an artificial reference is based on taking uniform samples of the sub-sequences and join them in the reference [LPMW16].

After choosing a representative reference, the sequence is compressed as a list of pairs $\langle p, l \rangle$ computed with the LZ77 parse, but instead of making reference to the sliding-window, they point to the reference. Notice that, RLZ uses pairs instead of triples, because it does not store the symbol that mismatches the sequence. Unlike the previous techniques, RLZ allows random access to any part of the data, without decompressing the whole previous information.

2.1.5.2 Grammar Compression: Re-Pair

Along the time, several techniques [NMW97a, LM00, KYNC00, YK00, KY00, NM96, NMW97b, Bry86] propose a hierarchical way for compressing text, well known as grammar compression. This kind of compression gives a more structured way of compression, which is more suitable for random access, pattern matching, etc. Those techniques compress the sequence S into a single sequence C and a context-free grammar G. With C and G, the original sequence S can be obtained without losing information. Re-Pair [LM00] is one of the most powerful grammar-based compression techniques since it was used during the development of this thesis, we explain it in more detail.

Re-Pair. Re-Pair [LM00] is a grammar-based compression method. Given a sequence of source symbols I (called *terminals*), the method proceeds as follows: (1) it obtains the most frequent pair of source symbols ab in I; (2) it adds rule $s \rightarrow ab$

to a dictionary R, where s is a new symbol not present in I (called a *nonterminal*); (3) every non-overlapping occurrence of ab in I is replaced by s, and (4) steps 1-3 are repeated until all pairs in I appear only once (see Figure 2.7). The resulting sequence after compressing I is called C. Every symbol in C represents a phrase (a sequence of one or more of the source symbols in I).

Figure 2.7 shows an example where Re-Pair is applied over a sequence of integers. After detecting that the most frequent pair of I is $\langle 1, 2\rangle$, a new *nonterminal* is created ($A \rightarrow 1, 2$). The nonterminal A replaces every pair $\langle 1, 2\rangle$. As a consequence, I is updated to the sequence of Step 1. This process is recursively repeated. The next step finds the most frequent pair $\langle 3, 4\rangle$, and replaces it by the *nonterminal* B. After replacing the pair $\langle A, B\rangle$ with D, there is no repeated pair in I. Hence, the algorithm stops, and I turns into the final sequence of Re-Pair, renamed as C. The result of Re-Pair is composed of C and the dictionary R.

If the length of the represented phrase is 1, then the phrase consists of an original (terminal) symbol; otherwise, the phrase is represented by a new (nonterminal) symbol. We consider that each nonterminal of C contains a grammar parse tree, that is, a tree whose root corresponds with the nonterminal symbol and its children are the right part of its rule. When those nodes are nonterminals, their subtrees are recursively obtained. For example, in the right part of Figure 2.7, we show the grammar parse tree of D. Re-Pair can be implemented in linear time [LM00], and a phrase may be recursively expanded in optimal time (i.e., proportional to its length).

2.2 Compact data structures

The objective of compact data structures is to represent data in a compact way, where the space for representing those data is the minimum possible but keeping the capacity to access any datum in an efficient way. That is, represent the data (text, sequences, trees, etc.) in such a way that the space of storing that representation is smaller than the size of the original data, and decompressing the whole representation is not necessary to access the data. Since those structures are compact and fit in main memory, they take advantage of memory hierarchies where accessing the data is faster in higher levels (e.g., main memory) than in lower levels (e.g., disk). In many cases, they also provide indexes that allow us to answer queries even faster than performing those queries over the uncompressed representation.

2.2.1 Rank and select over bit-vectors

Most compact data structures use bit-vectors supporting rank/select operations. Given a bit-vector B of size $|B| = n$ we can define three operations:

- $access(B, i)$, returns the bit value at a given position i of a bit-vector B.

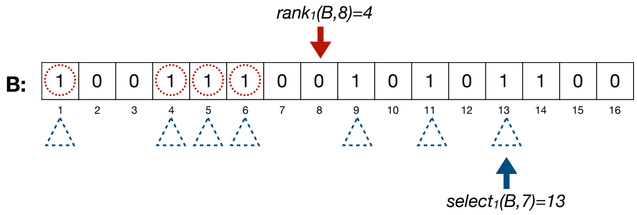


Figure 2.8: Examples of *rank* and *select*.

- $rank_b(B, i)$ counts the number of times the b bit appears from start to the given position i.
- $select_b(B, j)$ returns the position of B where is located the j-th occurrence of the b bit.

These operations are illustrated in Figure 2.8, and they are the basis of most of the compact data structures. Jacobson, whose Ph.D. thesis can be taken as the starting point of the study of compact data structures [Jac89], showed that the *rank* operation can be answered in constant time over plain bit-vectors.

Given a bit-vector B of size $|B| = n$, in order to solve the previous operation Jacobson [Jac89] proposes a structure of two levels. The first level is composed of superblocks of size $s = \lfloor \log n \rfloor \lfloor \log n/2 \rfloor$. Each superblock stores the result of $rank_1(B, i)$ for each i multiple of s. In the second level, there are blocks of size $b = \lfloor \log n/2 \rfloor$, where each block stores the relative rank within the superblock until the beginning of that block. We can compute $rank_1(B, i)$ using those two directories. From the first level, we obtain the *rank* value until the previous multiple of s, while the second level returns the *rank* value until the previous multiple of b. By the addition of s and b we can know the result until the last position to the block that contains j. Finally, the algorithm counts how many 1-bits there are between the beginning of the block and j, and that value will be added to the final result. This last step can be solved in constant time by using a *lookup table* storing the result of rank for all possible subsequences of size b. Notice that numbers in $polylog(n)$ can be encoded in $O(\log \log n)$. That means that rank can be solved in $O(1)$ time using an additional space of $o(n)$ bits. However, solving select requires binary searches in both levels. Hence this operation takes $O(\log \log n)$ time. Clark and Munro [Cla96, Mun96] proposed a new solution that solves both operations in constant time by using additional structures for computing *rank* and *select*. Those structures add an extra-space of $o(n)$ bits to the original bit-vector, thus the total required space is $n + o(n)$ bits.

Another operation that can be implemented using *rank* and *select* is $select_{next}(B, j)$. This operation returns the position of the next bit set to 1 after position j (included) in the bit-vector B, that is,

$select_{next}(B, j) = select_1(B, rank_1(B, j-1)+1)$. Though *rank* and *select* are $O(1)$ time operations, there is a more practical structure for $select_{next}$ [Nav16], which keeps the $o(n)$ extra-space and $O(1)$ time. In practice, it achieves less space and better response times than using *rank*, and then *select*. That structure is similar to the classical rank structure [Jac89], but instead of storing the number of ones preceding a position, it stores the location of the next 1-bit.

2.2.2 Compressed bit-vector representation

There exist other solutions that provide the operations *access*, *rank*, and *select*, but storing the bit-vector in a compressed way.

Pagh [Pag99] compressed the bit-vector by splitting it into equal-sized blocks. For each block, the number of 1-bits that contains is explicitly stored. Those blocks are compressed with a schema that clusters consecutive blocks into intervals of varying length. An additional two-level structure and lookup table allow us to extract the *rank* information.

A proposal that obtains a space result close to nH_0, where H_0 is the zero-entropy, was proposed by Raman et al. [RRR02]. They propose a technique that can solve *rank* and *select* in $O(1)$ time. The sequence is split into different blocks, and they are classified into classes. Each class gathers all blocks with the same number of 1s. Each block has associated a pair (c_i, o_i), where c_i identifies the class of the block; and o_i is the offset of that block inside the class, which identifies how the 1-bits are distributed inside the associated class. Let b be the size of each block, the cost of representing c_i is $\lceil \log(b+1) \rceil$ bits and o_i uses $\lceil \log(\binom{b}{c_i}) \rceil$ bits.

Other strategies were focused on sparse bit-vectors, those where the number of 1-bits $m \ll n$. Okanahora and Sadakane [OS07] presented several solutions: *esp*, *recrank*, *vcode*, and *sdarray*. All of them assume that the input bit-vector is sparse and has different advantages and disadvantages in terms of speed, size, and simplicity. Also, based on those sparse bit-vectors Navarro [Nav16] proposes a way to compress and support efficient *rank* and *select* operations over bit-vectors with runs, a sequence of identical consecutive symbols, in this case, bits.

2.2.3 Partial sums

Partial sums is a well-known problem. Given an array $A[1, n]$ of small numbers, this problem tries to answer two types of queries: $sum(A, i)$, which computes the sum of the numbers $A[1], A[2] \ldots A[i]$; and $search(A, j)$, which looks for the smallest index i in A whose $sum(A, i)$ is greater than or equal to j. The simplest way of solving it is to store an array S where $S[i] = sum(A, i)$. Consequently, we can solve $sum(A, i) = S[i]$ in constant time, and we can solve $search(A, j)$ with a binary search on S in $O(\log n)$ time.

By using bit-vectors with *rank* and *select* auxiliary structures, it is possible to compute both operations in constant time. An Elias-Fano representation [Fan71, Eli75] of the partial sums considers the previous array S, and builds a bit-vector B of size $S[n]$ where $B[S[i]] = 1$, where $i \in [1, n]$, and the remaining positions are set to 0. It can be understood as the list of the values of S represented in unary. Therefore we can compute $sum(A, i)$ as $select_1(B, i)$, that can be computed in constant time, and $search(A, j)$ as $select_1(rank_1(B, j))$. The space representation is $log(n/m) + O(m)$ bits, close to a differential representation of the S values.

2.2.4 Compressed tree representations

Trees are a largely used structure in many algorithms. For a general tree of n nodes, its classical representation uses $O(n)$ words. Each node requires $w \geq \log n$ bits, thus the space turns out $O(nw)$ bits. The constant of this bound is at least 2, which allows to solve basic operations like visiting the first child or the next sibling. By increasing this factor, we can solve more operations (e.g. obtaining the depth, moving to the parent, obtaining the lowest common ancestor, etc.). Different works which require $2n + o(n)$ bits and can solve most of the operations in $O(1)$ time were proposed [CLL05, Jac89, MRR01, MR01, MR04, GRRR04, GRR04, GRRR06, GMR06, GGG$^+$07, DRR06, HMR07, Sad07, JSS07, LY08, BHMR07, BDM$^+$05, FM08, FLMM05]. The main difference between them resides in the kind of operations they can solve, and the nature of $o(n)$ space, which can fluctuate from $O(n/(\log \log n)^2)$ to $O(n/polylog(n))$. Those representations can be divided into three categories:

- **BP**: An ordinal tree can be represented as a balanced sequence of parentheses (BP), that is, a sequence of opening and closing parentheses identified by 1-bits and 0-bits, respectively. By following a depth-first order traversal, when the algorithm reaches a node for the first time, it adds to the sequence an opening parenthesis "(". The position of every opening parenthesis identifies the corresponding node. When the subtree of a node is completely processed, it appends a closing parenthesis ")". Assuming that the number of nodes in the tree is n, this representation requires $2n$ bits, one for the opening and another for the closing parentheses. The main property of this representation is that any subtree is contiguously stored in the bit-vector. We can compute three different core operations on a sequence of parentheses B:

  - $close(B, i)$: returns the position of the closing parenthesis corresponding to the opening parenthesis "(" at position i.
  - $open(B, j)$: with respect to a closing parenthesis ")" at position j, it returns the position of its corresponding opening parenthesis.
  - $enclose(B, i)$: returns the smallest segment of opening and closing parentheses that contains i. That is, the position $k < i$ such that $[k, close(B, k)]$ is the minimum interval containing i.

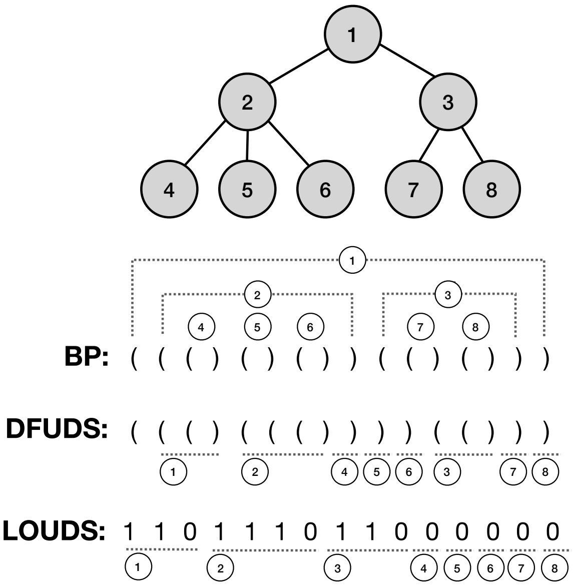

1
2
3
4
5
6
7
8
BP: ((( ) ( ) ( )) (( ) ( )))
DFUDS: ((( ) ((( )))) (( )))
LOUDS: 1 1 0 1 1 1 0 1 1 0 0 0 0 0 0

Figure 2.9: Examples of compressed tree representations.

Jacobson made the first studies in this topic [Jac89] that were later improved by Munro and Raman [MR01] that presented a solution that requires asymptotically optimal space, and every core operation takes constant time. However, the most used implementation is based on *range min-max trees* [SN10], making it possible to resolve these queries in $O(\log n)$ time by using $o(n)$ additional space.

- **DFUDS**: Depth First Unary Degree Sequence [BDM+05, JSS07] is built by traversing the tree in a depth-first order traversal. When the traversal reaches a node, the number of children is appended to the final sequence in unary. For example, if a node contains 3 children, it adds to the sequence the value *1110.* By adding an artificial root, the resulting sequence can turn out to be a balanced sequence of $2n$ parentheses. Therefore the core operations of BP can be used by DFUDS in order to solve basic operations in constant time. More sophisticated operations can be solved in constant time, but adding some additional space.

- **LOUDS**: Level-Ordered Unary Degree Sequence [Jac89] is a tree representation for ordered trees. For each node, from left to right, in a level order traversal, the representation appends to a binary sequence the unary code 1^d0, where d is the degree of the current node, that is, the number of children.

Therefore, the sequence of a tree of n nodes has $2n-1$ bits. Specifically, n bits correspond with the last 0-bit of each node, and $n-1$ bits are set to 1-bit, because each node is a child of another node, except the root. Adding an artificial root node *super-root*, every node of the tree is a child of another node. Thus we can maintain the property that all the nodes correspond to one 1-bit. As a consequence, the length of the final sequence increases by 2 bits. The navigation of the tree is supported by *rank* and *select* operations. Among different operations, LOUDS supports access to children, retrieving the position of the parent or counting the number of children. For example, given a node x and the position i of its 1-bit in S, its first child is located as $select_0(rank_1(i))$. On the other hand, the parent can be computed as $p = select_1(rank_0(i))$.

2.2.4.1 Fully Functional Succinct Tree

In this thesis, we use the recent proposal called fully-functional succinct tree (FF) [SN10]. Based on a BP representation, it combines wide functionality, with little space usage and good time performance. The main component is a *range min-max tree*, which allows us to solve basic and sophisticated operations in constant time. Unlike previous proposals [Sad07, MRR01, MR04, CLL05, LY08], FF does not need auxiliary structures for each supported operation.

The fully-functional succinct tree proposal reduces every operation considered in the state of the art to core operations on BP, that can be solved efficiently by the range min-max tree. Given a sequence $B[0 \ldots n-1]$ of balanced parentheses, we can define $excess(i) = rank_{(}(i) - rank_{)}(i)$ as a function which returns the difference between the number of opening and closing parenthesis in $B[0 \ldots i]$. Note that when $P[i]$ is an opening parenthesis $excess(i)$ is the depth of the corresponding node, while in case of a closing parenthesis, it is the depth minus 1. As a consequence, the core operations on BP can be reduced to:

- $close(B,i) = j$: where $min_{j>i}\{j|excess(j) = excess(i) - 1\}$

- $open(B,i) = j$: where $max_{j<i}\{j|excess(j) = excess(i) + 1\}$

- $enclose(B,i) = j$: where $max_{j<i}\{j|excess(j) = excess(i) - 1\}$

By considering the operator $excess(B,i,j) = excess(B,j) - excess(B,i-1)$, we can define two core primitives on range min-max trees:

- $fwd\_search(B,i,d)$ returns the smallest $j > i$, such that $excess(B,i,j) = d$

- $bwd\_search(B,i,d)$ returns the greatest $j < i$, such that $excess(B,j,i) = d$

These core primitives can be used for solving different operations on trees:

$$\begin{gathered} close(B,i) = fwd\_search(B,i,1) \\ open(B,i) = bwd\_search(B,i,0)+1 \\ enclose(B,i) = bwd\_search(B,i,2)+1 \\ level\_ancestor(B,i,d) = bwd\_search(B,i,d+1) \\ level\_next(B,i) = fwd\_search(B,close(B,i),0) \\ level\_prev(B,i) = open(B,bwd\_search(B,i,0)) \end{gathered}$$

Therefore, the efficiency of those operations depends on the ability of the min-max tree to compute the core operations (fwd_search and bwd_search). The min-max tree is built over a virtual array of $excess(i)$ by splitting it into blocks of size $s = \frac{w}{2}$, where w is the machine word length. For each block, the total excess and the minimum/ maximum local left-to-right excess are stored. After that, blocks are recursively assembled into groups of size $k = O(w/\log w)$, and each formed superblock stores the local excess and minimum/maximum excess within the blocks it holds. The min-max tree turns into a k-ary balanced search tree, which requires $O(n\log(s)/s) = o(n)$ bits of space (see Figure 2.10).

To compute $fwd\_search(B,i,d)$ by the range min-max tree, we first check if it can be solved in the block where the position i is located. For instance, we consider that the block $q = \lfloor i/s \rfloor$ corresponds to the range $[l,r]$ of B. The scan inside the block can be computed in constant time by using lookup tables. Whether the solution is not in the current block, the algorithm has to run a bottom-up traversal from the leaves to the root until finding the first right node which contains $excess(i-1)+d$. If the node is a leaf block, it is scanned by using lookup tables. If the node is internal, the range min-max tree is traversed top-down from the node until finding the leftmost child that contains the desired excess. Analogously, we can solve $bwd\_search(B,i,d)$.

Figure 2.10 solves $fwd\_search(B,6,0)$. Firstly, with lookup tables or a sequential scan, the local excess e_l within that block regarding the position 6 is computed. Since $B[7] = 1$ and given that it is the last position in that block, the relative excess e_l is 1. As the second block is a right child, the algorithm moves up to the parent, N_8. Since this node is a left child, the search continues on its right sibling, N_9. We observe that $e_l + N_9.m \leq d$, since the excess can only increase by ± 1, the solution is contained within the interval covered by N_9. The algorithm moves to the leftmost child of N_9, that is N_3. However, the solution is not contained in N_3 because $e_l + N_3.m \nleq d$. The next node to process is N_4, the right sibling of N_3. As the N_3 was completely skipped, e_l has to be updated with the addition of $N_3.e$. In this case, $N_3.e = 0$ and e_l keeps its previous value. The node N_4 contains the solution ($e_l + N_4.m \leq d$), by scanning its content, we look for the position where e_l turns out $d = 0$, in our case, at position 14.

Another important operation is $rmq(i,j)$ (resp. $rMq(i,j)$), which computes the

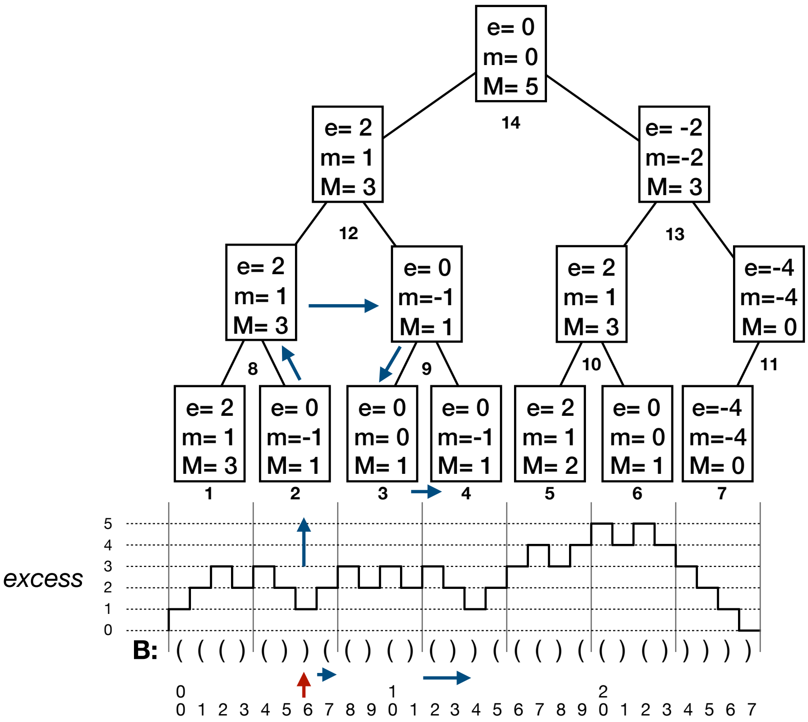


Figure 2.10: Example of *fwd_search*.

position of the minimum (resp. maximum) excess within the interval $[i, j]$. For solving it in min-max trees, the algorithm firstly computes the minimum value in that range. The algorithm starts with a linear scan in the first block intersecting the queried interval between positions $[s, e]$. During that scan, the algorithm tracks the local excess e_l and the minimum excess m_l. If $e \geq j$, m_l is the minimum value. Otherwise, the algorithm keeps looking for the minimum in the range $[e+1, j]$ by traversing the nodes of the min-max tree completely contained in $[e+1, j]$, from the leaves to the root. Once the right sibling is not completely contained in $[i, j]$, the algorithm runs a top-down traversal that recursively checks which of its left descendants contain $j+1$, and its corresponding leaf is scanned. During the whole traversal, m_l value is updated with the minimum excess processed, thus m_l contains the minimum excess in $[i, j]$. Finally, the leftmost position where that minimum excess can be computed as $fwd\_search(B, i-1, m_l)$.

In our example of Figure 2.11, the algorithm starts scanning the second block (N_2). Since $B[7] = 1$, e_l and m_l are updated to 1. The algorithm continues with the parent of N_2 (N_8). In that case, N_8 is a left child, thus its covered area was processed,

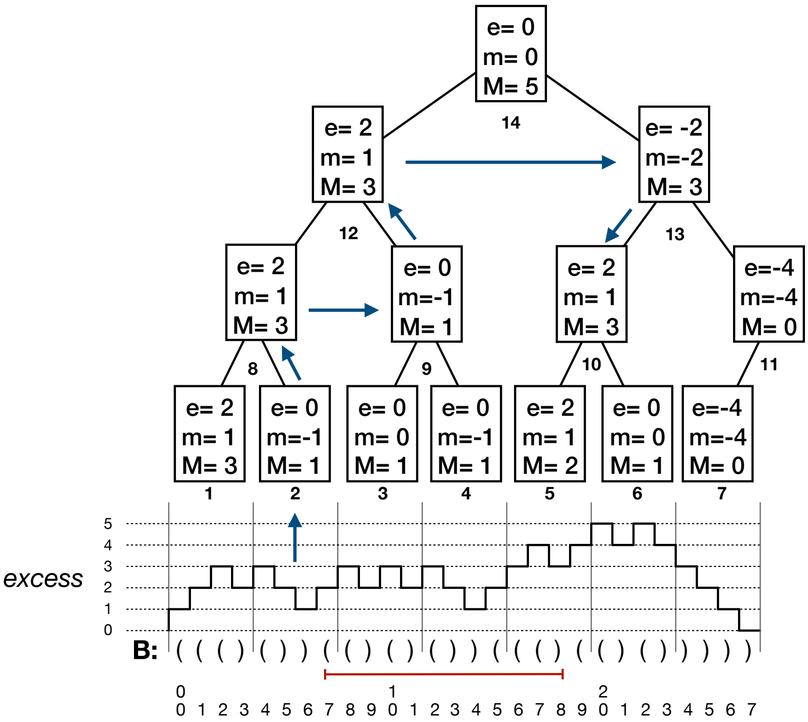


Figure 2.11: Looking for the minimum value between $[7, 18]$.

and consequently, the algorithm moves to its right sibling N_9. Since $e_l + N_9.m < m_l$ $(0 < 1)$ the value of m_l is updated to $e_l + N_9.m = 0$. The algorithm follows with N_{12} and updates e_l to $e_l + N_9.e = 1$, that is, the excess previous to the right child of N_{12} (N_{13}). Now, as N_{12} is a left child, the next node to process is N_{13}. This is the first node that is not completely contained in the queried interval, thus the algorithm starts, from N_{13}, a top-down traversal looking for a better local minimum excess. N_{13} can improve m_l because $e_l + N_{13}.m < m_l$ $(-1 < 0)$, so it continues checking its children. The left child N_{10} cannot improve m_l $(e_l + N_{10}.m \geq m_l)$, and the right one does not intersect the queried interval. Therefore, the algorithm stops and returns the local minimum excess, $m_l = 0$. In other words, the algorithm has find the local minimum excess m_l. By running $fwd\_search(B, i-1, m_l) = fwd\_search(B, 6, 0)$, as in the previous example (see Figure 2.10), the minimum value between $[7, 18]$ is computed at position 14.

2.2.5 Permutations

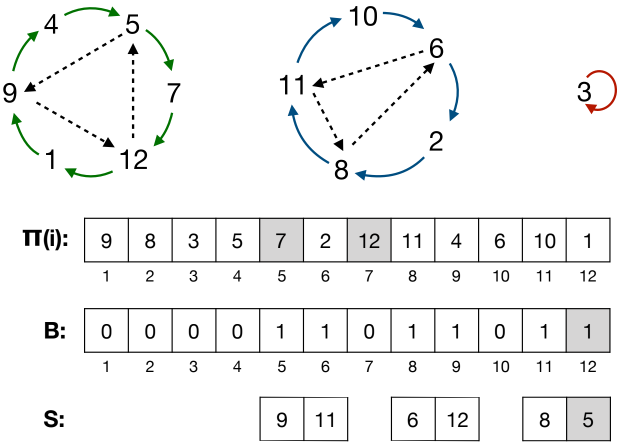
A permutation π of size n is a reordering of the values $\{1, \ldots, n\}$, it can be represented by an array $\pi[1, n]$ where each value in $\{1, \ldots, n\}$ occurs only once. Storing the permutation in an array requires $n \log n$ bits, we can solve in constant time the basic operation $\pi(i)$, which recovers the value at position i.

In some cases, more advanced operations are required, for instance, the inverse permutation of i, $\pi^{-1}(i) = j$, which returns the number j where $\pi(j) = i$. The simplest way for solving the last operation in constant time is to store an additional array $I[1, n]$ where $I[i] = \pi^{-1}(i)$, but it doubles the space. However, the permutations can be decomposed in *cycles*, which can speed up these kinds of queries [MRRR12]. Let us define the recursion of applying π over i as $i_0 = i$, $i_1 = \pi(i)$, $i_2 = \pi(i_1) = \pi(\pi(i))$, and so on. We discern a cycle, when starting at position i, after k recursive steps of ap plying π we reach $i_k = i$. Since $i_{k-1} = \pi^{-1}(i)$, we can solve $\pi^{-1}(i)$ in $O(k)$ time.

Since k can be as large as n and the expected value of k is $\Theta(n/\log n)$, this solution turns impractical. However, this idea is the basis of a technique called *cycle decomposition*, which splits the permutation into its cycles. Each of those cycles can be of different sizes. Therefore the performance of solving $\pi^{-1}(i)$ depends on the length of the cycle such that i belongs. By introducing a parameter $t \geq 1$, we can guarantee that $\pi^{-1}(i)$ requires at most t steps. To achieve it, on those cycles whose length is greater than t, every t steps in a cycle, we add a shortcut that points to $\pi^{-t}(i)$. An additional shortcut is added when the length of the cycle is not multiple of t. In order to compute $\pi^{-1}(i)$, we start with $j = i$, and we repeat these steps: if $\pi(j) = i$ return j; otherwise, the procedure advances to the next j. Usually $j \leftarrow \pi(j)$ but, in case that there is a shortcut in j pointing to s, the new value of $j \leftarrow s$. It is important to notice that the algorithm only follows the first shortcut.

The shortcuts can be represented by a bit-vector $B[1, n]$ where $B[i] = 1$ such that i contains a shortcut, and an additional array S stores the targets. Notice that the size of S is the number of shortcuts, and the target of a shortcut whose source is at position i can be computed as $S[rank_1(B, i)]$. For example in Figure 2.12 the permutation π contains three cycles: $\{4, 5, 7, 12, 1, 9\}$, $\{10, 6, 2, 8, 11\}$ and $\{3\}$. The shortcuts are placed with $t = 2$. Except the for the last cycle, the remaining ones have these shortcuts: $5 \rightarrow 9$, $12 \rightarrow 5$, $5 \rightarrow 9$, $6 \rightarrow 11$, $8 \rightarrow 6$, and $11 \rightarrow 8$. In order to compute $\pi^{-1}(7)$, we start at 7 checking B and π. Since at position 7 there is no shortcut ($B[7] = 0$), the next position to check is at $\pi[7] = 12$. However $B[12] = 1$, thus there is a shortcut that points to $S[rank_1(B, 12)] = 5$. At $\pi[5]$ we find 7, therefore $\pi^{-1}(7) = 5$.

As *rank* operation can be solved in constant time with an additional space of $o(n)$ bits, computing $\pi^{-1}(i)$ takes $O(t)$ time. Given a length of cycle $\ell > t$, there are $\lfloor \ell/t \rfloor$ shortcuts, which means that the size of S is at most $\frac{2n}{t+1}$ and requires $\frac{2}{t+1} n \lfloor \log n \rfloor$ bits. Besides, B and π need $n + o(n)$ and $n \lfloor \log n \rfloor$ bits, respectively. By denoting $\epsilon = \frac{2}{t+1}$, the total space is $(1 + \epsilon) n \log n + O(n)$ bits and can solve π^{-1}



Π(i):	9	8	3	5	7	2	12	11	4	6	10	1
	1	2	3	4	5	6	7	8	9	10	11	12

B:	0	0	0	0	1	1	0	1	1	0	1	1
	1	2	3	4	5	6	7	8	9	10	11	12

S:												
					9	11		6	12		8	5

Figure 2.12: Example of Permutation.

in $O(1/\epsilon)$ time.

2.2.6 Range Minimum Queries

We can extend the query presented in Subsection 2.2.4.1, which finds the minimum excess of a fully-functional succinct tree, to an array of integers. Let us define an array of integers $A[1, n]$. The range minimum query $rmq(A, i, j)$ between positions i and j of A returns the position k, where $A[k]$ is the minimum value in $A[i \dots j]$. In case that there are two identical minimums, the rmq can return the leftmost or rightmost minimum; by default, it returns the leftmost one. The range maximum query $rMq(A, i, j)$ computes the position of the maximum instead of the minimum. In this explanation, since rmq and rMq are analogous, we only refer to rmq.

The rmq problem in an array is related to obtaining the lowest common ancestor (lca) on a tree, it means, the lowest node that is an ancestor of two given nodes. Specifically, rmq operation can be equivalent to computing an lca [GBT84] on the *Cartesian tree* [Vui80]. A *Cartesian tree* of an array of values $A[1, n]$ is a binary tree whose root is the minimum value in the range $[1, n]$. By assuming that the minimum is at position p, the left and right subtrees of the root are the Cartesian tree of $A[1, p-1]$ and $A[p+1, n]$, respectively. For example, on the left of Figure 2.13 we can observe the *Cartesian tree* of A. Firstly, we look for the lowest number in array A, which is the 1 at position 5. Therefore, we create a root node with a value of 1 and label it with 5 (the position of the value in A). Then, the left and right subtrees will be composed of the values in $A[1, 4]$ and $A[6, 12]$, respectively. We obtain the minimum of $A[1, 4]$, which is the value 3 at position 3, and the minimum of $A[6, 12]$, which is the value 2 at position 10. Consequently the second level is composed with a

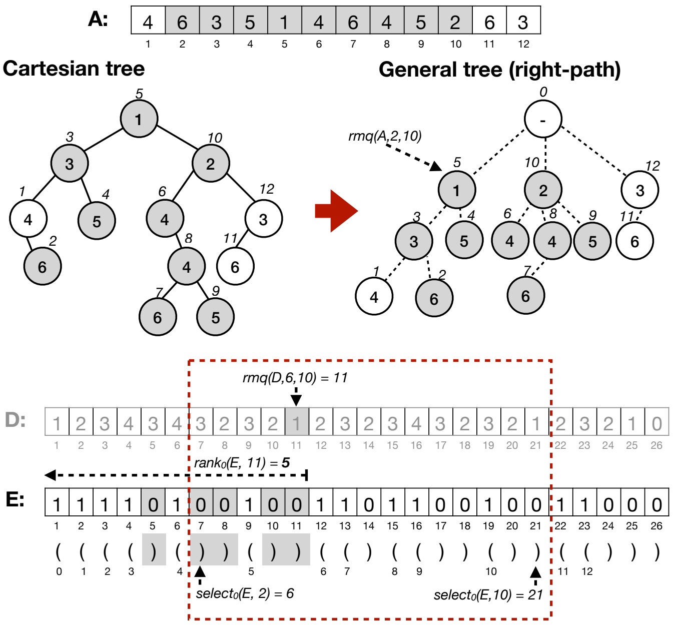


Figure 2.13: Example of Range Minimum Queries.

left child node that contains 3, and a right child node with 2. The next subtrees will be created by applying theses steps recursively to $A[1,2]$, $A[4]$, $A[6,9]$, and $A[11,12]$.

Notice that given a node in a *Cartesian tree* with an inorder position q, its value is located at position q in A, and we can detect the minimum between two nodes by computing its lowest common ancestor. Hence, $rmq(A,i,j) = inorder(lca(innode(i), innode(j)))$, where *inorder* maps from a node to its inorder value, and $innode(i)$ performs the reverse process [BV93, BFCP$^+$05]. That is, we first compute the corresponding nodes at the extremes of the queried interval with *inorder*. Then the node which contains the minimum between those two nodes is computed with *lca*. Finally, we translate the position of the node in the Cartesian tree to its position in array A with *innode*.

The first succinct solution without accessing to A was proposed in [Sad07]. That proposal requires to add $n-1$ artificial leaves on a *Cartesian tree*, getting a space of $4n + o(n)$ bits. In[FH11], the authors propose the first *rmq* structure which can solve this query in $O(1)$ time using $2n + o(n)$ bits. Recently, a structure proposed in [FN17] simplifies the formula of computing the *rmq*, keeping it in $O(1)$ time and $2n + o(n)$ bits of space. In practice, this new structure obtains the best compression ratios and response times, for this reason we explain it in more detail. The main idea is to represent the *Cartesian tree* with BP. Firstly, we need to build the general tree of the *Cartesian tree*. That transformation creates an artificial root, and its children are the nodes in the right-most path of the *Cartesian tree*. In Figure 2.13, the rightmost path traverses the nodes labeled with 5, 10 and 12. Therefore, they turn out the children of the artificial root. With the left subtrees of each node x, the algorithm is recursively applied and takes x as its additional root. For example, in Figure 2.13, the children of the node labeled with 3 are the rightmost nodes from its left subtree of the *Cartesian tree* (nodes labeled with 1 and 2). Once the general tree is computed, an array of depths $D[1, 2(n+1)]$ is created. That array D stores the general tree's depths in a depth-first traversal. Since in D every consecutive cells differ by ± 1, it is transformed into a bitmap E, where $E[i] = 1$ and $E[i] = 0$ whether the difference between $D[i-1]$ and $D[i]$ is positive or negative, respectively. As a consequence, E can be seen as a BP representation where we can solve efficiently *rmq* on D by using a min-max tree. However, the result obtained from $rmq(D, i, j)$ is not equivalent to $rmq(A, i, j)$. Notice that the node u from the *Cartesian tree* with inorder i is the general tree node with postorder i. Since the general tree is represented with E, we can compute the closing parentheses of the inorder position i as $select_0(E, i)$. Consequently, the mapping from the closing parentheses at position p to the inorder value of the node is $rank_0(E, p)$. Therefore, the *rmq* operation on A is computed by the following equation:

$$rmq(A, i, j) = rank_0(E, rmq(D, select_0(E, i), select_0(E, j)))$$

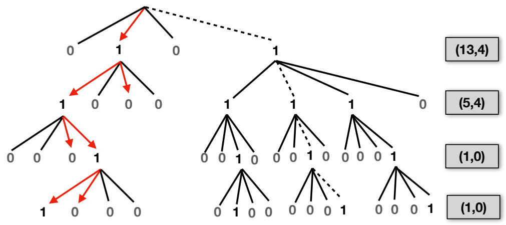
Firstly, the closing parentheses' positions from the inorder values are obtained as we explained before with $select_0$. Then, the lowest common ancestor of those nodes is computed by using the reduction of the *lca* into a *rmq* operation [BV93]. Finally, the inorder value of the node is computed with $rank_0$. Notice that storing D is unnecessary because *rmq* in D can be solved on E with the min-max tree. Hence the structure only requires $2(n+1)$ bits for E and $o(n)$ bits for the min-max tree. On the bottom of Figure 2.13, we can observe the array D and the bitmap E for the array A. Notice that D is shaded because it does not need to be explicitly stored. Additionally, over B and D the algorithm for solving $rmq(A, 2, 10) = 5$ is illustrated.

2.2.7 k^2-tree

The k^2-tree is a compact data structure initially designed to represent Web graphs within reduced space, allowing them to be navigated directly in compressed form [BLN14]. In general, a k^2-tree can be used to represent the adjacency matrix of any graph, as well as binary matrices.

Conceptually, the k^2-tree is a k^2-ary tree built from a binary matrix by recursively subdividing the matrix into k^2 submatrices of the same size. It starts by subdividing the original $n \times n$ matrix into k^2 submatrices of size n^2/k^2. The submatrices are ordered from left to right and from top to bottom. Each submatrix generates a child of the root node whose value is 1 if there is at least one 1 in the cells of that

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
T	0	1	0	1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1

Figure 2.14: Example of k^2-tree.

submatrix, and 0 otherwise. The subdivision proceeds recursively for each child with value 1 until it reaches a submatrix full of 0s, or until it reaches the cells of the original matrix (i.e., submatrices of size 1×1). Figure 2.14 shows an example of a k^2-tree. In the first level, the first and third nodes are set to 0, because in the submatrices $[0, 8] \times [7, 15]$ and $[0, 0] \times [7, 7]$ there is no cell with a 1. Instead the second and fourth nodes are marked with a 1, and they are split into four submatrices, in the next level.

Given an area, the positions with the values set to 1-bit can be detected by going through the tree in a top-down traversal, following those branches with nodes that contain any part of that area. Notice that for a node whose submatrix has the top-left corner in (x, y), its i-th submatrix at the next level ℓ starts at position:

$$(x + ((i-1) \mod k) \times side, y - (i-1)/k \times side)$$

, where $side = n/k^{\ell}$ is the side of the submatrix. When $\ell = 0$, (x, y) is initialized to $(0, n)$.

For example, in Figure 2.14, given the area $[9, 10] \times [10, 13]$ the arrows show the followed path to find the 1s within that area. In the first level, that area is contained within the second submatrix of side 8 whose top-left corner is $(8, 15)$, so the traversal goes through the second branch. In the second level, the first and third branches contain that region, but the submatrix of the third branch is empty, thus the traverse continues only with the first branch. That branch corresponds with the submatrix whose top-left corner is $(8, 15,)$ of side 4. A similar case occurs in the next two levels, where two nodes are contained in the area, but only one has information. Finally, the algorithm reaches the leaf of the cell $(10, 13)$ that contains the one.

We can determine the location of a specific 1 in a cell by traversing the tree from the corresponding leaf to the root. To obtain that location, we initialize $(x, y) \leftarrow (0, 0)$. After reaching the i-th child of a node at level ℓ, we update (x, y) as:

$$(x + ((i - 1) \mod k) \times side, y + (k^2 - i)/k \times side)$$

, where $side = n/k^\ell$ is the size of the side in the current level ℓ.

Figure 2.14 illustrates the bottom-up traversal to obtain the location of the third 1 with the dashed lines. In the right, we can observe how the values of (x, y) are updated in each level.

Instead of using a pointer-based representation, the k^2-tree is compactly stored using bit-vectors T and L (see Figure 2.14). T stores all the bits of the k^2-tree, except those in the last level. The bits are placed according to a level-wise traversal: first, the k^2 binary values of the children of the root node, then the values of the second level, and so on. L stores the last level of the tree, consisting of cell values of the original binary matrix.

We can simulate the navigation of the tree via *rank* and *select* operations over bit-vectors T and L. For example, assuming a value of 1 at position p in T, its k^2 children start at position $p_{children} = rank_1(T, p) \cdot k^2$ of T. If the children of a node return a position $p_{children} > |T|$, the actual values of the cells are retrieved by accessing $L[p_{children} - |T|]$. Similarly, the parent of position p in $T : L$ is $q - (q \mod k^2)$, where $q = select_1(T, \lfloor p/k^2 \rfloor)$, and $q \mod k^2$ indicates the submatrix of p within that of its parent. Those operations make it possible to compute in logarithmic time the 1s within a region and the location of a specific 1.

2.2.8 Direct Addressable Codes

Directly Addressable Codes (DACs) [BLN13] is a structure that gives direct access to variable-length codes, that is, given a sequence of variable-length codes DACs support the decoding of the i-th code without the need to decompress the preceding integers. If the sequence of integers has many small numbers and few large ones, then DACs obtain a very compact representation.

The basis of DACs is splitting the variable-length codes into blocks of fixed length and store them in different levels. Given a sequence of integers $X = x_1, x_2, \ldots, x_n$, DACs take the binary representation of that sequence and rearrange it into a level-shaped structure as follows: the first level, B_1, contains the first (least significant)

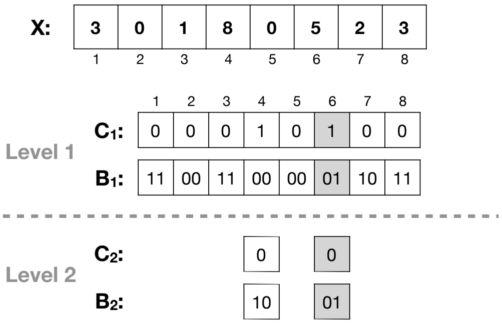


Figure 2.15: Example of Direct Addressable Codes.

n_1 bits of the binary representation of each integer. A bit-vector C_1 is added to indicate whether the binary representation of each integer requires more than n_1 bits (1) or not (0). In the second level, B_2 stores the next n_2 bits of the integers with a value of 1 in B_1. A bit-vector C_2 marks the integers that need more than $n_1 + n_2$ bits, and so on. This process is repeated for as many levels as needed. The number of levels ℓ and the number n_l of bits at each level l, with $1 \leq l \leq \ell$, are calculated in order to maximize the compression. Each value x_i is then retrieved using less than ℓ *rank* operations on the bit-vectors C_l and extracting chunks from the arrays B_l.

Figure 2.15 shows how to obtain the value of x_6, that is, the value on X at position 6. Firstly the algorithm looks into $B_1[6]$ for its least significant bits, that is 01. Since $C_1[6] = 1$, x_6 requires more bits, thus it has to retrieve information from the second level. Specifically, its two most significant bits are stored at position $k = rank_1(C_1, 6) = 2$ in B_2. By appending $B_2[2]$, the value of x_6 up to the second level is 0101. Finally, $C_2[k]$ is checked, as it contains a 0-bit, there are no more bits for x_6. Therefore the algorithm retrieves $x_6 = 0101_2 = 5$.

Chapter 3

Previous work

The different proposals in state of the art for the management of moving objects and their trajectories could be roughly classified in two groups depending on the kind of objects: objects whose movements are restricted to a network, e.g., road, street or public transportation networks; and objects that are moving without any restriction in the space (boats, planes, birds, etc.). This thesis fits in the second group, we consider trajectories where objects move without any restriction, that is, moving freely in a two dimensional space.

In this chapter, we introduce different techniques for the treatment of those trajectories. Section 3.1 shows different techniques for indexing trajectories. Then, Section 3.2 explains different lossy and lossless compression techniques for trajectories. Section 3.3 presents several systems that support indexation and compression at the same time. Finally, Section 3.4 presents the capabilities of each type of representation to solve queries, and the main advantages of our proposals compared with the previous work.

3.1 Indexing trajectories

Since this thesis covers trajectories without considering the constraints of a network, we can define a trajectory as a set of tuples $\langle t, (x, y)\rangle$ where t is a time instant and (x, y) are the coordinates of the object in the space at t. Though there are different approaches, most of the spatio-temporal indexes are based on the *R-tree index* [Gut84].

3.1.1 Spatio-temporal indexes based on R-trees

The *R-tree index* [Gut84] is a classic spatial structure designed to index spatial objects. The key concept behind the R-tree is the *Minimum Bounding Rectangle*

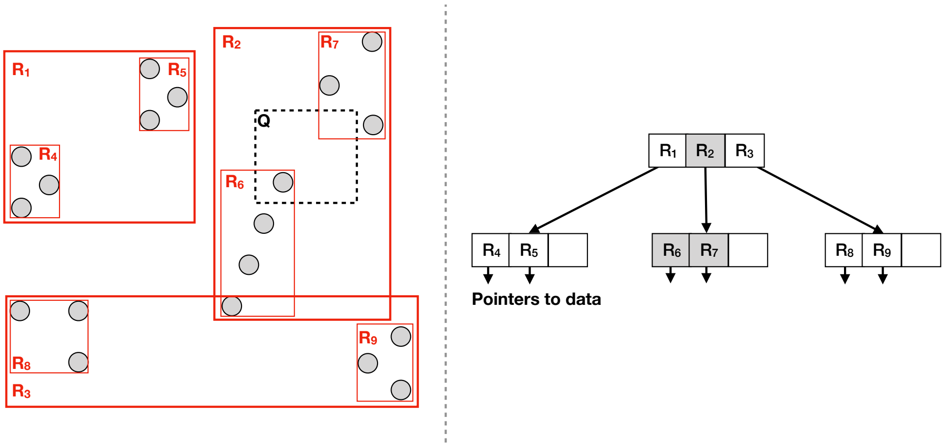


Figure 3.1: Example of R-tree

(MBR), the minimum rectangle which covers a set of objects. The structure of an R-tree is quite similar to a *B-tree*. We can define an *R-tree* as a balanced search tree where each node includes an MBR which wraps the MBRs of their children or objects, in case that the node is an internal node or a leaf, respectively. An example of R-tree is shown in Figure 3.1. The left part shows the aggregation of MBRs, and in the right part we can observe the corresponding R-tree.

In order to know which objects are contained by an area or region, the search can be solved efficiently by descending through the nodes whose MBR intersects with the queried area. In Figure 3.1 we illustrate how to obtain the objects within the queried area Q. As Q overlaps R_2 we descend to its children. In the next level, there are two MBRs (R_6 and R_7) and both intersect with Q. Hence, the pointed data of both MBRs are checked. Since the data contain the actual location of each object, we can discern that the only object within Q is located on the top-right corner of R_6. Therefore, the queries can be solved faster when the number of intersected MBRs is minimized. Hence the top-down traversal of the tree can be sped up by reducing the size of the MBRs. For this reason, the strategy of building MBRs tries to arrange groups of objects in such a way that the built MBRs are as small as possible.

Though they were designed as a dynamic structure, there is a version of a static R-tree [BLNS13] where each MBR is represented with its four corners in compressed form. The bottom-left corner is stored as the difference with the bottom-left corner of the MBR from the parent. The remaining corners of the current MBR are encoded as the difference with its bottom-left corner.

Some spatio-temporal indexes like the 3DR-tree [VTS98] replace MBRs by Minimum Bounding Boxes (MBBs). An MBB is composed of three dimensions, the two spatial coordinates, and an additional dimension that represents the temporal characteristics. Since this new dimension can cover a considerable interval of time, the MBBs become large, the search performance is damaged. For this reason, [PJT00] tries to solve this problem with two different approaches: *STR-Tree* and *TB-Tree*. The first index is an extension of an R-tree that modifies the strategy of construction of MBBs. Their proposal considers the trajectory orientation. That is, the MBBs are built by keeping segments of the same trajectory together, not only regarding the spatial dimensions. Instead, the TB-Tree inserts partial trajectories as MBBs of an R-tree.

Another family of spatio-temporal indexes is the family of versioned R-trees, which stores an R-tree (*version*) per each covered time instant and a B-tree to select the relevant R-trees. Storing a version per time instant requires a large amount of space. To overcome this, instead of storing the complete R-tree for each time instant, these techniques store only the part of the version that is different from the previous one. For instance, MR-Tree [XHL90] and HR-tree [NS98] can share branches between consecutive R-trees. Their main disadvantage is the duplication of objects, which results in high space consumption. Additionally, solving queries that involve a significant interval of time is not efficient. The HR+-tree [TP01a] is an improved version of the HR-tree, which reduces the space of the HR-tree to 20%, and performs better in all kind of queries.

3.1.1.1 Multi-version R-tree

A Multi-version R-tree (MVR-tree) [TP01b] can contain multiple R-trees (*versions*). In a similar way to HR-trees, those R-trees can share between them those parts that do not suffer any change in their MBRs (see Figure 3.2). In order to simulate that behavior, the MVR-tree is composed of a set of records where each entry is a tuple $\langle S, t_s, t_e, ptr\rangle$. S is the MBR covered by that node during the interval of time $[t_s, t_e]$. For an internal node, *ptr* points to the next level; otherwise, the node is a leaf, thus it points to the data corresponding to that MBR. Notice that an entry can cover different time instants since it is *alive* during $[t_s, t_e]$.

In Figure 3.2, we illustrate an example of MVR-tree where the first version corresponds with Figure 3.1. The updates of each MBR through different versions are shown on top of Figure 3.2. Notice that each version is associated with an R-tree, and dashed lines show nodes that are replaced by nodes from a previous version. For example, in the third version v_3, the children of R_1 are the same as those of R_1 in v_2 which are identical to those of v_1. The children of R_3 in v_3 are equal to the previous R-tree.

Solving queries that involve one time instant is very efficient in MVR-trees. For example, in order to know which objects are within a region, the algorithm looks for the *version* which involves the queried time instant, and then the query is solved as a range search in an R-tree. However, solving queries covering an interval of time need to check multiple R-trees. In Figure 3.2 we can observe how to compute the

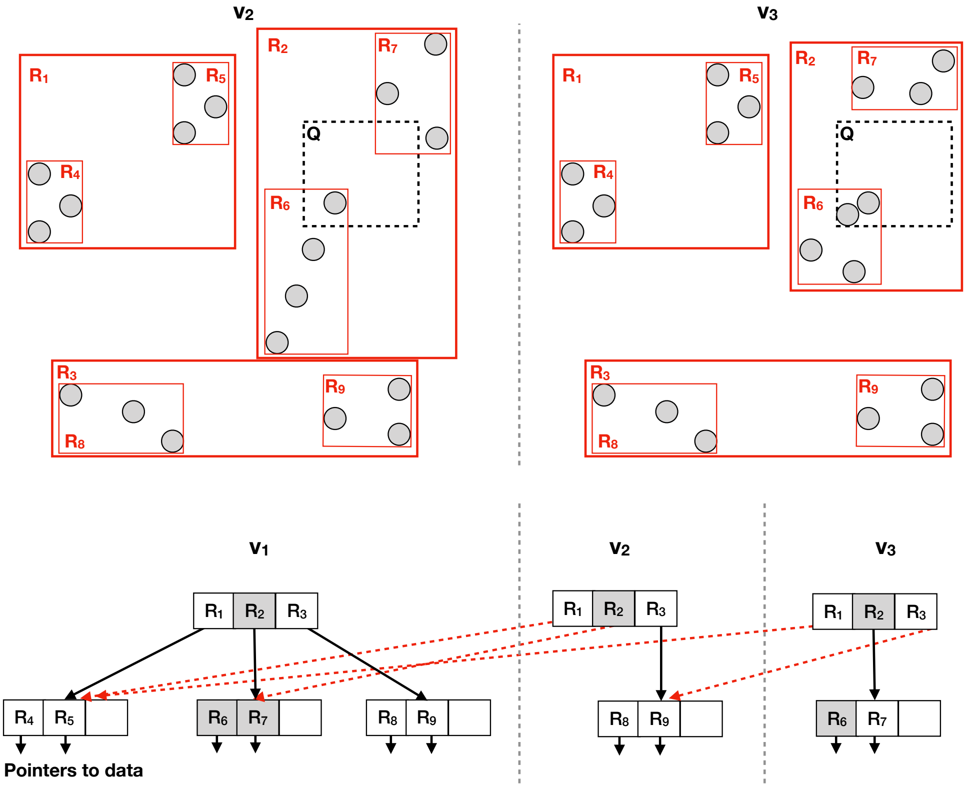


Figure 3.2: Example of MVR-tree

objects within Q during a range of time that covers versions v_1, v_2 and v_3. Thus the algorithm needs to check all these versions. The algorithm starts with a top-down traversal through the first version v_1, and follows the nodes that intersect Q. In the first level, R_2 is checked because R_2 overlaps Q. In the next level, since R_6 and R_7 intersect with Q, both are checked. By accessing the data with the pointers of the leaves, we obtain the objects within Q. In the example, there is only one object, and it is added to the solution. The very same traversal is repeated for v_2, obtaining the object already in the solution. In v_3, the main difference is that R_7 does not intersect Q. Therefore, only the data from R_6 is checked. This produces another object that is contained by Q, and thus it is appended to the previous solution. After processing these three versions, the algorithm returns the two objects found in v_1 and v_3.

In the MV3R-tree [TP01b] the versions are composed of a combination of MVR-trees and auxiliary 3DR-trees [VTS98]. The former part is built on the leaves of the

MVR-trees and tries to improve the performance on queries that involve more than one time instant. This structure obtains the best time performance in interval and time instant queries.

3.1.2 Grid-based indexes

Grid-based indexes split the space into several partitions and build a temporal index for each partition. The Scalable and Efficient Trajectory Index (SETI) [CEP03] divides the space into cells and, for each cell, indexes the trajectories by time with a variant of an R-Tree, called R*-Tree [BKSS90]. This variant improves the split heuristic and gets a better query performance. Another example of grid-based indexes is Multi Time Split B-Tree [ZZS$^+$05] where all the cells are indexed with a TSB-Tree [LS89]. Instead, the Compressed Start-End tree [WZXM08] uses a combination of B+-trees, dynamic arrays and different structures depending on the update frequency of the data. PIST [BMNS08], and GCOTraj [YHC18] are other examples of these indexes.

3.1.3 Other spatio-temporal indexes

The PA-tree [NR07] uses a completely different approach to the previous ones. This index tries to avoid MBBs and spatial indexes, by approximating a series of line or curve segments with a single continuous polynomial. Consequently, the original trajectory and its approximation are not identical. In order to detect false negatives, they keep the maximum deviation between both trajectories. By minimizing this maximum deviation, the index provides better accuracy than MBRs, and its query performance can be significantly improved.

Distributed computing frameworks to handle trajectories have recently appeared. Their structure has two layers: a framework for distributed computing and a set of spatio-temporal indexes. PRADASE [MYQZ09] is a framework based on MapReduce for querying trajectory data by using Hadoop and a spatio-temporal index. Another example of using Hadoop is CloST [TLN12], which splits the data in a hierarchical way. That classification of the data takes into account spatial and temporal dimensions for efficient parallel processing of spatio-temporal queries. TrajSpark [ZJM$^+$17] is a distributed framework based on Spark, which adds a two-level spatio-temporal index called IndexTRDD. One level treats the global data, whereas the other exploits the local data of segments for speeding up the performance of trajectory queries.

Other indexes are based on distributed key/value storage. Most of them are composed of two layers: *storage* and *index*. The first one allows high performance on insertions and managing large data volumes. The second part supports efficient spatio-temporal query processing. Examples of those indexes are MD-HBase [NDAEA13], R-HBase [HWZ$^+$14], and GeoMesa [HAE$^+$15].

Indexes like SEST-Index [GNR$^+$05, Wor05] use snapshots and logs. The snapshots represent the area where each object is located by using a spatial index (e.g R-tree). Each log corresponds with an individual object and is composed of labels that trigger two events: *moves in*, the associated object goes within a region, or *moves out*, the object leaves a specific region.

3.2 Compression of trajectories

There are different techniques for compression of trajectories. However, the simplest one is *trajectory simplification*, which reduces the size of a trajectory by discarding some of its points. This deletion of some points makes it impossible to retrieve the original trajectory from the compressed one, thus it is considered a lossy method.

One way of trajectory simplification is taking points at regular intervals of times, and discarding the remaining ones [PPS06]. In practice, the larger the span of the interval, the smaller the representation of the trajectory. However, that new representation loses too much precision, and the resultant trajectory can be quite different compared to the original one. In order to avoid that gap between the original and compressed trajectory, more sophisticated algorithms were designed.

The Douglas-Peucker algorithm [DP73] keeps the most relevant points and discards those that are redundant. It defines a parameter ϵ and traces a line between the first and last point of the trajectory. When there is a point whose perpendicular distance to that line is greater than ϵ, the furthest point (*outlier*) to that line is chosen as a relevant point. The algorithm is repeated recursively splitting the trajectory into two parts, from the first point until the *outlier* and from the *outlier* until the last point. The algorithm stops when every non-relevant point is located closer than ϵ to its corresponding line. Another similar method is top-down time ratio [MdB04] that takes into account the time. Instead of computing the distance to a perpendicular point of the line traced in Douglas-Peucker, it is measured with respect to the interpolated point at the corresponding timestamp in that traced line. Similar algorithms with different heuristics for measuring the importance of a point are SQUISH [MOH$^+$14], and OPERB [LMZ$^+$17].

Other algorithms try to represent the maximum number of points with a linear segment, for instance, sliding window [KCHP01]. Firstly, the algorithm takes the leftmost position and approximates the next point as a linear function. If that approximation has an error lower than a given threshold, the next point is represented by that segment and continues trying to add new points. Otherwise, the algorithm stops adding points to that segment and starts building a new segment.

Previous algorithms only take into account the spatio-temporal context, other techniques exploit speed and heading of objects to discern which points are more representative in each trajectory. Examples of those algorithms are dead reckoning [TCS$^+$06] and STTrace [PPS06]. On the other hand, methods like [SRL09, TLCF16] decide about the relevance of points by taking into account a network.

Delta compression is the most well-known lossless compression method for trajectories. The first position of the trajectory is stored as an absolute position. The remaining points are represented as the difference between the current position and the previous known location. For example, Trajic [NH15] predicts the next position by taking the previous location, and stores the difference from the actual and the predicted position. Since small numbers can be stored with a short number of bits, it obtains a better compression ratio when the predicted positions are accurate.

3.3 Trajectory compression and indexing

Previous algorithms for compression of trajectories are classical methods in the sense that they do not provide any possibility of querying the compressed data. However, a small number of techniques can compress and search without decompressing all the data.

An example of system that compresses and indexes trajectories is TrajStore [CMWM10]. Every trajectory is divided into subtrajectories, and each one belongs to a cell whose size depends on the data distribution. Those subtrajectories are compressed in each cell by clustering them into similar trajectories and only storing one representative trajectory. For this reason, TrajStore can be considered a lossy method. Additionally, delta compression is used over the chosen trajectories. Concerning the indexation layer, each cell contains a temporal and a spatial (quadtree) indexes.

SharkDB [WZX$^+$14] is another system that supports indexing and compression. The information is split into intervals of time of a fixed length, for each trajectory and range of time, only one spatial point is stored. They are saved on a column of a column-oriented database management system. The data of each column is compressed with delta compression.

3.4 Conclusions

As we explained in Chapter 2, we classified the queries into two groups: *object queries* and *spatio-temporal range queries*. In the previous work, we can observe as each type of representation of trajectories is focused on only one of those two types. For example, the structures presented in Sections 3.2 and 3.3 are good compressing and solving *object queries* like retrieving the original trajectory. However, they cannot solve in an efficient way *spatio-temporal range queries*. Notice that, for detecting whether an object is within a region, those structures must obtain the original trajectory of each object and check if each single point of that trajectory is in that region.

On the other hand, those structures presented in Section 3.1 can efficiently solve *spatio-temporal range queries*. For example, in a MVR-tree, by traversing a version, we can detect the objects which are within a region at a given time instant. Instead, computing the trajectory of an object requires to traverse all the nodes of all the versions of the MVR-tree, looking for that specific object. Therefore, it cannot efficiently solve *object queries*.

As we will see, the main goal of our structures is to join the main advantages of two worlds: compression of trajectories and indexing. They would allow us to compress the trajectories and to solve *object queries* and *spatio-temporal range queries* in an efficient way.

Chapter 4

Basic structure

In this chapter, we explain the basic structure for all the proposals presented in this thesis. To simplify the explanations, we assume that all trajectories start at the same time instant, and there is no absence of information. Therefore, for this conceptual explanation we assume that at any tracked time instant, the position in the space of each object is known. Of course, those conditions are not satisfied when using real data, for this reasons we present, in Chapter 8, how our data structures must be adapted to treat with erroneous and missed data.

Firstly, let us define a trajectory as a sequence $\ell_1, \ell_2, \ldots, \ell_n$. Each ℓ_i is a pair $\langle (x, y), t \rangle$, where (x, y) is the object's location at time instant t. We can denote a *relative movement* m_i as the difference between the location at ℓ_i and ℓ_{i-1}. Therefore, we can obtain a specific location of an object starting from an explicitly stored position (*absolute position*) and the addition of the relative movements of that object up to the desired time instant (*cumulative movement*).

We explain how those trajectories are stored with the basic structure common to all of our proposals in Section 4.1. A brief introduction of the different elements that are part of our basic structure and the mechanism they use to retrieve the location of an object are presented in Sections 4.2 and 4.3.

4.1 Introduction

In this thesis, we present eight different structures that provide a compact and a self-indexed representation of moving object trajectories that support object and spatio-temporal range queries efficiently. All of them share the very same properties:

- A raster model is used to represent the space, which is divided into cells of a fixed size, and objects are assumed to fit into one cell. The size of those cells can be adjusted depending on the domain. Note that smaller cells require more space, but provide more precision.

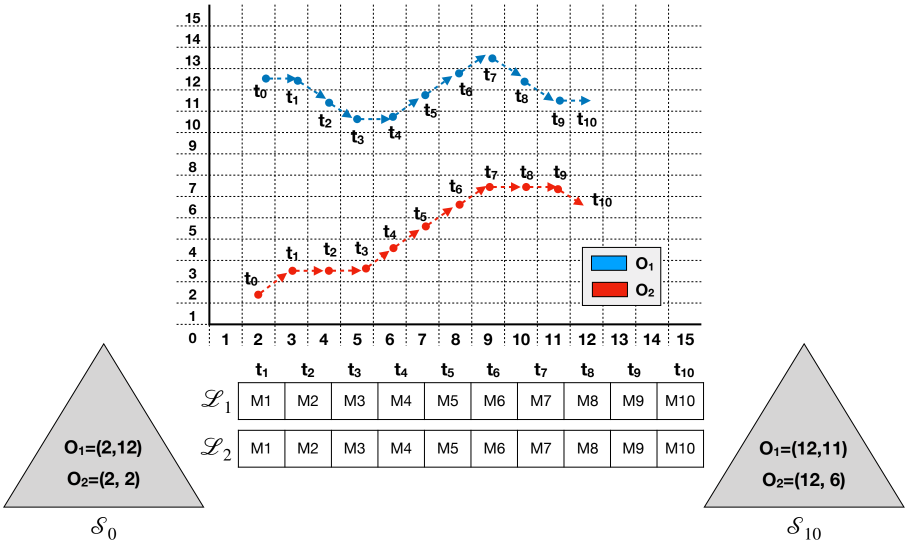

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10
O1
O2
$\mathscr{L}_1$ M1 M2 M3 M4 M5 M6 M7 M8 M9 M10
$\mathscr{L}_2$ M1 M2 M3 M4 M5 M6 M7 M8 M9 M10
O1=(2,12)
O2=(2, 2)
$\mathcal{S}_0$
O1=(12,11)
O2=(12, 6)
$\mathcal{S}_{10}$

Figure 4.1: Example of basic structure and its elements

- The structures assume that the positions of all the objects are synchronized and stored at regular time instants (e.g., every minute). The length of the period between represented time instants is a parameter that can be adapted to the specific domain. The shorter the length of the period is, the more accurate the trajectory representation will be, though achieving less compression.

- All of them are composed of the two same essential elements: snapshots and logs.

  – **Snapshots**: a snapshot stores spatial information of all the moving objects in our space. That spatial information can refer to a time instant or an interval of time. The main goal of this component is to support spatial-temporal range queries and work as a spatial-index. We have designed two different implementations to represent this element.

  – **Logs**: there is a log per object. Each one stores the displacements of its object along the time. It can be considered as the representation of the trajectory, thus makes it possible to compute the position of the object in each time instant. We propose two different ways to encode those displacements.

Figure 4.1 shows two trajectories and their representation with our basic structure. In this case, the snapshots $\mathcal{S}_0$ and $\mathcal{S}_{10}$ store the absolute positions of the objects at time instants 0 and 10, respectively. For example, $\mathcal{S}_0$ represents the position $(2, 12)$ of the object O_1 at time instant 0, and the location $(2, 2)$ of the object O_2 at the same instant. Between the snapshots, there are two logs, one per each object. The first log $\mathcal{L}_1$ store the sequence of displacements of O_1, and $\mathcal{L}_2$ the corresponding sequence of movements of O_2. For example, we denote with M1 in $\mathcal{L}_2$ the movement from the time instant 0 to the time instant t_1. This value depends on the implementation of the log. In the example, M1 represents that the object O_1 moves one position to the East and one position to the North from the previous location.

Since we design two types of snapshots and four types of logs, the combination of those elements builds eight structures for the representation of trajectories, each one with its own properties. All those structures are experimentally evaluated in Chapter 9.

4.2 Snapshots

The snapshots are elements that store the absolute positions, that is, the cell where the object is located in the raster model at a specific time instant. We denote with $\mathcal{S}_h$ the snapshot at time instant h. In addition, the snapshots work as a spatio-temporal index, making it possible to accelerate the computation of some queries.

Since the size of the snapshots can be significant, they are stored every d time instants, that is, there is a snapshot at time instants t_0, t_d, t_{2d}, and so on. Notice that, as the value of d reduces, the space required for the snapshots increases, but some queries can be solved more efficiently, which introduces a space/time tradeoff. That parameter d can be specified depending on the domain of the data. Notice that the initial position of each object is stored at the first snapshot $\mathcal{S}_0$.

In this thesis, we designed two different data structures for the representation of snapshots: snapshots based on k^2-trees and those based on R-trees.

4.2.1 Snapshots based on k^2-trees

This kind of snapshot stores the location of every object at the corresponding time instant by using a k^2-tree and a permutation of the object identifiers. Since the space can be considered as a matrix, where each cell represents a location of the space, we can transform it into a binary matrix, where the 1-bit values mark those cells with objects, and the 0-bit the absence of objects in that cell. That binary matrix can be represented with a k^2-tree, and the identifiers of the objects that are within each cell (leaf of the k^2-tree) are stored in the permutation.

Every time we need to retrieve the location of an object, we can obtain it with two steps. Firstly, we compute the corresponding leaf of that object in the k^2-tree with the π^{-1} operation on the permutation. Finally, the location of that leaf in the space (cell) is computed by running a bottom-up traversal of the k^2-tree.

In addition, we can compute the objects that are within a region, by traversing the k^2-tree from the root to the leaves with objects that are within the queried region and retrieving the object identifiers that correspond with each one of those leaves by using the permutation.

In Section 6.1 we present the snapshots based on k^2-trees in detail, with its structure and algorithms.

4.2.2 Snapshots based on R-trees

The snapshots based on R-trees are focused on storing and indexing the Minimum Bounding Rectangles (MBR) of the trajectory of an object during the interval of time between the current snapshot and the next one: $[t_h, t_{h+d}]$. We store together the MBRs of each object in a snapshot by using a compact representation of an R-tree.

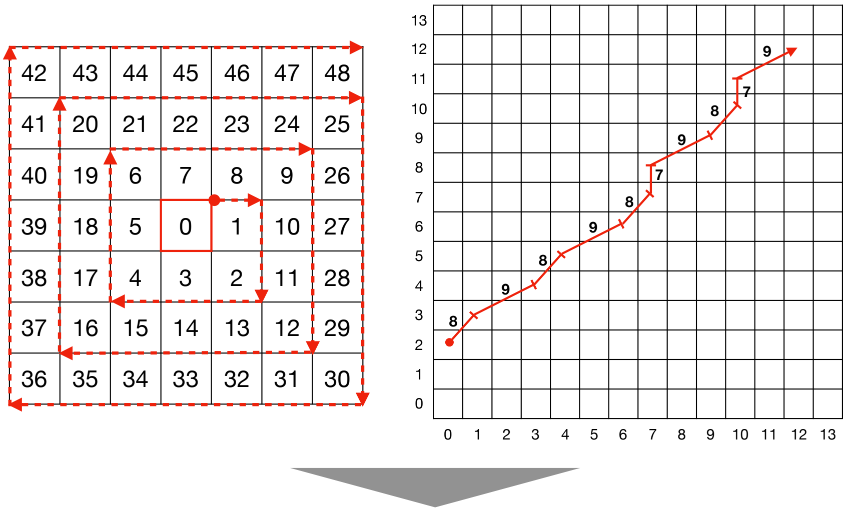
By running a top-down traversal on the R-tree, following the nodes whose MBR intersects with a queried region, we know the MBRs of the objects whose trajectories could intersect with the queried area. That is helpful to detect those objects that are likely to be within the queried region during the interval of time $[t_h, t_{h+d}]$. Once, the objects with chances are retrieved, the log is used to confirm that the object is within the region. Unlike the snapshots based on k^2-trees, those based on R-trees cannot obtain the objects within a region at the snapshot time instant.

The absolute position of every object in the snapshot at time instant t_h is stored by using two arrays X and Y, each one stores the location in the horizontal and vertical axis, respectively. Therefore, we can compute the location of the object in constant time by accessing those two arrays.

A detailed explanation of the snapshots based on R-trees can be found in Section 6.2, where we show its structure and algorithms.

4.3 Logs

We designed four compact data structures to store the log, that is, the sequence of relative movements of each object. Our four structures are composed by a log per object denoted as $\mathcal{L}_{id}$, where id is the identifier of its associated object. Each of these logs stores a sequence of movements represented as the displacement on both axes from the location of the object at time instant t_i to the next one at t_{i+1}. Therefore, given a location $(2, 5)$ and a relative movement $(2, 1)$, we know that the addition of those two values gives us the next location $(4, 6)$. With the help of the log, every position of the objects can be computed as the accumulation of the relative movements up to the queried time instant, from the closest absolute position. For example, in Figure 4.1, the cumulative movement of the object O_1 from t_0 to t_4 is $(4, -2)$. Since the closest absolute position is $(2, 12)$ at t_0, the position at t_4 will be $(4, -2) + (2, 12) = (6, 10)$.



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
$\mathscr{L}_1$	8	9	8	9	8	7	9	8	7	9

Figure 4.2: Example of spiral encoding representation.

We encode the relative movements with two strategies. The first one consists of representing each movement as an integer (*spiral encoding representation*) representing the displacements in the two axis, and the second one uses the classical representation of a vector of coordinates (*coordinates representation*). Below, we explain those two strategies, and we classify the structures that we propose for the representation of the log.

4.3.1 Spiral encoding representation

Our spiral encoding proposes a strategy to store the displacement of an object in a two-dimensional space by using a single positive integer that is shorter when the displacement is nearer to the previous location in the space. For this reason, the cells around the actual position of an object are enumerated following a spiral in which the origin is the previously known position of the object, as it is shown in Figure 4.2 (left).

As an example, assume that an object moves one cell to the East and one cell to the North with respect to the previous known position. With the encoding on the left of Figure 4.2 that movement is encoded as an 8. Figure 4.2 (right) shows the trajectory of an object starting at cell (0,2). Each number indicates a movement between consecutive time instants. Since most relative movements involve short distances, this technique usually produces a sequence of small numbers. At the bottom of Figure 4.2 we can observe the log representation of the trajectory on the right. Since that log is compressible, we designed two different structures of log that use the spiral encoding representation, where the only difference is the approach to compress the log.

- **ScdcCT** assumes that the consecutive displacements of an object stored in the log as a sequence of integers tend to be small. For this reason, it encodes shorter movements with fewer bits than larger movements. To accomplish this, it uses a statistical zero-order byte-oriented compressor, namely (s, c)-Dense Code (SCDC) [BFNP07].
- **GraCT**: observe that the same type of objects tend to do similar movements, that produces a sequence of relative movements where there are identical subsequences between all the objects. GraCT exploits the repetitiveness of patterns of movement between all the objects by using RePair [LM00], a grammar compressor. That is, the log is compressed as a sequence of symbols of two types: *terminals* or *nonterminals*. The terminals correspond with the values of the spiral encoding and, each nonterminal represents a sequence of symbols from the spiral encoding. Additional information is stored for each nonterminal of the grammar, allowing us to improve the time performance of some queries.

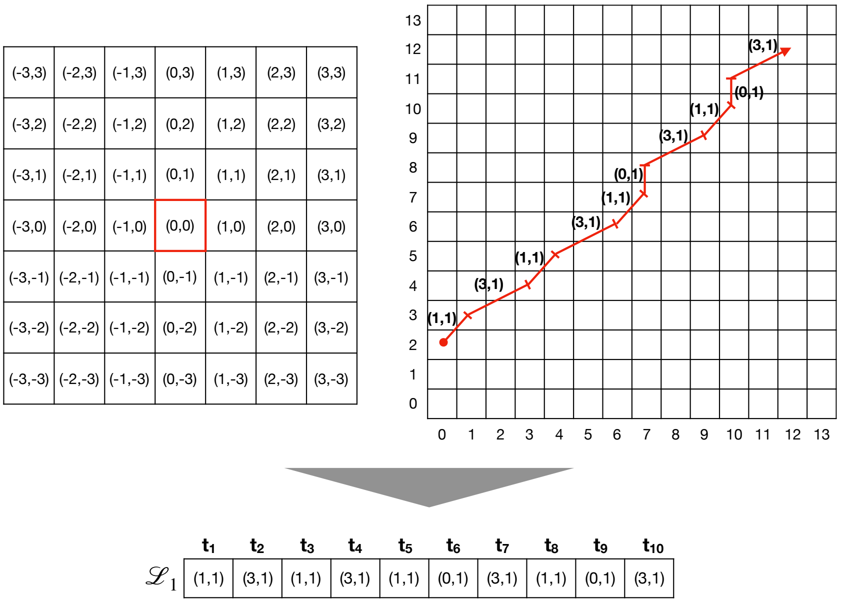
On these techniques, every time a position of an object has to be computed, the algorithm has to retrieve a convenient absolute position from the closest snapshot, and to process the log in order to obtain the displacement of the object from that snapshot until the queried time instant. That is, the algorithm has to accumulate the movements and add them to the absolute position. That procedure requires scanning all the entries from the log up to the queried time instant. Therefore, in both cases, we need a linear traversal of the log to retrieve the desired location.

The details of ScdcCT and GraCT are explained in Sections 7.1 and 7.2, respectively.

4.3.2 Coordinates representation

As we explained before, each log is a sequence of relative movements, which is the displacement of an object from a position to the next one. We need to represent that information in such a way we can efficiently solve the queries. In this case, instead of storing only one positive integer that encodes the displacement in both axis of a two dimensional space, we represent each relative movement with the classical representation of a vector of coordinates, that is, a pair of values where each one corresponds to an axis, and they can acquire positive or negative values.

Figure 4.3 shows an example of a log whose relative movements use the coordinates representation of the left part of the figure. That matrix shows several examples of



(-3,3)	(-2,3)	(-1,3)	(0,3)	(1,3)	(2,3)	(3,3)
(-3,2)	(-2,2)	(-1,2)	(0,2)	(1,2)	(2,2)	(3,2)
(-3,1)	(-2,1)	(-1,1)	(0,1)	(1,1)	(2,1)	(3,1)
(-3,0)	(-2,0)	(-1,0)	(0,0)	(1,0)	(2,0)	(3,0)
(-3,-1)	(-2,-1)	(-1,-1)	(0,-1)	(1,-1)	(2,-1)	(3,-1)
(-3,-2)	(-2,-2)	(-1,-2)	(0,-2)	(1,-2)	(2,-2)	(3,-2)
(-3,-3)	(-2,-3)	(-1,-3)	(0,-3)	(1,-3)	(2,-3)	(3,-3)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
$\mathscr{L}_1$	(1,1)	(3,1)	(1,1)	(3,1)	(1,1)	(0,1)	(3,1)	(1,1)	(0,1)	(3,1)

Figure 4.3: Example of log using coordinates representation.

how to represent the movements, for example, the value $(1, -1)$ corresponds with an object which moves from the previous location one position to the East, and one position to the South. This technique is applied to the movements of the trajectory on the right. The resultant log is at the bottom of the figure.

As in the previous technique, we designed two structures to compress the log with this approach:

- **ContaCT**: the aim of this structure is computing the cumulative movement of an object from the initial time instant up to a given one in constant time, avoiding the linear traversal of the log. Thus we need to obtain the cumulative displacement in both axes until the queried time instant in constant time. To achieve that goal, we store the relative movements of the log in two arrays per axis, where one represents the positive displacements and the other stores the negative displacements in that specific axis. For example, in the vertical axis, those two arrays store the movements to the North (positive) and South (negative). Notice that we can compute the total displacement on the vertical axis as $N - S$, where N and S are the cumulative displacements to the North and the South, respectively. The cumulative displacement on the horizontal axis can be computed analogously.

  We can reduce the computation of those cumulative displacements to a partial sums problem on each of those four arrays. Since the problem of the partial sum can be solved in constant time by using an Elias-Fano representation, we can compute the cumulative movement in constant time.

- **RCT**: this technique is based on RLZ [KPZ10] and tries to join the advantages of GraCT and ContaCT in one single structure. Firstly, an artificial reference, composed of the most frequent patterns of relative movements, is built. Since that reference can be considered as the log of a trajectory, and our goal is to exploit the advantages of ContaCT, we store it by using ContaCT. Finally, each individual log is compactly stored with RLZ. Therefore, the resultant log is composed of a list of phrases from the reference. With the use of $O(z)$ extra-space, where z is the number of phrases, this structure can compute the cumulative movement between two time instants in constant time.

More details about ContaCT and RCT are explained in Sections 7.3 and 7.4, respectively.

Chapter 5

Queries

As we explained in Chapter 3, there are queries focused on retrieving the individual information of an object (*object queries*), and the queries of the classical spatio-temporal indexes, which are interested in obtaining information about those objects that are inside a spatio-temporal range (*spatio-temporal range queries*). Notice that object queries are required on those representations of trajectories that compress the data, and the spatio-temporal range queries are the classical queries supported by spatio-temporal indexes.

Section 5.1 classifies the queries and gives a brief description of each one of them. Section 5.2 and Section 5.3 explain the algorithms to solve the object queries and spatio-temporal range queries on our structures.

5.1 Types of queries

As we explained, we can classify the interesting queries about moving objects into two groups: object queries and spatio-temporal range queries. Below we show the queries that compose those groups and introduce their functionality. For the formalizations, let us define the trajectory of n movements of an object id as $\mathcal{T}_{id} = \{\langle t_0, p_0\rangle, \langle t_1, p_1\rangle, \ldots, \langle t_n, p_n\rangle\}$, where each pair $\langle t_i, p_i\rangle$ stores the position p_i of the object id at time instant t_i.

5.1.1 Object queries

This group is composed of three queries that retrieve the individual information of an object during an interval of time:

- **Object Position**: given an object identifier id and a time instant t_q, this query computes the position of that object at the queried time instant t_q.

Formally, the object position query, for an object identifier id and a time instant t_q, returns the location p_q such that $\langle t_q, p_q \rangle \in \mathcal{T}_{id}$.

- **Object Trajectory**: like the previous query, it computes the position of an object during an interval of time. That is, the object trajectory query, for an object identifier id and a time interval $[t_b, t_e]$, returns the sequence of locations $\langle t_i, p_i \rangle \in \mathcal{T}_{id}$ such that $t_b \leq t_i \leq t_e$, in increasing order of t_i.

- **Minimum Bounding Rectangle (MBR)**: although it is not a classical query, occasionally, we require a summary about the path followed by an object, instead of computing the whole trajectory. The MBR query returns, for an object identifier id and a time interval $[t_b, t_e]$, the smallest rectangular area R such that, for every element $\langle t_i, p_i \rangle \in \mathcal{T}_{id}$ with $t_b \leq t_i \leq t_e$, it holds that $p_i \in R$.

5.1.2 Spatio-temporal range queries

Three queries compute which objects are within a spatial region of the space during an interval of time or a specific time instant.

- **Time Slice**: this query computes those objects within a given rectangular region at a given time instant t_q. That is, this kind of query returns, for a rectangular region r_q and a time instant t_q, the set O of object identifiers such that, for each $id \in O$, there exists a pair $\langle t_q, p_q \rangle \in \mathcal{T}_{id}$ where $p_q \in r_q$.

- **Time Interval**: it is an extension of *time slice* that expands t_q to an interval of time $[t_b, t_e]$. Hence, the time interval query returns, for a rectangular region r_q and a time interval $[t_b, t_e]$, the set O of object identifiers such that, for each $id \in O$, there exists at least one pair $\langle t_i, p_i \rangle \in \mathcal{T}_{id}$ where $t_b \leq t_i \leq t_e$ and $p_i \in r_q$.

- **K-Nearest Neighbors**: given a point p_q in the space and a time instant t_q, it returns the K closest objects to p_q at t_q. Formally, the K-Nearest neighbor query for a point p_q at time instant t_q returns a set O of objects such that $|O| = K$ and $d(p_q, id_1) \leq d(p_q, id_2)$ for any objects $id_1 \in O$ and $id_2 \notin O$, where $d(p_q, id)$ is the Euclidean distance from point p_q to the position of object id at time instant t_q (i.e., p such that $\langle p, t_q \rangle \in \mathcal{T}_{id}$).

The classical solutions to represent moving objects and their trajectories are not able to solve in an efficient way both types of queries in the same structure. For example, the traditional spatio-temporal indexes, which are variants of R-trees, can solve the last group of queries, but the object queries cannot be efficiently solved. On the other hand, those structures that compress the trajectories by using delta-compression or other techniques (see Chapter 3) obtain excellent performance in object queries, but they are incapable of answering spatio-temporal range queries.

Our structures join the advantages of spatio-temporal indexes and compression in the same structure, by using our two elements: snapshots and logs. Consequently, our structures permit us to solve both kinds of queries in an efficient way.

5.2 Solving object queries

5.2.1 Object Position

Given an object O_{id} and a time instant t_q, the query computes the location of the object O_{id} at the time instant t_q. In case that t_q corresponds to a snapshot, we can directly retrieve that information from that snapshot. If the implementation of that snapshot is based on a k^2-tree, it can be obtained with a bottom-up traversal of the tree. Otherwise, the snapshot is based on a R-tree, and we can get the absolute position from the arrays X and Y, which store the absolute positions of all the objects.

In case that t_q is not associated with a snapshot, the algorithm has to obtain the location of that object from the closest snapshot to t_q. For example, if the closest snapshot is at time instant t_h, the location of the object at the time instant t_h is obtained from $\mathcal{S}_h$. Then, assuming that $t_h < t_q$ the algorithm has to process the log to compute the cumulative movement from t_h to t_q, that is, the sum of the relative movements. Finally, the algorithm computes the desired position as the addition of the cumulative movement to the absolute position. Note that, if $t_h > t_q$, the algorithm has to consider the cumulative movement from t_q to t_h, and subtract it to the previous position.

For example, let us look for the position of an object O_1 at time instant t_2, whose log is composed of the relative movements $\{(1,2),(2,0),(1,-1),(-1,0),(2,1)\}$. We assume that the closest snapshot is at time instant 0, and contains the absolute position $(3,4)$ of O_1. The cumulative movement up to t_2 is the sum of the first two relative movements from the log, that is, $(1,2)+(2,0)=(3,2)$. The addition of that cumulative movement to $(3,4)$ give us the location $(6,6)$ of O_1 at the time instant 2.

As we will see in Chapter 7, logs like ContaCT and RCT can compute in constant time the cumulative movement. Therefore, they only need to store the first tracked location of each object and add the cumulative movements from the initial position up to the queried time instant. Consequently, they avoid retrieving the absolute position from the snapshot.

5.2.2 Object Trajectory

To obtain the trajectory $\mathcal{T}$ from an object O_{id} during an interval of time $[t_b, t_e]$ the algorithm is similar to the presented for obtaining the location of an object at a given time instant. Firstly, the algorithm computes the position of the object at t_b by scanning the log, as we have seen above. Then, $\mathcal{T}$ is initialized with that position.

After computing the first position, the algorithm reads the next entry of the log, that is, the following relative movement, which corresponds with the displacement from t_b to t_{b+1}. Consequently, if we sum that relative movement to the last absolute position of $\mathcal{T}$, we obtain the next position of the object at t_{b+1}, which is added back to $\mathcal{T}$. By repeating those steps for each read entry from the log that is covered by the interval $[t_b, t_e]$, we obtain the resultant trajectory.

For example, considering that the time interval $[t_1, t_3]$ and that closest snapshot corresponds with the time instant t_0, we can compute the trajectory of an object by traversing its log $\{(1, 2), (2, 0), (1, -1), (-1, 0), (2, 1)\}$. Firstly we retrieve the location of the object from the snapshot, that is $(3, 4)$. Then we compute the location of that object at the first time instant, $(3, 4) + (1, 2) = (4, 6)$. Therefore, $\mathcal{T}$ is updated to $\mathcal{T} = \{(4, 6)\}$. The algorithm reads the next value of the log, which corresponds to the relative movement $(2, 0)$ and computes the next location as $(4, 6) + (2, 0) = (6, 6)$. After updating the trajectory with the new information, $\mathcal{T} = \{(4, 6), (6, 6)\}$, the last position is computed as $(6, 6) + (1, -1) = (7, 5)$. Hence, the solution is $\mathcal{T} = \{(4, 6), (6, 6), (7, 5)\}$.

5.2.3 Minimum Bounding Rectangle

Given an interval of time $[t_b, t_e]$ and an object, this query computes the minimum rectangle $[x_1, y_1] \times [x_2, y_2]$ that covers the trajectory of that object from t_b to t_e, where (x_1, y_1) and (x_2, y_2) are the bottom-left and top-right corner of that region, respectively.

Some applications do not need to know the exact trajectory of an object to support the queries of its domain. For example, in those applications that detect the objects that are moving together during an interval of time, we can discern which objects move together by computing the area where they move, instead of their trajectory. As we will see, in some structures, computing the MBR can be solved more efficiently than retrieving the trajectory, making it a quite interesting query. It can also be used as a tool for solving time interval queries and other queries.

The general approach of computing the MBR in the interval of time $[t_b, t_e]$ simulates retrieving the trajectory between t_b and t_e, gathering the minimum and maximum values of each axis. The final minimum and maximum values after computing the positions correspond with the resultant MBR.

For example, we consider the time interval $[t_1, t_3]$ to compute the MBR of an object whose log is $\{(1, 2), (2, 0), (1, -1), (-1, 0), (2, 1)\}$ and its previous and closest position is $(3, 4)$. The algorithm computes the position $(3, 4)$ at the time instant t_1 as we showed before, and it starts with the MBR $[3, 4] \times [3, 4]$. The next position is $(3, 4) + (2, 0) = (5, 4)$, since that point is not covered with the current MBR, the MBR is updated to $[3, 4] \times [5, 4]$. Finally, the position $(5, 4) + (1, -1) = (6, 3)$ is computed, and the MBR is extended to the rectangle $[3, 3] \times [6, 4]$ in order to include that point.

Although this general approach requires traversing the log and retrieving each relative movement, in Chapter 7, we show that using the log of GraCT speeds up this query without needing to retrieve the trajectory. In addition, we explain how the logs of ContaCT and RCT solve this query in constant time.

5.3 Solving spatio-temporal range queries

5.3.1 Time Slice

Time Slice computes the objects within a region r_q at a specific time instant t_q. All of our structures start by retrieving from the closest snapshot ($\mathcal{S}_h$) those objects (*candidates*) that have chances to be within r_q at t_q. That is, to avoid scanning all the objects stored in the structure, the first step is selecting those objects that have chances to be within r_q at t_q. To obtain those candidates, the algorithm depends on the implementation of the snapshot.

On those snapshots based on k^2-trees, we have to compute, at the time instant associated with the snapshot, the region around r_q that contains all the objects that can be within r_q at t_q. That region is called *expanded region*. Therefore, every object outside the *expanded region* has no chances to be within r_q at time t_q.

Definition 5.3.1. *The expanded region of a given region r_q during the interval of time $[t_i, t_q]$ is denoted as $\mathcal{E}(r_q, t_i, t_q)$. That region is the result of expanding r_q in all directions $|t_i - t_q| \times M_s$ cells, where M_s is the speed of the fastest object of the dataset.*

Since that region considers the maximum speed of our dataset, every object outside that region cannot reach region r_q after $|t_i - t_q|$ time instants, otherwise the object has to overcome the maximum speed. For example, in Figure 5.1, we assume that the maximum speed is $M_s = 1$ cells per time instants, and the closest snapshot is at time instant t_0. We show two examples, on the left part of the figure $t_q = t_1$, and on the right part $t_q = t_2$. Since we are looking for the objects within $r_q = [5,5] \times [8,8]$, the expanded regions are $\mathcal{E}(r_q, t_0, t_1) = [4,4] \times [9,9]$, in the first case, and $\mathcal{E}(r_q, t_0, t_2) = [3,3] \times [10,10]$, in the second. Therefore, the objects $\{O_1, O_2\}$ have chances to be within the region $[5,5] \times [8,8]$ at the time instant t_1. The other objects cannot be within r_q. Instead, in the second example, since the difference between the snapshot and the t_q is longer, the region is bigger, so the candidates are $\{O_1, O_2, O_3\}$.

On the other hand, on the snapshots based on R-trees, we can directly detect which objects have chances to be within r_q from the previous snapshot to t_q, that is, the one that stores the MBRs whose time interval includes t_q. Since those MBRs provide an idea of the path followed for each trajectory, we can run a top-down traversal through the R-tree to obtain those objects whose MBR intersects with r_q. Note that in case of not intersecting r_q, the object cannot be within r_q at time instant t_q.

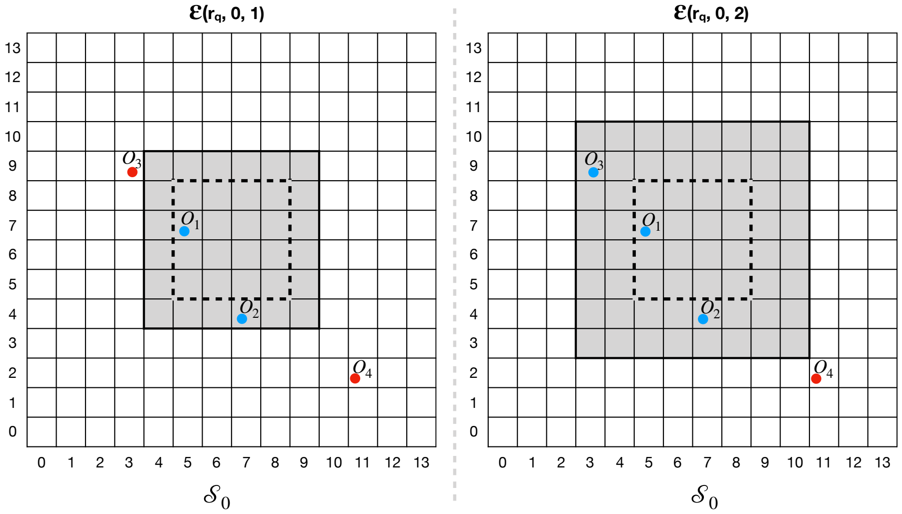


Figure 5.1: Two examples of expanded region, with distinct differences between the snapshot and the queried time instant.

We abstract the operation of obtaining the candidates to be within r_q from the implementation of the snapshot by defining the operation $reach(\mathcal{S}, r_q, t_q)$.

Definition 5.3.2. *$reach(\mathcal{S}_h, r_q, t_q)$ is the function that computes the object candidates stored in $\mathcal{S}_h$ that can be within r_q at time instant t_q.*

Once the object candidates are obtained, the algorithm has to process the log of each one of them to get the position of them at t_q time and check if the object is within r_q to add it to the final solution.

Since some log representations need to traverse the log for retrieving the desired position, we can take advantage of it to detect which objects are moving in the wrong direction. We can detect those objects that are moving further away from r_q, and stop processing its log before computing its location at t_q. To accomplish it, we need to define a new operation *chance*. It is conceptually similar to *reach*, but it considers the position of the object at a specific time instant, which can be a snapshot time instant or not.

Definition 5.3.3. *We define $chance(p_i, t_i, r_q, t_q)$ as a function that returns true when an object located in the cell p_i at t_i have chances to reach the region r_q at time instant t_q.*

After reading an entry of the log, we compute the new location of the object p_i at t_i. In case that $chance(p_i, t_i, r_q, t_q)$ does not hold, the algorithm stops processing that object and continues with the next candidate. We can know whether an object has possibilities of being within r_q at t_q by computing $\mathcal{E}(r_q, t_i, t_q)$, in case that p_i is within that expanded region, the object has chances to be within r_q at t_q.

5.3.2 Time Interval

Time interval queries return the objects that were within a region r_q at any time instant during a time interval $[t_b, t_e]$. An easy solution is to use an algorithm similar to *time slice* but taking into account that the interval of time can involve more than one snapshot. That is, the locations in the interval $[t_b, t_e]$ can be solved by considering snapshots $\{\mathcal{S}_i, \mathcal{S}_{i+d}, \ldots, \mathcal{S}_{j-d}, \mathcal{S}_j\}$, where $\mathcal{S}_i$ and $\mathcal{S}_j$ are the first and last covered snapshots. Therefore, the solution can be computed as the union of the results from the partial solutions of each interval of time between two snapshots: $\{[t_b, t_{i+d-1}], [t_{i+d}, t_{i+2d-1}], \ldots [t_{j-d}, t_{j-1}], [t_j, t_e]\}$. That is, we compute a time interval query in each one of those time intervals.

To solve a partial solution whose corresponding time interval is $[t_i, t_j]$, and the previous snapshot is $\mathcal{S}_h$, we have to obtain the candidates that can be in r_q at t_j by using $\mathit{reach}(\mathcal{S}_h, r_q, t_j)$. Then we have to compute the trajectory of each candidate with a traversal through the log for obtaining the locations between t_i and t_j. In each time instant $t_i \leq t_c \leq t_j$, we obtain a position p_c, and the algorithm checks if p_c is within r_q. If the object is contained, the algorithm stops and adds the object to the partial solution. Otherwise, the algorithm checks $\mathit{chance}(p_i, t_i, r_q, t_j)$. In case that the object has no opportunities to be within r_q at t_j, the algorithm stops processing this object. In case that the object still has chances to be within r_q, the algorithm continues processing that log. The partial solution is obtained when all the logs of the candidates were processed.

In Figure 5.2, assuming that our candidates are O_1 and O_2, let us compute the objects within region $r_q = [8, 5] \times [10, 8]$ at $[t_0, t_{10}]$. Firstly O_1 is processed, the algorithm computes each time instant up to t_5, and O_1 continues with chances to be within r_q. However, after computing the position of O_1 at t_6, $(8, 12)$, we observe as the expanded region $\mathcal{E}(r_q, t_6, t_8)$ does not contain O_1. Therefore, O_1 has no chances to be within r_q during the given interval of time. Hence the object is discarded, and O_2 is processed. O_2 keeps the chances to be within r_q up to t_5, and then it computes the position at t_6, $(8, 6)$. Since O_2 is within r_q, it is added to the solution, and the algorithm stops processing O_2. There are no more objects to process, thus the solution is only composed of the object O_2.

This general approach for solving the partial solutions of time interval queries is improved on those structures that can solve in constant time the Minimum Bounding Rectangle between two time instants. In that case, the algorithm performs a binary search looking for an MBR during the interval of time $[t_b, t_e]$ included within r_q.

Therefore, the algorithm starts computing the MBR of $[t_b, t_e]$ and checks three cases:

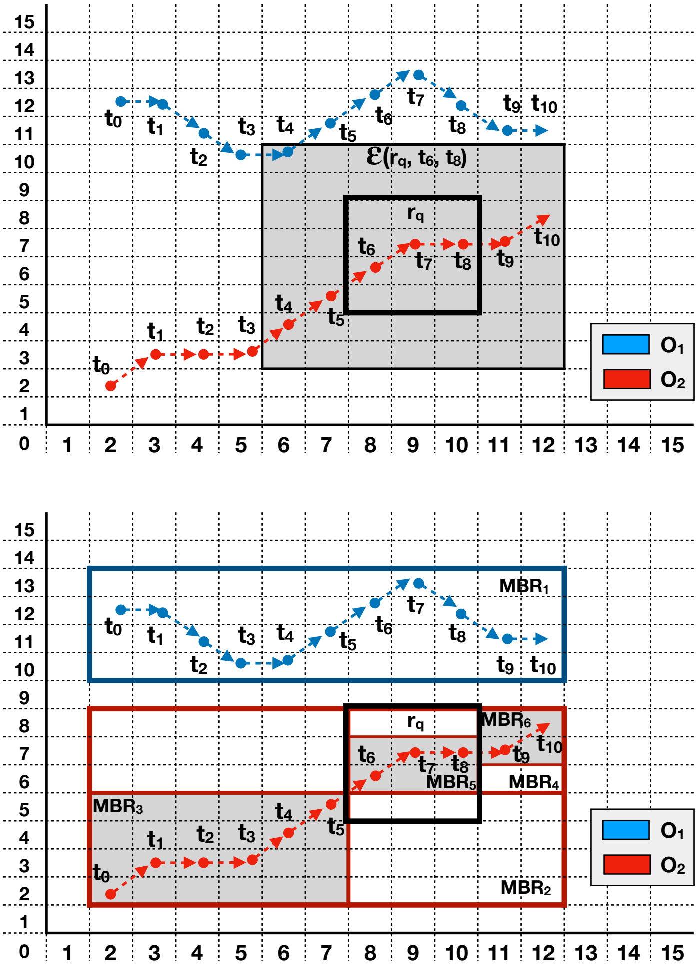


Figure 5.2: Example of the algorithm to solve time interval queries: retrieving the trajectory (top) and binary search through the MBRs (bottom).

- $MBR \subseteq r_q$. The object is within r_q during the whole interval $[t_b, t_e]$, thus the current candidate is part of the solution. The algorithm continues with the next candidate and adds this object to the solution.

- $MBR \cap r_q = \emptyset$. Since there is no intersection with r_q during the whole interval, the object has no chances to be within r_q. Therefore, this object is discarded, and the algorithm continues with the next candidate.

- $MBR \cap r_q \neq \emptyset$. Though there is an intersection, the current candidate could be outside r_q, thus the process continues recursively with the two halves of the queried range $[t_b, t_m]$ and $[t_{m+1}, t_e]$, where t_m is the time instant in the middle, i.e. $t_m = \lceil \frac{t_b+t_e}{2} \rceil$.

At the bottom of Figure 5.2, we show how to compute the time interval query during the time interval $[t_0, t_{10}]$ and with the queried region $r_q = [8, 5] \times [10, 8]$. Our candidates are the objects O_1 and O_2. Firstly, we compute the MBR of O_1 in the queried interval (MBR_1). We can observe that MBR_1 does not intersect r_q. Therefore that object is discarded as part of the solution, and the algorithm continues with O_2. Since its MBR (MBR_2) during the queried interval contains r_q, it starts the binary search through the MBRs. That is, MBR_2 is split into $MBR_3 = [2, 2] \times [7, 5]$ and $MBR_4 = [8, 6] \times [12, 8]$. MBR_3 does not cover any part of r_q, hence the algorithm stops breaking it. Instead, MBR_4 intersects with r_q, and it is divided in two parts: $MBR_5 = [8, 6] \times [10, 7]$ and $MBR_6 = [11, 7] \times [12, 8]$.

Notice that the binary search involves the time interval $[t_b, t_e]$. That is the main difference with the previous approach, where the chances are checked with respect to the last instant of the sub-interval. Consequently, with the binary search of the MBRs, once an object is candidate in a snapshot, it will not be longer considered in the remaining snapshots.

5.3.3 K-Nearest Neighbor

K-Nearest Neighbor or *KNN* is a spatio-temporal range query that returns the K closest neighbors (objects) to a given point p_q at a time instant t_q. That is, it computes the distance from p_q to the K-th nearest object, and returns the objects in a radius lower or equal to that distance. In order to obtain the result, the algorithm uses a priority queue of *candidate nodes* with objects from the closest snapshot (Q_c), and a priority queue of *best known results* (Q_r).

Before explaining how to solve this query, we define the minimum and maximum Euclidean distance, necessary to discern which objects are closer to p_q in the space.

Definition 5.3.4. *We denote with* $\overline{pr}$ *the minimum Euclidean distance between the region* $r = [x_1, x_2] \times [y_1, y_2]$ *and the point* $p = (x, y)$.

Definition 5.3.5. *We denote with* $\overline{\overline{pr}}$ *the maximum Euclidean distance from the region* $r = [x_1, x_2] \times [y_1, y_2]$ *to the point* $p = (x, y)$.

Q_c is a min-priority queue that recollects the nodes of the chosen snapshot prioritized by their proximity to p_q. Notice that all the structures of the snapshots are trees whose nodes contain some spatial data. For example, in the snapshot based on k^2-trees the internal and leaf nodes represent regions and cells of the space, respectively. Instead, in those snapshots based on R-trees, each leaf stores the MBR of the trajectory of an object during the corresponding interval of time. Therefore, each internal node stores a MBR that wraps the MBRs of its children. Consequently, every node, independently of the implementation of the snapshot, corresponds with a region of the space (of size 1×1 in case of cells) and we can prioritize all of them by their proximity to p_q.

In order to compute that proximity, we define the minimum and maximum reachable Euclidean distance, that is, the minimum and maximum Euclidean distance that an object can achieve with respect to a point at a time instant.

Definition 5.3.6. *The minimum reachable Euclidean distance, denoted as $d_{min}(r_i, t_i, p_q, t_q)$, is the Euclidean distance of the objects within r_i at t_i can be with respect to the queried point p_q at the queried time instant t_q.*

Definition 5.3.7. *The maximum reachable Euclidean distance, $d_{\max}(r_i, t_i, p_q, t_q)$ is analogous to the previous definition, but instead of the minimum Euclidean distance they compute the maximum Euclidean distance.*

Notice that the main difference of these definitions with respect to Definitions 5.3.4 and 5.3.5 is that the Definitions 5.3.6 and 5.3.7 consider that the object can move during the interval time $[t_i, t_q]$.

The minimum and maximum reachable distance depends on the snapshot's implementation. For example, if the snapshot is based on an R-tree and the node contains r_1 as its MBR, the objects are moving within r_1 during an interval that includes t_q. Therefore, the minimum and maximum reachable distance to p_q is the minimum and maximum Euclidean distance between r_1 and p_q, that is, they can be computed as $\overline{r_1\ p_q}$ and $\overline{\overline{r_1\ p_q}}$, respectively.

On the other hand, in the case of a snapshot that is based on a k^2-tree, the region r_1 does not correspond with any trajectory. Therefore we have to assume that the included objects move at the maximum speed of the dataset during the interval $[t_h, t_q]$. Consequently, we expand r_1 to $r_e = \mathcal{E}(r_1, t_h, t_q)$, where t_h is the time instant of the snapshot. That expanded region corresponds with the area where the objects within r_1 can be moving during the interval $[t_h, t_q]$. Hence, the minimum and maximum reachable distance to p_q can be computed as $\overline{r_e\ p_q}$ and $\overline{\overline{r_e\ p_q}}$, respectively.

For example in Figure 5.3 we show $d_{\min}(r, 0, p_q, 2)$ and $d_{\max}(r, 0, p_q, 2)$. Let us assume that the chosen snapshot corresponds with time instant 0, the region of a node is $r = [3, 4] \times [4, 5]$, the maximum speed is $M_s = 1$, and $p_q = (8, 3)$. Hence, the objects within r could be moving during the interval of time $[0, 2]$ to any part of r_e, that is, the resultant region of expanding the region of the node $(2 - 0) \times 1 = 2$ cells in all directions. Consequently, the closest position to p_q that an object within r can reach will be the nearest point of r_e to p_q. In that example, the closest point is $(6, 3)$, and its Euclidean distance to p_q is equivalent to $d_{\min}(r, 0, p_q, 2) = \sqrt{(8-6)^2 + (3-3)^2} = 2$. Analogously, $d_{\max}(r, 0, p_q, 2)$ is computed with respect to the furthest cell $(1, 7)$ of r_e concerning to p_q.

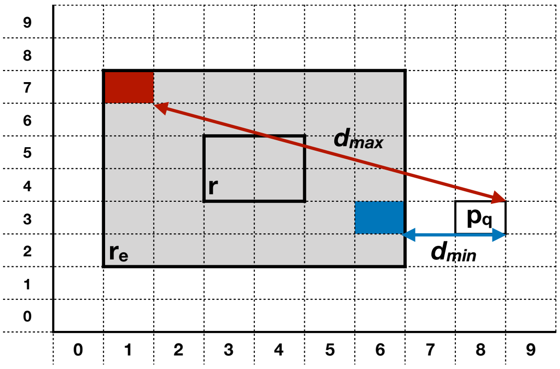


Figure 5.3: Example of minimum and maximum reachable distance on snapshots based on k^2-trees.

Therefore, prioritizing the nodes of Q_c by its proximity to p_q means sorting them in ascending order by using its minimum reachable Euclidean distance, thus the closest nodes to p_q are on top of Q_c. The ties between two nodes are broken with the maximum reachable distance.

Assuming that $\mathcal{S}_h$ is the previous snapshot[1] to t_q, the algorithm starts adding the root of $\mathcal{S}_h$ to Q_c. Then, the procedure continues retrieving the element on top of Q_c, that is the closest one to p_q. Depending on the type of node, the algorithm chooses between the following steps:

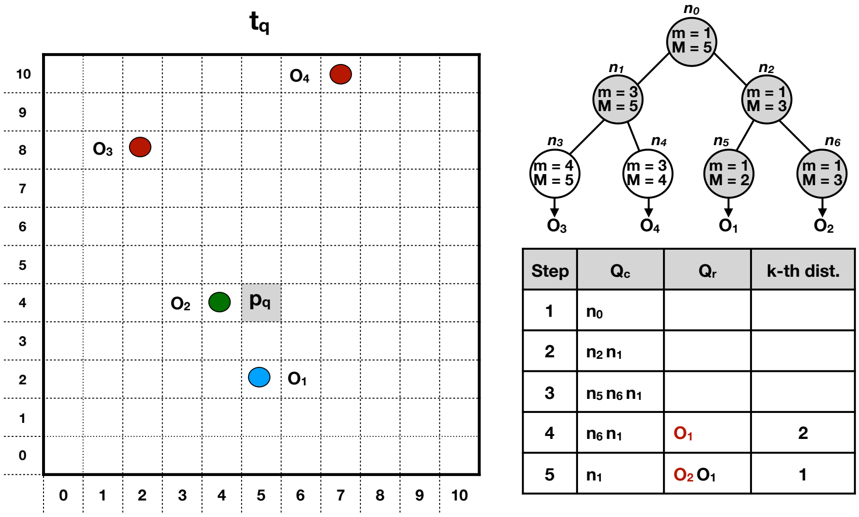
- If it is an internal node, it is removed from Q_c and its children are added to Q_c, considering the minimum and maximum reachable Euclidean distance.

- Otherwise, if it is a leaf node, the algorithm computes the position of its objects at t_q, and they are added to Q_r.

Q_r is a max-priority queue that stores objects sorted by their distances to p_q. The maximum size of this queue is K, that is, once Q_r contains K elements, every time a new object is added, the farthest is deleted. This queue can compute in constant time the distance to p_q of the K-th closest object (the last element of the queue), which decreases as the search progresses.

Therefore, the algorithm can stop in two cases: (i) when Q_c is empty, there are no more objects to check; or (ii) the candidate on top of Q_c cannot improve the distance of the K-th element of Q_r, that is, the best candidate cannot improve the current solution of Q_r.

[1] When the closest snapshot corresponds to a time instant after t_q, the process is analogous but backwards.



Step	Q_c	Q_r	k-th dist.
1	n_0		
2	n_2 n_1		
3	n_5 n_6 n_1		
4	n_6 n_1	O_1	2
5	n_1	O_2 O_1	1

Figure 5.4: Example of KNN query with K=1 and the followed steps.

To illustrate the algorithm, the left part of Figure 5.4 shows the location of a set of objects at t_q and a point p_q. On top of the right part, we observe a conceptual tree of a snapshot where each node has two values: m and M. m is the minimum reachable Euclidean distance from the region represented with that node, and M corresponds with the maximum reachable Euclidean distance.

Following the steps in the right part of Figure 5.4, we can compute the KNN of that point with $K = 1$. Firstly, n_0 is added to Q_c, since it is the root and the only node on Q_c it is located on top of Q_c. In the second step, n_0 is retrieved and split into n_2 and n_1. Once n_1 and n_2 are added to Q_c, n_2 is on top because its minimum reachable Euclidean distance is lower than its corresponding value to n_1 ($n_2.m < n_1.m$). In the next step, n_2 is divided into n_5 and n_6. Both have the same priority, but in our case, we decide to process the first n_5 because of the maximum Euclidean distance ($n_5.M < n_6.M$). Since it is a leaf and contains the object O_1, we compute the corresponding position of O_1 at the queried time instant t_q. The distance between its location and p_q is 2, thus it is added to Q_r with priority 2. As it turns out the current solution to our query, we track its distance. We look if the distance of the recently added object (since we are computing $K = 1$) can be improved by checking the minimum reachable Euclidean distance of the object on top of Q_c. In that example, the node on top is n_6, since $n_6.m$ is 1 and it is lower than the distance between O_1 and p_q, the next node to process is n_6. This node only contains the object O_2, thus its position at t_q is computed. The distance between O_2 and p_q is 1, moving it to the top of Q_c and removing O_1. Consequently, the K-th distance is 1, and it cannot be improved because the first node of Q_c has a minimum reachable Euclidean distance ($n_1.m = 3$) greater than 1. Hence, the algorithm stops and returns the solution $\{O_2\}$.

Chapter 6

Snapshots

In this chapter, we present the different structures designed for the representation of snapshots. All of them assume a raster model that splits the space into cells of a fixed size. That is, the space can be understood as a matrix, and the coordinates of a trajectory are mapped into their corresponding cells. The size of those cells can be established depending on the domain, notice that the smaller the cells are, the higher the precision we obtain.

This chapter is divided into Sections 6.1 and 6.2 that introduce our two implementations of the snapshots: based on k^2-tree and based on R-trees. In each section, we explain both, the structure and the required algorithms to compute the queries of Chapter 5.

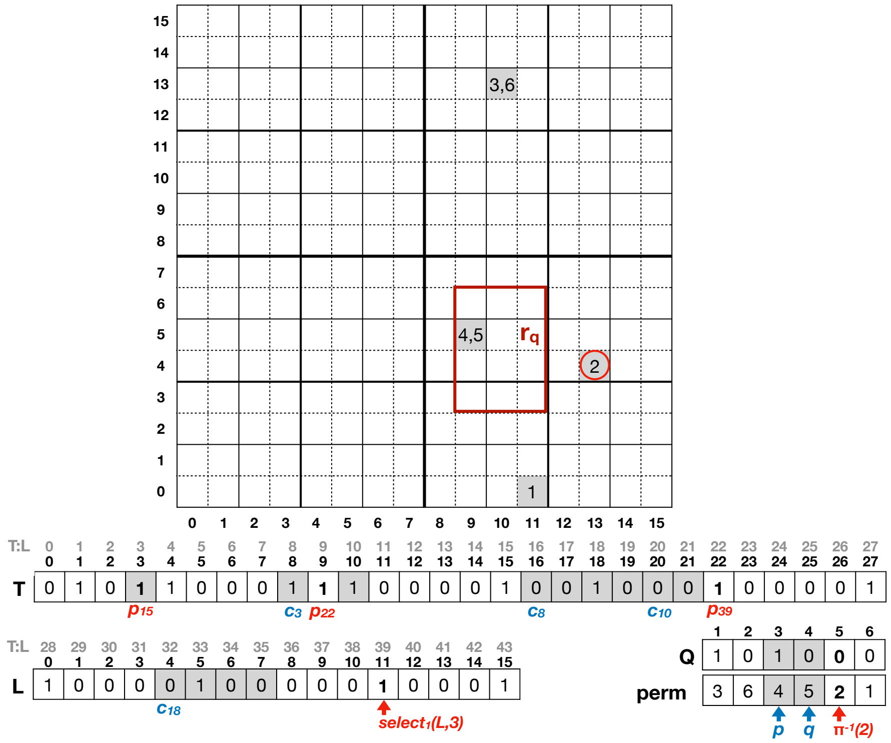
6.1 Snapshot based on k^2-tree

The k^2-tree represents a binary matrix where a cell set to 1 indicates that the cell contains one or more objects. Recall that the k^2-tree exploits the clustering and sparsity of that matrix by recursively splitting into k^2 parts those submatrices containing information. Therefore, the nodes of the navigable tree represent regions of the space, and their leaves refer to a specific cell, which can contain objects or not. In case of the existence of objects, each 1-bit is associated with the corresponding leaf of its cell. Below we present the data structure and the algorithms to compute the spatial operations.

6.1.1 Data structure

Let $\mathcal{S}_h$ denote the snapshot representing the position of all the objects at time instant t_h. The components of $\mathcal{S}_h$ are:

- The *time instant* represented by the snapshot, in this case t_h, a multiple of d.



T:L	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
T	0	1	0	1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1

T:L	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1

	1	2	3	4	5	6
Q	1	0	1	0	0	0
perm	3	6	4	5	2	1

Figure 6.1: Example of snapshot and the different operations that supports. The indexes marked as $T : L$ denote the indexes of concatenating the bitmaps T and L.

- A k^2-tree storing the positions in the space (i.e., the cells of the raster) where there are objects.
- An array of integers *perm* storing the identifiers of the objects at each cell.
- A bitmap Q, which is an auxiliary data structure of *perm* that is used to determine the correspondence between positions in the space and the positions of *perm*.

Recall that the k^2-tree represents a binary matrix where a cell set to 1 indicates that the cell contains one or more objects. Although the k^2-tree provides a fast way to retrieve the cells with objects, we need to know the identifier of those objects. Each 1 in the binary matrix corresponds to a bit set to 1 in bitmap L of the k^2-tree. The object identifiers corresponding to every cell that have at least one object are stored in *perm* according to their order of appearance in L. This array turns out to be a permutation of all the object identifiers. Bitmap Q is aligned with *perm*; a 0 at a given position of Q means that the object identifier aligned in *perm* is the last object in the cell, while a 1 marks that the next object identifier in *perm* is in the same cell.

Figure 6.1 shows an example. On top of this figure, we can observe the matrix representing the space and the objects placed in the cells (these cells are the same containing a 1 in Figure 2.14). The four arrays at the bottom are the actual data structures that represent the snapshot information. Arrays T and L are the same as those in Figure 2.14 and mark the positions having objects. The object identifiers are stored in arrays Q and *perm*. For instance, the object identifiers corresponding to the second 1 in L (which is at position 5 in L and corresponds to the cell at position (9,5)) are stored starting at position 3 in *perm*, because the first 0 (signaling the end of the entries corresponding to the first 1 in L) is at position 2. Since the last object inside this cell is stored in the position 4 of *perm*, $Q[4] = 0$, the object identifiers corresponding to the third 1 in L start at position 5 in *perm*, and so on. Therefore, the identifiers of those objects inside (9,5), i.e. 4 and 5, are stored at positions 3 and 4 in *perm*.

6.1.2 Queries

As we explained in Section 2.2, the k^2-tree supports efficient navigation from a parent node to its children and vice-versa. With the help of this type of snapshot, we can solve the queries presented in Chapter 5. Below, we explain the different algorithms designed in the snapshot to support those queries.

6.1.2.1 Object queries: obtaining the absolute position

Recall that queries like *Object Position*, *Object Trajectory*, and *MBR* need to obtain the absolute position where an object is in the closest snapshot. Considering a snapshot $\mathcal{S}_h$, to obtain the absolute position of the object, the algorithm starts by using $\pi^{-1}(id)$ operation over *perm*, that obtains the position i in *perm* where the identifier id is located. Then the leaf of the k^2-tree corresponding to O_{id} is computed with the help of the bitmap Q. Since Q is aligned with *perm* and marks with 0 the last object of a leaf of the k^2-tree, the number of leaves with objects before the object in position i of *perm* can be computed as $y = rank_0(Q, i-1)$. Consequently, the position of L corresponding to O_{id} is the $(y+1)$-th 1; that is, $select_1(L, y+1)$. With that position in L, we can traverse the k^2-tree upwards in order to obtain the position in the space of that cell, and thus the position of the object.

For example in Figure 6.1, we illustrate how to obtain the position of the object with identifier 2 (O_2). We start computing its location in *perm* with $\pi^{-1}(2) = 5$. By using $rank_0(Q, 5-1) = 2$, we know that there are two leaves with objects before the leaf of O_2, therefore it is stored into the third cell with objects, and it corresponds with the 11-th leaf ($select_1(L, 3) = 11$). Finally, with a bottom-up traversal of the k^2-tree from the 11-th leaf up to the root, we can compute the cell containing O_2. That traversal requires of $O(\log s)$ time, where $s \times s$ is the size of the matrix that represents the space. In Figure 6.1, p_a is the position in T of the 1-bit corresponding to the parent of the node whose position in $T : L$ is a, which can be computed as $select_1(T, \lfloor a/k^2 \rfloor)$. The values in bold are accessed during that traversal.

Notice that, this operation is the only one that requires a bottom-up traversal of the k^2-tree, and requires an additional space of $(1 + \epsilon)m \log m + O(m)$ bits to solve π^{-1} in $O(1/\epsilon)$ time, where m is the number of objects that contains the dataset. In case that this operation is not required, the structure does not need to store the additional space for π^{-1}.

6.1.2.2 Time Slice and Time Interval: choosing the candidates

Recall that *Time Slice* and *Time Interval* queries need to obtain the object candidates that can be within the queried region r_q during a specific time instant or interval to avoid the sequential scan of the logs of all the objects. In order to obtain those candidates, we need to compute $reach(\mathcal{S}_h, r_q, t_q)$, where $\mathcal{S}_h$ is the chosen snapshot, and t_q the considered time instant.

Computing $reach(\mathcal{S}_h, r_q, t_q)$ depends on the type of snapshot, in order to solve it on a snapshot based on k^2-tree, we use Definition 5.3.1, that is, the expanded region. We have seen that those objects that are not contained within $\mathcal{E}(r_q, t_h, t_q)$ at time instant t_h have no chances to be within r_q at t_q; otherwise, they should move faster than the maximum speed of the dataset. Consequently, the candidates of these queries can be computed by obtaining those objects that are within the region $\mathcal{E}(r_q, t_h, t_q)$ at t_h. For example, given $r_q = [10, 4] \times [10, 5]$, $|t_h - t_q| = 1$ and $M_s = 1$, the maximum distance that an object can move from t_h to t_q is only one cell. Therefore, the expanded region is $[9, 3] \times [11, 6]$, and includes objects that can be at r_q after one time instant.

To retrieve the information within a region from this snapshot, we first run a top-down traversal in the tree, from the root until we reach those positions in L corresponding to the queried area. Let us assume that, after the traversal, the algorithm reaches only one leaf at position j in L. We compute the number of leaves of objects up to the j-th leaf with $x = rank_1(L, j)$. Then, we compute the position of the last bit of the previous leaf with objects, the $(x-1)$-th 0 in Q, and we add 1 to obtain the first position in our current leaf. That is, $p = select_0(Q, x - 1) + 1$ is the position in *perm* of the first object identifier corresponding to the searched position. Similarly, the last position of our leaf can be computed as $q = select_0(Q, x)$. Since Q is aligned with *perm*, the objects' identifiers are from position p to q in *perm*.

Figure 6.1 shows the steps followed to compute the objects within region $r_q = [9, 3] \times [11, 6]$. Notice that the cells colored in gray correspond with the checked positions. The top-down traversal starts at position 3 in T, because its corresponding submatrix $[8, 0] \times [15, 7]$ contains r_q. Since $T[3] = 1$, we compute the positions of the k^2 children of the node at $T[3]$. That operation is denoted as c_a, and computes $rank(T, a) \times k^2$ where a is the position of the node in T. In our case we compute $c_3 = 8$. There are two nodes that overlap r_q whose positions in T are 8 and 10; notice that, their corresponding submatrices are $[8, 4] \times [11, 7]$ and $[8, 0] \times [11, 3]$. Then, c_8 and c_{10} are computed, and their children intersecting r_q are checked. In that case, only the node at position 18 has objects. We reach the leaves with c_{18}, and just one leaf contains objects, that at position 5 of L. By computing $rank_1(L, 5) = 3$, we know that this leaf is the third cell with objects. Finally we compute $p = select_0(Q, 2) + 1 = 3$ and $q = select_0(Q, 3) = 4$, hence we know that our result are those values in the range $[3, 4]$ of *perm*; that is, the object identifiers 4 and 5.

Observe that when $t_q = t_h$ the operation is solved directly in the snapshot, without the use of an extended region ($r_q = \mathcal{E}(r_q, t_h, t_q)$). Consequently, when $t_q = t_h$, time-slice queries do not need to traverse the log because the solution corresponds with the objects retrieved from $reach(\mathcal{S}_h, r_q, t_q)$.

6.1.2.3 K-Nearest Neighbor: prioritizing the objects

To solve *KNN* queries, the nodes of the snapshot are collected in a priority queue Q_c, taking into account the minimum and maximum possible distance of each node. Those distances correspond with the Definitions 5.3.6 and 5.3.7, respectively. That is, given a snapshot $\mathcal{S}_h$ and a node n_i of $\mathcal{S}_h$ whose corresponding region is r_i, we can compute the minimum and maximum reachable distances of the objects to p_q at time instant t_q with $d_{\min}(r_i, t_h, p_q, t_q)$ and $d_{\max}(r_i, t_h, p_q, t_q)$.

This kind of snapshots does not store the trajectory of the objects. Thus we have to assume that the objects are moving at the fastest speed of the dataset. Therefore, those objects that are within r_i, can only be moving within $r_{exp} \leftarrow \mathcal{E}(r_i, t_h, t_q)$, otherwise the object would be moving faster than the maximum speed. Therefore, in the case of the minimum, the result is the distance between the expanded area r_{exp} and p_q, that is, $\overline{r_{exp}\ p_q}$, whereas the maximum possible distance is computed as $\overline{\overline{r_{exp}\ p_q}}$.

For example in Figure 6.1 assuming we are processing the node that corresponds with the region $r_i = [8, 8] \times [15, 15]$, $p_q = (13, 4)$, $t_h = 0$, $t_q = 1$ and $M_s = 1$, let us consider the operation $d_{\min}(r_i, 0, (13, 4), 1)$. Firstly, it computes $r_{exp} \leftarrow \mathcal{E}(r_i, 0, 1) = [7, 7] \times [16, 16]$, the area where any object within r_i can be moving. Then, it returns $\overline{r_{exp}\ p_q} = 3$ because from the closest point in r_{exp}, $(13, 7)$, with respect to p_q the distance in cells is 3. On the other hand, $d_{\max}(r_i, 0, (13, 4), 1) = \sqrt{6^2 + 11^2} \simeq 12.53$ cells, the Euclidean distance from $(13, 4)$ to $(7, 15)$, the furthest point of r_{exp} to $(13, 4)$.

Note that $d_{\min}(r_i, t_h, p_q, t_q)$ is equivalent to $\overline{r_i p_q}$ when $t_h = t_q$. That feature makes it possible to improve the KNN queries. Recall that two priority queues are used, Q_c that contains the nodes of the snapshot, and Q_r, the known results. An object is added to Q_r after retrieving its position at t_q, and computing its distance to

p_q. Since the snapshot stores the location of the object at t_h, obtaining the distance when $t_h = t_q$ is straightforward. When a leaf is reached, the position p_i of the object is known, and it can be directly added to Q_c with the distance set to $\overline{p_i p_q}$.

6.2 Snapshot based on R-tree

The classical *R-tree* stores the locations of the objects to compute an imprecise extended region, and uses the Minimum Bounding Rectangles for representing the spatial distribution of those objects. Recall that it is a tree, where the MBR of each internal node wraps the MBRs of their children, instead, each leaf node contains the MBR of several objects. All the MBRs of the tree are chosen following a heuristic that minimizes the size of the MBRs.

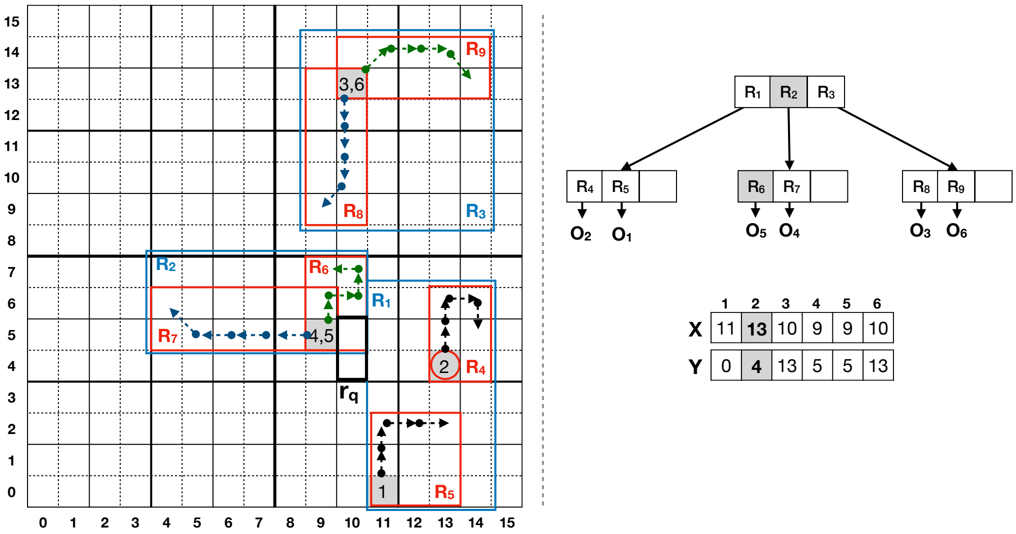
Unlikely our snapshot based on k^2-tree that only stores the position of the objects at the corresponding time instant of the snapshot, the snapshots based on *R-tree* store the MBR that encloses the trajectory of an object between the time instants of the cited snapshot and the next one. More precisely, since snapshots are created every d time instants, a snapshot $\mathcal{S}_h$ based on *R-tree* stores the MBRs of the trajectories of the objects in the interval $[t_h, t_{h+d}]$. Therefore, it avoids to consider the maximum speed of the dataset when computing spatio-temporal range queries, providing a more accurate area where the object can be moving during the interval of time $[t_h, t_{h+d}]$. In the following sections, we show the complete data structure and the algorithms for solving the spatial operations.

6.2.1 Data structure

Since the *R-tree* stores MBRs that wrap the trajectory of objects, we cannot obtain the position of any object at a specific time instant of the trajectory, including the time instant t_h. Therefore, we need to add some additional structures to the *R-tree* to specifically store the absolute positions of the objects at t_h. Consequently, every snapshot $\mathcal{S}_h$ is composed of:

- The *time instant* represented by the snapshot, in this case t_h, a multiple of d.
- A static and compressed R-tree [BLNS13] storing the MBR of the trajectories of each object during the interval of time $[t_h, t_{h+d}]$.
- The absolute positions of those objects at t_h are stored into two arrays of integers X and Y, for the displacement on the horizontal and vertical axis, respectively.

Figure 6.2 shows an example. In the left part, we can observe the location of the objects of Figure 6.1 and their trajectory during the following four time instants until the next snapshot. All those trajectories are surrounded by a rectangle that



	1	2	3	4	5	6
X	11	**13**	10	9	9	10
Y	0	**4**	13	5	5	13

Figure 6.2: Example of snapshot based on R-tree at time instant t_h.

represents their $MBRs$. The superposed rectangles involving more than one MBR belong to the first levels of the *R-tree*. For example R_1 encloses the $MBRs$ of objects O_1 and O_2. The locations of the objects at t_h are stored in X and Y. That is, the first entry of X and Y corresponds to the first object, the second to the second object, and so on. Therefore, the location of the object O_2 at t_h is stored at $X[2]$ and $Y[2]$, that is $(13, 4)$.

6.2.2 Queries

The use of this type of snapshot in the different queries of Chapter 5 is explored in the followings sections.

6.2.2.1 Object queries: obtaining the absolute position

Notice that the queries that belong to the group of object queries need to obtain the absolute position of an object from the closest snapshot. Once we figure out the closes snapshot, we retrieve the absolute position from X and Y. Since the locations are contiguously stored into those two arrays, and the identifiers of the objects index them, we know that the id-th entry in X and Y corresponds with O_{id}, thus the solution is the point $(X[id], Y[id])$.

For example, in Figure 6.2 we know that the absolute position of O_2 at t_h is $(X[2], Y[2]) = (13, 4)$.

Notice that the arrays X and Y require of $O(m \log s)$ bits, where $s \times s$ is the size of the matrix and m the number of objects of the dataset. As we will see later, when we use some structures of log we do not need to compute the absolute position in the snapshot, because the log supplies that information. Since X and Y are only focused on storing the absolute positions of the objects in each snapshot, those types of logs can save those $O(m \log s)$ bits.

6.2.2.2 Time Slice and Time Interval: choosing the candidates

With the operation $reach(\mathcal{S}_h, r_q, t_q)$, we can obtain the objects that have chances to be within r_q at t_q. This operation is necessary in order to obtain the objects that have chances to be part of the solution of *Time Slice* and *Time Interval* queries. Since our snapshots based on R-trees keep MBRs that enclose every individual trajectory during $[t_h, t_{h+d}]$, we have to choose the snapshot that stores the trajectories that include t_q. In that snapshot, each object whose MBR of its trajectory intersects with r_q has chances to be within it. Therefore, by running a top-down traversal following the nodes of the *R-tree* that overlap with r_q, we get the objects that intersect that area.

In Figure 6.2, the traversal of the tree starts with the second node of the first level because it is the only one that intersects r_q ($R_2 \cap r_q \neq \emptyset$). The algorithm continues with the children of R_2, but only R_6 intersects r_q ($R_6 \cap r_q \neq \emptyset$). Since R_6 only contains object O_5, it is the solution of the operation.

Notice that this technique to select candidates is more accurate than extending the region, as we did in the snapshots based on k^2-tree, because it takes into account the trajectory of each individual object. This particular change makes it possible that the snapshots based on *R-tree* yield a better prune of the objects that can reach r_q at t_q.

As in the case of snapshots based on k^2-tree, when $t_q = t_h$, the snapshots based on R-tree that retain X and Y can solve time slice queries without using the log. Thus, once the objects whose trajectories intersect with the queried region are obtained, for each of them, its position at $t_q = t_h$ is computed in the snapshot. With that information, the algorithm adds to the solution the objects within r_q.

6.2.2.3 K-Nearest Neighbors: prioritizing the objects

Recall that to solve *KNN* queries we have to prioritize the nodes of the closest snapshot $\mathcal{S}_h$ to t_q by its proximity to p_q in a min-priority queue Q_c, but taking into account that the objects included in that node can move closer to p_q during the interval of time $[t_h, t_q]$. To consider that proximity we compute the operations $d_{\min}(r_i, t_h, p_q, t_q)$ and $d_{\max}(r_i, t_h, p_q, t_q)$ for each node of the snapshot, where r_i is the region of that node. Since r_i corresponds to the MBR that encloses its children's trajectories during the interval of time $[t_h, t_{h+d}]$ and $t_q \in [t_h, t_{h+d}]$, the minimum and maximum reachable Euclidean distance can be easily computed as $\overline{r_i\ p_q}$ and $\overline{\overline{r_i\ p_q}}$.

This type of snapshot obtains a better approximation that the k2-tree based counterpart, especially when the difference between t_q and t_h is significant. For example, in Figure 6.2, with $p_q = (10, 5)$, $t_q - t_h = 4$ and $M_s = 1$, with the k2-tree-based approach, the minimum distance for O_4 and O_5 is 0, because their expanded region, $[5, 1] \times [13, 9]$, overlaps r_q. However, with this new approach the minimum distance $\overline{R_7\ p_q} = 1$ for O_4. Therefore O_4 has less probabilities to be within r_q than O_5 whose minimum reachable Euclidean distance is $\overline{R_6\ p_q} = 0$.

With this technique, KNN queries when $t_q = t_h$, can be sped up. Recall that, in these queries, the solution is composed of the K-th first objects of a priority queue Q_r where the objects are sorted by its minimum distance to p_q at t_q. When $t_q = t_h$ that distance is directly obtained from the information stored in X and Y. This allows adding the objects directly to Q_r without needing to check the log, hence improving this kind of query. Notice that, in case of not storing X and Y, such improvement is not possible.

Chapter 7

Logs

The log is the second element of our structure. We design four different types of log structures and for each of them we provide the necessary algorithms to solve the queries presented in Chapter 5. Each log corresponds with an individual object, and represents its relative movements along time. Therefore, each log can be considered as a sequence of values where each element represents the displacement of an object from a position to the next one. Consequently, given the location of an object, we can compute the next position by applying its corresponding relative movement. Additionally, with the log, we can retrieve the actual location of all the objects, at the time instants not covered by snapshots.

In this chapter, we propose four data structures to represent the log. All of them support the presented queries, but they have different features related to compression and time performance, as we explain in the next sections. Section 7.1 presents a structure that assigns fewer bits to shorter movements by using (s,c)-Dense Codes. The technique explained on Section 7.2 shows how to exploit the repetitiveness of movement patterns by using grammar compression. A technique that is focused on time performance instead of on compression, and is based on a partial-sums structure is introduced in Section 7.3. Finally, Section 7.4 illustrates a structure whose main goal is to obtain a good space/time tradeoff by using relative compression.

7.1 ScdcCT

By considering the concept of the first law of geography [Tob70], which states that “near things are more related than distant things”, we can assume that short movements are more common than long displacements. *ScdcCT* tries to exploit this fact by encoding the movements with a variable-length code that assigns shorter or longer codewords depending on how long is the movement. That is, shorter movements require fewer bits. Although this technique is simple, it turns out to be a good starting point for obtaining compression on trajectories.

7.1.1 Data structure

The relative movements in a log can be represented with two integers that denote the displacement of the object in each dimension. To save space, *ScdcCT* encodes the two values with one positive integer by using the spiral encoding representation, which was presented in Chapter 4.

The result of that encoding is a sequence of integers representing the movements, which at the same time are encoded with (s, c)-Dense Codes. Recall that (s, c)-Dense Codes need a first phase to compute the frequencies. We avoid that step by assuming that the shortest movements are more frequent, thus short movements require fewer bits than longer displacements.

Let us consider the trajectory of an object O_{id} as the list of points $p_0, p_1, \ldots, p_n$ where p_i is the position of the object at time instant i. $\mathcal{L}_{id}$ stores $m_1, m_2, \ldots, m_n$ where m_i is the relative movement from p_{i-1} to p_i, and they are represented with the spiral encoding and encoded with (s, c)-Dense Codes.

As the snapshots are distributed every d time instants, we can figure out that $\mathcal{L}_{id}$ is split into many sections, delimited by the snapshots. That is, the first section of the log corresponds with the movements from time instant 1 to d, the second from $d + 1$ to $2d$, and so on. For solving the queries of Chapter 5, we need to access the closest snapshot and traverse the corresponding section of the log.

Since the log of each object is as unique sequence and each entry has variable-length, we need a mechanism to access the first and last entry of the log between two snapshots. Therefore, we store $idx = [l_0, l_1, \ldots, l_k]$ where each l_j is a pair that stores the position in $\mathcal{L}_{id}$ that contains the previous and posterior entry of the log from a snapshot. Hence, we can access to the first or last position of the log between two consecutive snapshots in constant time without requiring to traverse the log from the start to the previous or posterior position of the desired snapshot.

7.1.2 Object queries

7.1.2.1 Object Position

In order to compute the position of an object we have to take the absolute position of that object stored in the closest snapshot $\mathcal{S}_h$, and add the cumulative movement from the snapshot to the queried time instant t_q. If $\mathcal{S}_h$ is previous to t_q that cumulative movement is computed by traversing the log from the entry that corresponds with t_{h+1} to t_q. Notice that we can compute the entry of t_{h+1} in $O(1)$ time from *idx*.

Let us denote (c_x, c_y) as the current position, and t_c as the current time instant to process. Hence we set $t_c \leftarrow t_{h+1}$ and (c_x, c_y) is initialized with the absolute position from $\mathcal{S}_h$. Since each entry of the log refers to one time instant, every time a new codeword from the log sequence is read, the algorithm repeat these steps:

1. Assuming that CW_{scdc} is the corresponding codeword of (s, c)-Dense Code for t_c, we decode CW_{scdc} to its spiral codeword (CW_{spiral}). Then, the CW_{spiral} is decoded and we obtain the movement (m_x, m_y), where m_x and m_y are the displacements in the horizontal and vertical axis, respectively, which can acquire positive and negative values.

2. By updating $(c_x, c_y) \leftarrow (m_x, m_y) + (c_x, c_y)$, the value of (c_x, c_y) corresponds with the position at t_c.

3. The algorithm continues repeating Steps 1–2 with the next entry, which corresponds with time instant $t_c \leftarrow t_{c+1}$.

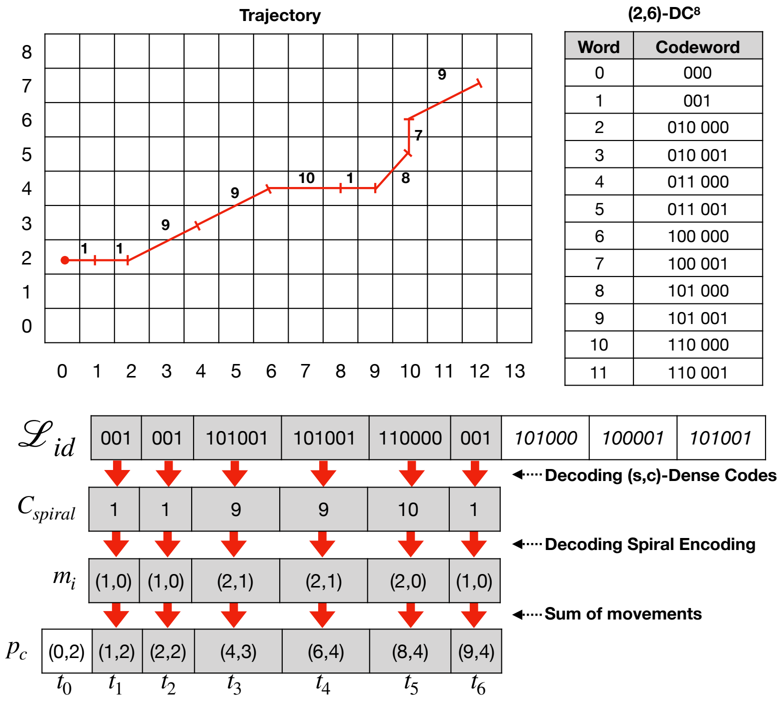
Notice that, after each iteration through Steps 1–2, the point (c_x, c_y) contains the object's position at t_c. Therefore, after $t_q - t_h$ iterations, the desired solution is stored in (c_x, c_y). For example, in Figure 7.1, we can observe a trajectory (left) and the mapping between a word and its codeword for $(2, 6)$-Dense Code, assuming that $b = 3$ (right), that is, the codewords are formed by chunks of 3 bits. On the bottom of both elements, there is the corresponding compressed log ($\mathcal{L}_{id}$), and the position of that object in $\mathcal{S}_h$ is $(0, 2)$. To obtain the position of the object after six time instants, the algorithm decodes the first $CW_{scdc} = 001$ to its corresponding $CW_{spiral} = 1$. Since the spiral codeword corresponds with the movement $(1, 0)$, the position at t_1 is computed as $(c_x, c_y) = (0, 2) + (1, 0) = (1, 2)$. Then the algorithm continues with the next entry of the log 001 and obtains the position $(c_x, c_y) = (1, 2) + (1, 0) = (2, 2)$. The process is repeated until reaching t_6, where the position $(c_x, c_y) = (9, 4)$ is obtained. Besides, comparing the first and third entries of $\mathcal{L}_{id}$, we can observe that short displacements require fewer bits than the long ones, thus obtaining compression.

7.1.2.2 Object Trajectory

In order to obtain the trajectory $\mathcal{T}$ between two time instants t_b and t_e, the algorithm starts by getting the position of that object at t_b, as in the previous explanation. Then the solution $\mathcal{T}$ is initialized with the first position of the trajectory. Since we have scanned the log up to the entry with the corresponding movement to t_b, by reading the next entry, we can obtain the relative movement from t_b to t_{b+1}. We apply it to the previous computed absolute position, and we obtain the absolute position of the object at t_{b+1}. Therefore, we only need to decompress the next entry by using the Steps 1–2 to compute the next position. The solution is obtained by scanning the log and decompressing the entries until the algorithm computes the absolute position at t_e.

7.1.2.3 Minimum Bounding Rectangle

This query requires considering the absolute positions of the trajectory of the object from t_i to t_j. Therefore, the MBR query can be implemented as an *Object*



Word	Codeword
0	000
1	001
2	010 000
3	010 001
4	011 000
5	011 001
6	100 000
7	100 001
8	101 000
9	101 001
10	110 000
11	110 001

Figure 7.1: Example of a log compressed with ScdcCT .

Trajectory query but with a small modification. That is, the algorithm computes the whole trajectory between the interval of time $[t_i, t_j]$, and gathers the minimum and maximum values in each axis. Once all the time instants are processed, the final minimum (min_x and min_y) and maximum (max_x and max_y) values of both dimensions correspond with the resultant MBR: $[min_x, min_y] \times [max_x, max_y]$.

Notice that, computing the trajectory is mandatory to compute the MBR. Therefore, it requires a linear traversal of the log, thus calculating the MBR takes $O(d)$ time, where d is the distance between snapshots.

7.1.3 Spatio-temporal range queries

7.1.3.1 Time Slice

As we presented before, to solve *Time Slice* queries, the algorithm has to compute the location of an object at t_q and check if it is within the queried region r_q. This requires computing the cumulative movement from the time instant of the snapshot, which provides the absolute position, until t_q. Therefore, a linear traversal of the log is necessary. To avoid checking all the logs of the objects, the algorithm starts computing the candidates with chances to be within r_q at t_q, as we explained in Section 5.3.

During the traversal of the log $\mathcal{L}_{id}$, we can detect objects that are moving away from the queried region. Therefore, at some point, the object can loose all the chances to be within r_q at t_q. For this reason, every time a new location is computed, the algorithm checks $chance(p_c, t_c, r_q, t_q)$, where p_c is the position of the object at t_c. If *chance* returns false, the object has no chances to reach r_q at t_q and the algorithm stops processing that object. With this mechanism, we avoid decompressing all the entries up to t_q, when the object has no possibilities to be within r_q.

7.1.3.2 Time Interval

Regarding *Time Interval* queries, for simplification, we assume that the queried interval $[t_b, t_e]$ is between two snapshots. In other case, as we explained in Section 5.3, the solution is computed as the union from the partial solutions of each interval of time between two snapshots.

After selecting the candidates in the closest snapshot, the algorithm simulates retrieving the trajectory of each candidate. However, obtaining the whole trajectory is not necessary in some cases. After computing each position p_c at time instant t_c in the trajectory, we check two conditions:

- If p_c is within the queried region r_q, the algorithm adds that object to the solution, and stops processing its log.

- As in *Time Slice*, we detect if an object cannot reach r_q, thus we check $chance(p_c, t_c, r_q, t_e)$. In case there are no chances, the object cannot be in the solution and the algorithm stops processing its log.

After processing the logs of all the candidate objects we will have obtained the solution.

7.1.3.3 K-Nearest Neighbors

In KNN queries, every time an object is obtained from the priority queue of candidates Q_c, we have to compute its position at the queried time instant t_q. Once it is computed, the object is added to the priority queue of known results Q_r. The algorithm continues until Q_r contains k elements and the first candidate of Q_c cannot improve the results of Q_r. As we can observe, in this query, the log is only used to compute the position at t_q, which is obtained as we explained in Section 7.1.2.1.

7.2 GraCT

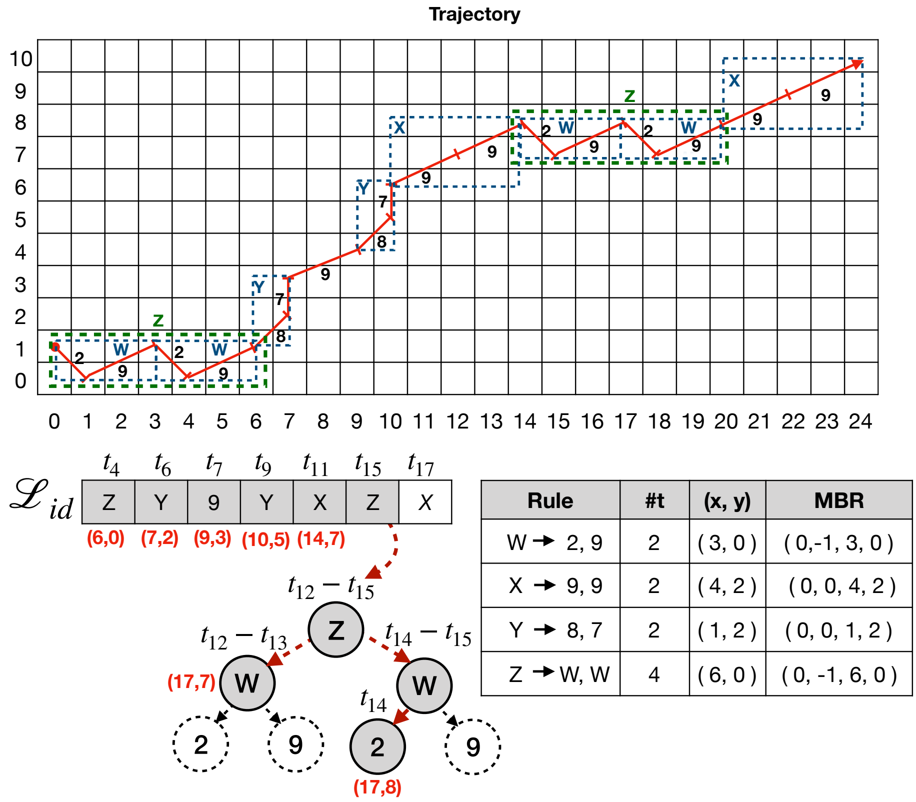
In many applications, objects spend most of their time either stopped or moving along a specific course at a fixed speed. These circumstances generate long sections of the log with numbers representing the same or contiguous values if we use the spiral encoding. For example, the moving object in Figure 7.2 follows a NE trajectory, moving one or two cells in the time elapsed between two consecutive time instants. The log represents the series of relative movements 2,9,2,9,8,7,9,8,7,9,9,2,9,2,9,9,9. These series of similar movements are compressed very efficiently using a grammar compressor. Indeed, it is particularly useful when many objects usually follow the same movements. This is the basis of *GraCT* that compresses all the log sequences with RePair [LM00]. In contrast to *ScdcCT*, *GraCT* can skip the decompression of some movements when computing queries by storing additional information along the rules of the grammar; consequently, it gets better performance in most cases.

7.2.1 Data structure

As in ScdcCT, this structure transforms all the movements of each object in a sequence of integers of the spiral encoding. Then, that sequence of integers is compressed by using Re-Pair. As a result of that compression, we obtain a sequence where every pair of symbols appears only once, and that sequence is composed of two types of symbols: *terminals* and *nonterminals.* Each terminal is an original integer, and each nonterminal represents a list of movements. Since the nonterminals are built by substituting pairs of symbols, each nonterminal contains at least two movements, but it can include more since the symbols of a pair can be nonterminals as well, and this can continue recursively. Therefore, each terminal corresponds with only one time instant, and each nonterminal includes at least two time instants. In addition, each nonterminal has a grammar rule associated, which identifies the two symbols that compose it. By recursively expanding those grammar rules over each nonterminal, we can obtain the original trajectory.

In order to query trajectories without completely decompressing them, the grammar rules in *GraCT* not only include the symbols to be replaced, but they are also enriched with additional information. Specifically, each rule in R has the following information: $s \rightarrow a, b, \#t, (x, y), mbr$, where:

- $s \rightarrow ab$ is the normal rule of Re-Pair.
- $\#t$ is the number of time instants covered by the rule.
- (x, y) are the relative coordinates of the final position of the object after the application of the rule. That is, (x, y) will be the location of the object after applying the movements of the nonterminal when the previous position is $(0, 0)$. Notice that (x, y) can acquire positive and negative values.



Rule	#t	(x, y)	MBR
W → 2, 9	2	(3, 0)	(0,-1, 3, 0)
X → 9, 9	2	(4, 2)	(0, 0, 4, 2)
Y → 8, 7	2	(1, 2)	(0, 0, 1, 2)
Z → W, W	4	(6, 0)	(0, -1, 6, 0)

Figure 7.2: Example of a log compressed with GraCT .

- *mbr* is the minimum bounding rectangle enclosing the movements of the rule. *mbr* is represented as $(x1, y1, x2, y2)$, where $(x1, y1)$ corresponds to the bottom-left corner and $(x2, y2)$ to the top-right corner. As in the previous explanation, we assume that the position preceding the nonterminal is $(0, 0)$.

For example, the rules in Figure 7.2 are enriched as follows. The first rule is $W \rightarrow 2,\ 9,\ 2,\ (3,0),\ (0,-1,3,0)$: 2 and 9 are two terminal symbols that are substituted by the nonterminal W; the next 2 indicates that W represents a sequence of two movements; $(3, 0)$ indicates the position of the object after the application of the rule if we start at $(0, 0)$, and the last four values are two corners (bottom-left and top-right) defining a rectangle that encloses all the movements encoded by the nonterminal.

These additional elements that enrich the rules are compressed with DACs. To obtain better compression, the times of all of the rules are compressed using one

DAC, separately from the three pairs of coordinates of all of the rules, which are compressed using another DAC.

As in ScdcCT, since the log sequence is contiguous and the entries have no fixed length, this representation stores the indexes $idx = [l_0, l_1, \ldots, l_k]$, where each l_j is a pair with the previous and following entry of the log related to the j-th snapshot. That information is necessary to access in constant time to the first or last entry of a log section between two snapshots, avoiding scanning previous parts in order to locate the desired entry.

7.2.2 Object queries

7.2.2.1 Object Position

.

After obtaining the absolute position from the closest snapshot, the algorithm for computing the position of an object at a given time instant computes the cumulative movement. As in *ScdcCT*, it needs to traverse the log to accumulate the relative movements. In GraCT, each nonterminal can cover more than one movement. That is, to retrieve each movement, we need to decompress the nonterminals.

However, with the additional information, most of the nonterminal symbols of a compressed log $\mathcal{L}_{id}$ do not need to be decompressed in order to obtain the cumulative movement of an object. Actually, we only need to decompress the nonterminal at the end of the queried time interval.

Consider, for example, the compressed log $\mathcal{L}_{id} = Z, Y, 9, Y, X, Z, X$ of Figure 7.2. Assume that we want to find the position of the object at t_{14}, that is, $(18, 7)$. Firstly, we inspect the log $\mathcal{L}_{id}$ from the beginning. The first value is a Z. The enriched rule indicates that this symbol represents four time instants, thus it covers from time instant t_1 to t_4. Since $t_{14} > t_4$ the cumulative movement is initialized to $(6, 0)$, which is the relative movement of the nonterminal Z. The algorithm continues with the next entry Y, which lasts two time instants up to t_6. As $t_{14} > t_6$ and the relative coordinates of Y are $(1, 2)$, the algorithm updates the cumulative movement to $(7, 2)$. The third entry is the terminal value 9, ant it is completely covered in the queried interval. It corresponds with moving two positions to the East and one to the North in the spiral encoding. Since every terminal involves only one time instant, the cumulative movement is updated to $(9, 3)$ at t_7. Then, the next entry is Y, with the enriched information, we know that it lasts two time instants between t_8 and t_9. As that interval does not surpass t_{14}, the relative coordinates are added to the cumulative movement, that is, $(9, 3) + (1, 2) = (10, 5)$. This process is repeated until the current log entry surpasses our last target time instant; in this example, it occurs at the sixth entry, with the cumulative movement set to $(10, 5) + (4, 2) = (14, 7)$. That entry would take us to t_{15}, and our target is t_{14}. Therefore, we have to decompress the rule and process its components: $Z \rightarrow W\ W$. The first W is a nonterminal that lasts two time instants, containing information about the position at t_{12} and t_{13}. As $t_{14} > t_{13}$, we use the enriched information from W and move the object three positions to the East, getting the value $(17, 7)$ at t_{13}. Then, the second W is checked, and we can observe that surpasses t_{14} because it contains information from t_{14} and t_{15}. Hence, we need to decompress W, its first value 2 is a terminal symbol that lasts one time instant, and thus it is enough to reach our target time instant. The terminal moves the object one position to the East, and one to the South, which applied to the current cumulative movement $(17, 7)$ computes the cumulative movement $(18, 6)$ between t_0 and t_{14}.

Once the cumulative movement is obtained, the position at t_{14} is computed by adding the cumulative movement to the absolute position of the closest snapshot. In this case, the closest snapshot corresponds with t_0 and the absolute position is $(0, 1)$. Therefore, the position at t_{14} is $(18, 6) + (0, 1) = (18, 7)$.

Notice that unlikely ScdcCT, this structure allows us to skip more than one time instant every time it reads a nonterminal symbol. As a consequence, in practice, the traversal of the log in GraCT is faster than in ScdcCT.

7.2.2.2 Object trajectory

In this kind of queries, the advantage of reading an entry from the log and advance more than one time instant cannot be exploited. Notice that, to solve the trajectory of an object during an interval of time $[t_b, t_e]$, we have to compute all the involved movements. Therefore, the algorithm needs to decompress all the nonterminals included in that range of time. Thus, the algorithm starts by obtaining the position at t_b, as we explained in the previous section.

The next position at t_{b+1} is computed by looking for the next leaf in the grammar parse tree. Since the value of the leaf is a terminal, it corresponds with a relative movement that spans only one time instant. That movement is added to the previous computed absolute position, obtaining the position at t_{b+1}. Those steps are repeated until the computed position corresponds with t_e. Therefore, we have to decompress all the entries of the log that contain the information of the interval $[t_b, t_e]$.

Notice that all those entries have to be fully decompressed except the last one. In case that the time instants covered by the last nonterminal are not fully contained in the queried interval, we only have to partially decompress it. For example, in the parse grammar tree of Figure 7.2, let us assume that the queried interval of time is $[t_0, t_{13}]$ and that we have processed until the time instant t_{11}. The next and last entry corresponds with Z. We decompress $Z \rightarrow W, W$. Since the first W covers $[t_{12}, t_{13}]$ and it is completely contained into the queried interval, W is decompressed into 2 and 9. Those terminals give us the positions at time instants t_{12} and t_{13}, which are $(15, 6)$ and $(17, 7)$, respectively. As t_{13} is the last time instant of the queried interval, we do not need to decompress the second W from Z.

7.2.2.3 Minimum Bounding Rectangle

Recall that each nonterminal rule stores its *mbr*, but it is relative to the position of the object that precedes the nonterminal. Therefore, the actual MBR will be the addition of the previous position (p_x, p_y) and *mbr*. On the other hand, the MBR of each terminal can be computed as $[p_x, p_y] \times [p_x + m_x, p_y + m_y]$, where (m_x, m_y) is the displacement corresponding to that terminal. With this two formulas, we can obtain the actual MBR of each entry of the log.

To obtain the MBR in the interval $[t_b, t_e]$, we consider the minimum subsequence of entries of the log that contains $[t_b, t_e]$. Assuming that no terminal intersects with the limits, the global MBR can be computed as the union of the MBRs of the symbols of the minimum subsequence.

When an intersection with a nonterminal occurs at the limits, we have to consider only the part of those nonterminals covered by $[t_b, t_e]$. Therefore, we decompress it with the parse grammar tree, considering only those subtrees that include some part of $[t_b, t_e]$. Notice that, when we find a node that is completely included in $[t_b, t_e]$, its MBR contains the MBRs of its children too, thus we do not need to continue decompressing its children.

For example, in Figure 7.2, to compute the MBR between t_6 and t_{14}, we access the snapshot at time instant 0 retrieving the location of the object $(0, 1)$, and then we traverse the log up to the symbol covering t_6 (first Y) to compute the position $(7, 3)$ at t_6. The result of this operation is initialized to $res \leftarrow [7, 3] \times [7, 3]$. The next entry is a terminal that contains the spiral codeword 9 (1 North and 2 East), as it lasts one time instant, we know that the object at t_7 is in position $(9, 4)$. As the current result does not cover $(9, 4)$ we have to enlarge the top-right corner of res, updating it to $res \leftarrow [7, 3] \times [9, 4]$. The next entry covers a nonterminal completely included in $[t_6, t_{14}]$, whose MBR is $[9 + 0, 4 + 0] \times [9 + 1, 4 + 2] = [9, 4] \times [10, 6]$, thus updates $res \leftarrow [7, 3] \times [10, 6]$.

These steps are repeated while any entry of the log intersects with the queried interval. In our case, it occurs in the sixth entry and reaches it having $res \leftarrow [7, 4] \times [14, 8]$. We traverse the parse grammar tree of that nonterminal Z, and we observe that the first child of the root is within $[t_6, t_{14}]$. Specifically, it covers the interval $[t_{12}, t_{13}]$ and its nonterminal is W. Therefore, we compute its MBR as $[14 + 0, 8 + (-1)] \times [14 + 3, 8 + 0] = [14, 7] \times [17, 8]$, that expands the result to $res \leftarrow [7, 4] \times [17, 8]$. Finally, the second child of the root is processed, and it is decompressed because it intersects the queried interval. The terminal 2 corresponds with t_{14} and moves one position to the South and one position to the East. Thus the object reaches the position $(18, 7)$. Consequently, the final result is updated to $res \leftarrow [7, 4] \times [18, 8]$.

7.2.3 Spatio-temporal range queries

7.2.3.1 Time Slice

Time Slice requires to compute the position at t_q of the candidates. As we saw above, it requires processing the log. As in ScdcCT, every time we read an entry from the log the position p_c at time instant t_c is computed. Therefore, we can check if the object still has chances to be within the queried region r_q. In case that $chance(p_c, t_c, r_q, t_q)$ does not hold, the algorithm stops processing the log of that object.

In addition, when te movement at t_q is stored within a nonterminal in $\mathcal{L}_{id}$ that corresponds with t_q, we can avoid the decompression of that nonterminal in some cases. Assume that we traverse the log, and our current entry contains the movement of t_q. With the addition of p_k and the *mbr* associated with that nonterminal, we compute the MBR during an interval of time $[t_i, t_j]$ that includes t_q. The nonterminal has to be decompressed only when that MBR intersects with r_q because in the remaining cases:

- $MBR \subseteq r_q$, MBR is within r_q, that is, the object is within r_q during the whole interval $[t_i, t_j]$. Since $t_q \in [t_i, t_j]$, the object is part of the solution.

- $MBR \cap r_q = \emptyset$, MBR does not cover any part of r_q; thus the object is outside r_q during $[t_i, t_j]$. Since $t_q \in [t_i, t_j]$, O_{id} is discarded as a possible solution.

7.2.3.2 Time Interval

In *Time Interval*, when the queried time interval involves several snapshots, we split the queried time interval $[t_b, t_e]$ into subintervals delimited every two snapshots. Consequently, the global solution corresponds with the union of each partial solution from those subintervals, as we explained in Section 5.3. For simplification, we assume that the queried interval only involves one snapshot.

Therefore, the algorithm starts obtaining the candidates from that snapshot. For each candidate object, we check with the log if it is in the queried region r_q at any time instant of the queried interval of time. To achieve it, we simulate to retrieve the trajectory and, for each point, we check if it is contained in r_q. If the object is within r_q at any of these points, we can stop scanning its log. In addition, we can stop processing the log of an object, when $chance(p_c, t_c, r_q, t_e)$ does not hold, that is, we know that the object cannot reach r_q.

Also, we can skip the decompression of several nonterminals as in *Time Slice*. In each entry of the queried interval of time $[t_i, t_j]$, the MBR is computed. When the MBR intersects r_q, the nonterminal has to be decompressed. Otherwise, we skip the decompression of the MBR because either it is fully contained in r_q, thus the object is part of the solution and we stop processing it; or it does not intersect r_q, and we have to process the next entry.

7.2.3.3 K-Nearest Neighbors

In KNN queries, every time an object is on top of Q_c, we have to compute the position of the object at the queried time instant t_q and add it to the priority queue of known results Q_r. The algorithm stops when there is no candidate that can improve the results of Q_r and Q_r has k elements. As we can observe, in this query, the log has the function of computing the position at t_q, as we show in Section 7.2.2.1.

7.3 ContaCT

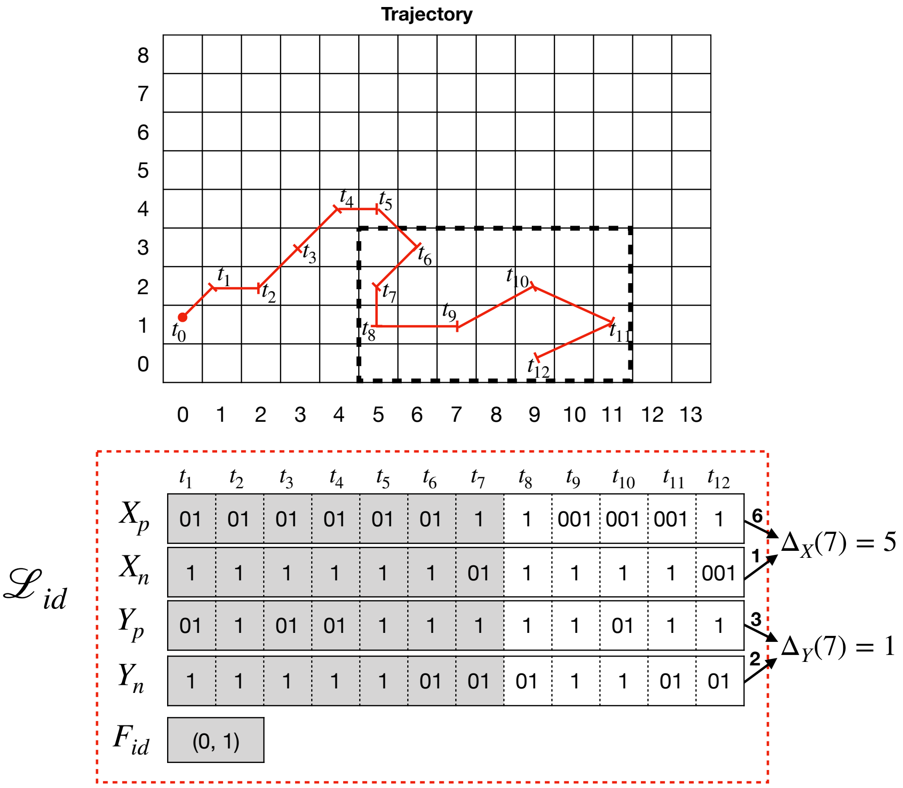
ContaCT aims to compute the position of an object at a given time instant t_q in constant time. Thus, it uses a representation of the log based on a partial sums structure, which we replicate in every direction (North, South, West, and East). Consequently, we can compute the cumulative movement from the beginning until a time instant in constant time, by using *select* operation over bitmaps. Notice that the cumulative movement must be added to a previous absolute position p_h in order to obtain the position at t_q. In the previous structures, the absolute position that is added to the cumulative movement is obtained from the closest snapshot, but it requires a traversal of the tree that takes logarithmic time. In order to keep the constant time, *ContaCT* stores the first position of each object, therefore we can compute the desired position by adding that first position to the cumulative movement.

Another important new feature of *ContaCT* is that it can rapidly compute the *Minimum Bounding Rectangle* (MBR) between two time instants. The result yields an approximate description of the movements. Additionally, computing the MBR allows us to speed up some spatio-temporal queries like *time interval.*

7.3.1 Data structure

By using bitmaps equipped with a *select* auxiliary structure, it is possible to build a data structure to store integers that allows the computation in constant time of the partial sum of all the numbers from the first one until any position i. Given a list of values $0 < x_1 < x_2 < ... < x_n$, we define the differences $d_i = x_i - x_{i-1}$, and $d_1 = x_1$. If we store the differences $d_1, d_2, \ldots, d_n$, it is easy to see that x_i can be computed by adding the differences from d_1 until d_i. That is $x_i = \sum_{j=1}^{i} d_j$. An Elias-Fano representation of the partial sums is a bitmap $B[1..n]$, where $B[x_i] = 1$, $1 \leq i \leq n$, and 0-bit in the rest. In other words, that bitmap is the list of the values d_i represented in unary. Therefore, we can retrieve x_i as $select_1(B, i)$, which can be computed in constant time.

In *ContaCT*, we use a small variation of the above partial sums structure, which allows us to represent differences that are 0. In our case, the value of a difference $d_i = v$ is represented as v consecutive zeros followed by a 1 to mark the end of the number. Therefore, the value x_i can be obtained as $select_1(B, i) - i$ because the



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
X_p	01	01	01	01	01	01	1	1	001	001	001	1
X_n	1	1	1	1	1	1	01	1	1	1	1	001
Y_p	01	1	01	01	1	1	1	1	1	01	1	1
Y_n	1	1	1	1	1	01	01	01	1	1	01	01
F_{id}	(0, 1)											

Figure 7.3: Example of a log compressed with ContaCT .

searched value corresponds to the number of zeros until that position, and that number is the position of the i-th 1 minus the number of 1s until there.

Based on this approach, we can represent the displacements through each axis with two bitmaps. First, we define *positive displacements* and *negative displacements* as those movements that increase or decrease the value of a specific axis. Consequently, the bitmap X_p (resp. Y_p) stores the positive displacements and X_n (resp. Y_n) the negative displacements through the horizontal (resp. vertical) axis. For example, in Figure 7.3, the movement from t_5 to t_6 corresponds to $(1, -1)$. On the horizontal axis, its positive and negative displacements are 1 and 0; that is, "01" is added to X_p and "1" to X_n. Concerning the vertical axis, there is no positive displacement, and the negative one is 1. Therefore, Y_p adds "1" and Y_n appends "01". In case of the object is at a static position on an axis, as in the movement between t_7 and t_8 where the object does not move on the horizontal axis, it is represented appending "1" to X_p and "1" to X_n. Notice that F_{id} stores the first location of the object.

7.3.2 Object queries

7.3.2.1 Object Position

The main goal of ContaCT is to compute the location of an object at t_q in constant time. Therefore, ContaCT avoids the use of the snapshots to obtain the absolute position. Instead, it uses its information stored in F_{id}. Consequently, we have to compute the cumulative movement from the initial time instant up to t_q.

We know that the horizontal axis's positive and negative movements are stored in X_p and X_n, respectively. The partial sum of the displacement after t_q movements in each bitmap, X_p or X_n can be computed in constant time as $\delta(X_p, t_q) = \mathit{select}_1(X_p, t_q) - t_q$ and $\delta(X_n, t_q) = \mathit{select}_1(X_n, t_q) - t_q$. Hence, the cumulative movement until t_q on the horizontal axis is $\Delta_X(t_q) = \delta(X_p, t_q) - \delta(X_n, t_q)$. If the value of $\Delta_X(t_q)$ either is positive or negative, it corresponds with movements to the right or left, respectively. Notice that we can simplify $\Delta_X(t_q) = \mathit{select}_1(X_p, t_q) - \mathit{select}_1(X_n, t_q)$, and, in the same way, we can compute $\Delta_Y(t_q)$ for the vertical axis. Therefore, the cumulative movement corresponds with $(\Delta_X(t_q), \Delta_Y(t_q))$.

Recall that this operation is the basis of the others. Since it does not require obtaining the absolute position from the snapshot, all the snapshots used with *ContaCT* uses its minimum-space setup, that is, the snapshots based on the k^2-tree do not use the additional structure for efficiently solving π^{-1}, and those based on the R-tree save the space of X and Y.

In Figure 7.3, we can observe the computation of Δ_X and Δ_Y for $t_q = 7$, that is, the movement on both axis until the time instant 7. Firstly, the values of δ are computed. Therefore, to compute $\delta(X_p, 7)$ we count the number of zeroes until the seventh 1-bit; that is 6. In the same way, we compute $\delta(X_n, 7) = 1$, and the subtraction of those two values is $\Delta_X(7) = 6 - 1 = 5$. Analogous we can compute $\Delta_Y(7) = \delta(Y_p, 7) - \delta(Y_n, 7) = 3 - 2 = 1$. Consequently, the cumulative movement up to the time instant 7 is $(5, 1)$. The absolute position could be computed as the addition of that cumulative movement and the initial position of the object stored at F_{id}, in this example, the position at time instant 7 is $(0, 1) + (5, 1) = (5, 2)$.

7.3.2.2 Object Trajectory

In order to obtain the cells where is an object during an interval of time $[t_b, t_e]$, we start by computing the position at t_b as we explained above. Then, we have to compute the locations in the interval time $[t_{b+1}, t_e]$. Recall that each individual movement is computed, and added to the previous location in order to obtain the next location. Therefore, to compute t_{b+1}, we have to compute the next movement and add it to the position at t_b.

Notice that every time the next movement is computed, its cumulative movement until the previous time instant is known, thus for each axis, the last select positions in its positive and negative bitmaps are known (i_p and i_n). Notice that all of them correspond with a 1-bit. Therefore, the displacement in an axis is the number of 0-bits between the index (i_p or i_n) and the next 1-bit. Since the next 1-bit after the index i_p can be computed as $select_{next}(X_p, i_p+1)$, the displacement on the x-axis of the next movement is $dis_x = (select_{next}(X_p, i_p+1)-(i_p+1))-(select_{next}(X_n, i_n+1)-(i_n+1))$. In the same way, we calculate dis_y, the displacement in the vertical axis, and the retrieved movement is (dis_x, dis_y).

For example, in Figure 7.3, let us compute the position at time instant t_8, considering that the position at time instant t_7 was computed. The last selected indexes from the vertical axis were $i_p = 10$ and $i_n = 9$. Therefore, the next displacement in the vertical axis corresponds with $(select_{next}(Y_p, 11) - 11) - (select_{next}(Y_n, 10) - 10) = (11 - 11) - (11 - 10) = -1$. Since the object does not move in the horizontal axis, the next movement is $(0, -1)$, that is, one position to the South. By adding it to the previous location (at time instant t_7), we obtain the position at time instant t_8: $(5, 2) + (0, -1) = (5, 1)$.

7.3.2.3 Minimum Bounding Rectangle

ContaCT can efficiently compute the MBR of a trajectory, that is, the smallest axis-aligned rectangle that contains every point visited by the object O_{id} during a given time interval $[t_b, t_e]$. This is an interesting operation since it provides summary information about the path followed by an object without computing the whole trajectory and, in addition, it is used as a tool to compute time interval queries efficiently.

A good option for solving the MBR in constant time would be building an *rMq* and *rmq* structure on both axes. Therefore, by computing where is the maximum (resp. minimum) on both axis and computing their position with the cumulative movement and the initial location; we retrieve the top-right (resp. bottom-left) corner of the MBR.

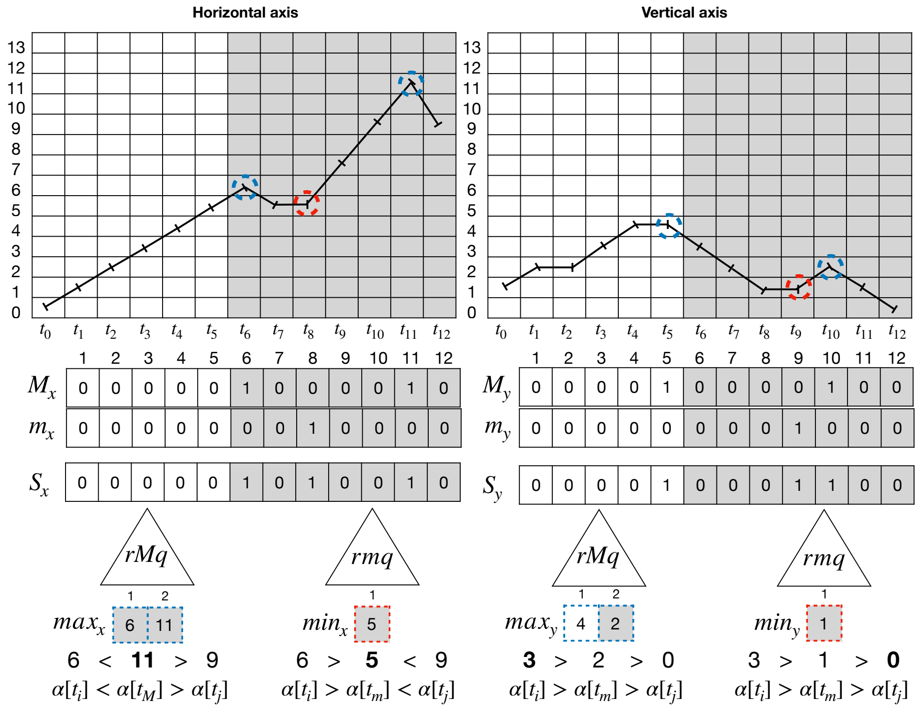
However, we notice that most of the trajectories follow a constant heading through its course. Whether we analyze each axis's values independently, we can find a lower number of local maxima and minima, comparing them with the number of movements within the trajectory. Notice that knowing the maximum (max_{local}) and minimum (min_{local}) from the local values for an axis $C \in \{x - axis, y - axis\}$ between $[t_b, t_e]$, we can compute the values of the corners of the MBR on that axis C during the interval $[t_b, t_e]$ as:

$$global\_min_C = min(min_{local}, C[t_b], C[t_e]) \tag{7.1}$$

$$global\_max_C = max(max_{local}, C[t_b], C[t_e]) \tag{7.2}$$

, where $C[t_b]$ and $C[t_e]$ are the value of the object on the axis C at the extremes of the interval and *min* (resp. *max*) is a function that computes the minimum (resp. maximum) value.

Considering Equations 7.1 and 7.2, we can reduce the space required by the *rmq* and *rMq*. Using *rmq* and rMq through the arrays of the local values (min_C


Horizontal axis
Vertical axis
13
12
11
10
9
8
7
6
5
4
3
2
1
0
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
rMq
rmq
max_x 6 11
min_x 5
max_y 4 2
min_y 1
6 < **11** > 9
$\alpha[t_i] < \alpha[t_M] > \alpha[t_j]$
6 > **5** < 9
$\alpha[t_i] > \alpha[t_m] < \alpha[t_j]$
3 > 2 > 0
$\alpha[t_i] > \alpha[t_m] > \alpha[t_j]$
3 > 1 > **0**
$\alpha[t_i] > \alpha[t_m] > \alpha[t_j]$

	1	2	3	4	5	6	7	8	9	10	11	12
M_x	0	0	0	0	0	1	0	0	0	0	1	0
m_x	0	0	0	0	0	0	0	1	0	0	0	0
S_x	0	0	0	0	0	1	0	1	0	0	1	0

	1	2	3	4	5	6	7	8	9	10	11	12
M_y	0	0	0	0	1	0	0	0	0	1	0	0
m_y	0	0	0	0	0	0	0	0	1	0	0	0
S_y	0	0	0	0	1	0	0	0	1	1	0	0

Figure 7.4: Example of a log compressed with ContaCT .

and max_C) and two additional bitmaps, m_C and M_C, marking the time instants when the local minimums and maximums occur in each axis, we can know the corresponding time instants for min_{local} and max_{local} (t_m and t_M).

In Figure 7.4, we can observe the evolution of the values on the horizontal axis (top-left) and the vertical axis (top-right) for the trajectory of Figure 7.3. Below those plots the values of the local maximums are stored in max_x, that is, the values 6 and 11. The time instants that correspond with that local values are marked in bitmap M_x at positions 6 and 11. An rMq structure is built on the max_x array, to know which of them is the local maximum in a given range. Analogously, we can build the structure for the minimums. Notice that for the vertical axis, this structure is replicated.

In order to obtain t_m between t_b and t_e, firstly, we find the range of local minimums of min_C contained in that interval, that is, $[i, j] = [rank_1(m_C, t_{b-1}) + 1, rank_1(m_C, t_e)]$. Then, a $rmq(min_C, i, j)$ operation in that range retrieves the position with the smallest value in the array min_C, and its corresponding time instant t_m can be computed as $select_1(m_C, rmq(min_C, i, j))$. Analogously, we can compute t_M, for example, to compute t_M in the range $[t_6, t_{12}]$, firstly the algorithm looks the range of local maximums between t_6 and t_{12} by using $rank_1(M_x, 5)+1 = 1$ and $rank_1(M_x, 12) = 2$. Then $rMq(1, 2)$ obtains the maximum of the locals on the range $[1, 2]$, that is, the second local maximum. Therefore, t_M corresponds with the time instant of the second local maximum, which can be computed as $select_1(M_x, 2) = 11$.

It is important to notice that rMq and rmq do not store the values of the max_C and min_C; for this reason in Figure 7.4, the borders of those values are dashed. Therefore, we can reduce Equations 7.1 and 7.2 to:

$$global\_min_C = min(C[t_m], C[t_b], C[t_e]) \tag{7.3}$$

$$global\_max_C = max(C[t_M], C[t_b], C[t_e]) \tag{7.4}$$

, those $C[t]$ values are obtained in constant time, as we explained before using the log of Figure 7.3. Once we can compute any $global\_min_C$ and $global\_max_C$ for each axis, the MBR is $[global\_min_x, global\_min_y] \times [global\_max_x, global\_max_y]$. In the bottom of Figure 7.4 we show the needed comparisons for computing the MBR during the interval of time $[6, 12]$, that is, $[5, 0] \times [11, 3]$.

To further save n bits of space, where n is the size of the trajectory, we exploit the fact that local maximums and minimums must alternate: we replace M_C and m_C by a single bitmap S_C, with $S_C[i] = 1$ iff $M_C[i] = 1$ or $m_C[i] = 1$ (see Figure 7.4). If the first bit of S_C came from M_C, then $t_M = select_1(S_C, 2i-1)$ and $t_m = select_i(S_C, 2i)$, and the other way if the first bit of S_C came from m_C.

7.3.3 Spatio-temporal range queries

7.3.3.1 Time Slice

In this kind of queries, after obtaining the candidates from the closest snapshot, the log only needs to compute the position of each candidate at the queried time instant t_q. In those structures that use the spiral encoding representation, they check if the object has chances to reach the region every time a position is computed. However, in ContaCT, the position is calculated in constant time; furthermore the algorithm avoids the traversal and checking if the object has chances. That is, the position at t_q is computed for all the objects selected as candidates.

7.3.3.2 Time Interval

Since *ContaCT* can solve the operation MBR in constant time, in *time interval* queries, it uses the algorithm that simulates a binary search looking for an interval of time in $[t_b, t_e]$ whose MBR is contained within the queried region.

7.3.3.3 K-Nearest Neighbors

In KNN queries, every time an object is on top of Q_c, we have to compute the position of the object at the queried time instant t_q and add it to the priority queue of known results Q_r. Therefore, the log only is useful to KNN queries for obtaining the position of the object at t_q, which can be solved as we show in Section 7.3.2.1.

7.4 RCT

We can consider *RCT* as an evolution of *ContaCT* that exploits the high repetitiveness of patterns between the different trajectories but keeping the capability of computing the position of an object in constant time. The log is composed of two parts: an artificial reference and the individual logs. The reference is a representative trajectory consisting of different parts from the set of trajectories, and it is compressed with *ContaCT*. The actual trajectories are compressed considering that reference by using *Relative Lempel-Ziv*. Therefore, each trajectory is composed of z phrases that point to the reference, being z the number of phrases of a LZ77 parsing. Although the reference is represented with the log of *ContaCT*, we need $O(z)$-sized structures on the sequences of phrases in order to speed up the search for the object's location. Additional $O(z)$-sized structures are also necessary to compute the *spatio-temporal* queries.

7.4.1 Data Structure

Though *RCT* is similar to *ContaCT*, there is a new element that is shared between all the objects, the artificial reference. This reference is a synthetic representative trajectory composed of parts from different trajectories. Besides, the reference is compressed by using *ContaCT*.

The actual trajectories are compressed using RLZ, that is, each trajectory is represented with z phrases: $w_1w_2w_3 \dots w_z$. Recall that each phrase is a pair (p_i, l_i) where p_i is a pointer that refers to the first movement of the i-th phrase in the reference, and l_i the length of that phrase. We store the information of the pairs separately, p_i values in an array p, and we mark in the bitmap l the beginning of all the z phrases from a specific trajectory. For example, in Figure 7.5, we can observe a trajectory compressed with respect to a reference by using three phrases $(1, 5), (8, 4), (10, 2)$. Therefore, we store in p values 1, 8 and 10. As the phrases start at 1, 6 and 10, those positions are set to 1-bit in l. As in ContaCT, F_{id} stores the first position of that trajectory.

As the phrases can be pointing to non-consecutive areas of the reference, in order to keep the constant time computation of the position of the object, we need to store, for each w_i, the cumulative movement from the start of the trajectory until the previous movement to w_i. As we can see in Figure 7.5, those values are stored in X and Y respectively. For example, the position before the second phrase

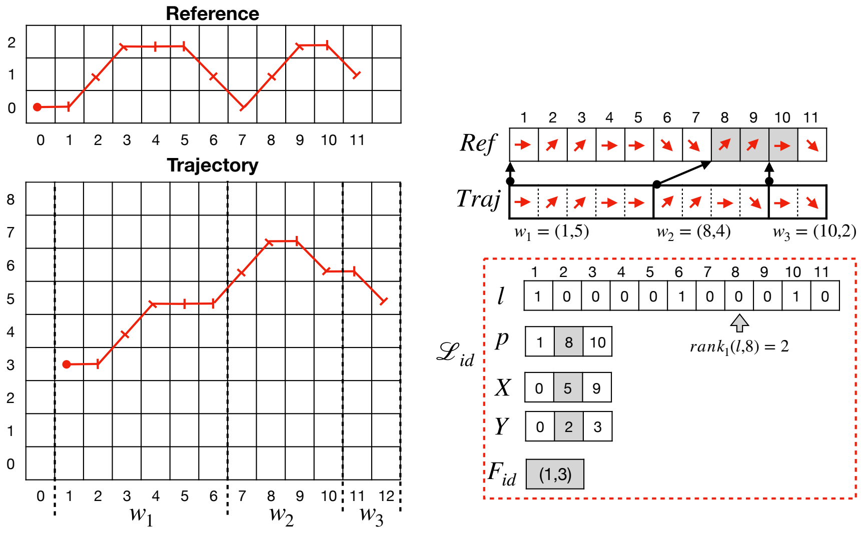


Figure 7.5: Example of a log compressed with RCT .

is $(6, 5)$ and the first known location is $(1, 3)$, thus $(6 - 1, 5 - 3) = (5, 2)$ its the cumulative movement. Therefore, 5 and 2 are stored at the second entry of X and Y, respectively.

7.4.2 Object queries

7.4.2.1 Object Position

Since *RCT* explicitly stores the first location of the trajectory in F_{id}, we can obtain the absolute position from it. As a consequence, the position at t_q is the addition of the first location and the cumulative movement from the first time instant until t_q.

The algorithm starts computing the phrase that contains t_q, which corresponds with the i-th phrase, where i is $rank_1(l, t_q)$. The first time instant of that phrase corresponds with $select_1(l, i)$, we denote it with t_w. Since we store explicitly the cumulative movement until every phrase in X and Y, the previous position to time instant t_w is obtained as $F_{id} + (X[i], Y[i])$. Now, we only need to add the cumulative movement from t_w to t_q. Since $m = t_q - t_w$ and the current phrase points to $p_w = p[i]$ in the reference, the information of that movement corresponds with the relative movement in the interval $[p_w, p_w + m]$ of the reference.

Recall that the reference is stored with ContaCT and we have to compute the cumulative movement from p_w to $p_w + m$ on it. However, in ContaCT every time we compute the cumulative movement, it is done with respect to the first time instant.

Since in RCT p_w can be different to the first movement, we compute the movement from p_w to $p_w + m$ as $(\Delta_X(p_w + m) - \Delta_X(p_w), \Delta_Y(p_w + m) - \Delta_Y(p_w))$.

For example in Figure 7.5 in order to compute the cumulative movement of the object up to the time instant t_8, first we look for the phrase that contains that information. By computing $rank_1(l, 8) = 2$ we know the information is included in w_2. Due to $select_1(l, 2) = 6$, w_2 starts at time instant t_6, and the position at t_5 is $F_{id} + (X[2], Y[2]) = (1, 3) + (5, 2) = (6, 5)$. The queried time instant is t_8, as we know the position at t_5, the first three movements of the chosen phrase will give us the queried position. The second phrase is pointing to the eighth movement at the reference; thus we have to compute the cumulative movement from 8 to 10. Since the cumulative movement is $(3, 2)$ and the previous position to w_2 is $(6, 5)$, the desired position of the object is $(6, 5) + (3, 2) = (9, 7)$.

7.4.2.2 Object Trajectory

Obtaining the trajectory of an object in a time interval $[t_b, t_e]$ is similar to the algorithm of *ContaCT*. We have to compute the location at t_b and from it, with the help of $select_{next}$ operations we compute the next movements.

However, as two consecutive phrases do not correspond to two successive subsequences of the reference, there are two possible ways of computing the next movement. Assume that the previous position was obtained from the i-th phrase. Since that position was computed with the ContaCT structure of the reference, we know for each axis X the values $i_p = select_1(X_p, t_c)$ and $i_n = select_1(X_n, t_c)$, where t_c is the time instant of the previous computed position. In addition, we know that the next phrase starts at time instant $t_w = select_1(l, i + 1)$ and points to the reference at position $p_w = p[i + 1]$. Depending of those values we have to choose between the following steps.

- If the next movement falls within the same phrase, that is, $t_c \neq t_w$. The movement is computed by using the $select_{next}$ operation as in ContaCT. That is, the displacement on the horizontal axis is $dis_x = (select_{next}(X_p, i_p + 1) - (i_p + 1)) - (select_{next}(X_n, i_n + 1) - (i_n + 1))$. In the same way, we calculate dis_y, the displacement in the vertical axis, and the retrieved movement is (dis_x, dis_y). By the addition of (dis_x, dis_y) to the previous position, the algorithm obtains the position at t_{c+1}. To prepare i_p and i_n to the next iteration, they are updated to $i_p = select_{next}(C_p, i_p + 1)$ and $i_n = select_{next}(C_n, i_n + 1)$, where C is the corresponding axis.

- If $t_c = t_w$, the movement is stored in the next phrase. Therefore, the movement will be stored at position p_w on the reference. As i_p and i_n can refer to a movement different to the previous one to p_w, we have to synchronize the indexes i_p and i_n of ContaCT in both axis. To achieve it, we compute, for each axis C, $i_p = select_1(C_p, p_w - 1)$ and $i_n = select_1(C_n, p_w - 1)$, that is, the previous information to p_w. Once those indexes are synchronized, the

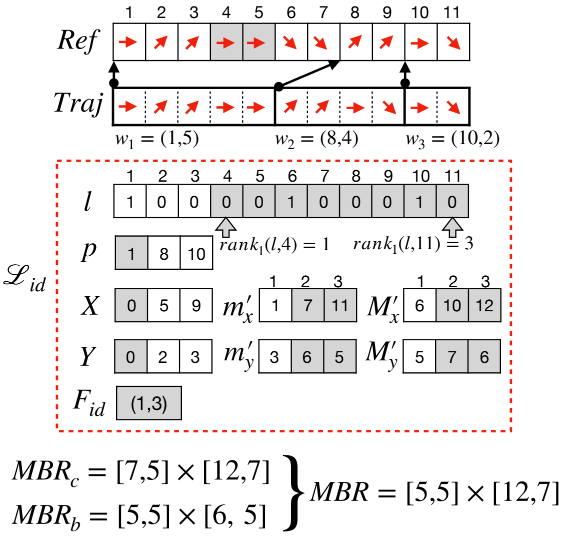


Figure 7.6: Example of computing the MBR with RCT.

algorithm can compute the next movement with ContaCT by counting the number of 0-bits from i_p and i_n to the next 1-bit by using the $select_{next}$ operation.

Therefore, we can use the same mechanism of ContaCT, but taking into account that we need to synchronize both i_n and i_p of each axis when the algorithm advances to the next phrase.

7.4.2.3 Minimum Bounding Rectangle

RCT can solve in constant time the MBR query, but it requires to add additional information to the structure. Firstly, we need to know which phrases from $\mathcal{L}_{id}$ are involved in that interval. We denote those phrases as $W = w_a w_{a+1} \dots w_{a+k-1}$, where w_a and w_{a+k-1} can be completely included in that interval or not. Notice that every pointer in W can be pointing to different non-consecutive parts of the reference. Hence, we would need to compute the MBR for each phrase in W. That means, there are two cases: (i) computing the MBR covered by a whole phrase or (ii) computing the MBR of an interval within the phrase. Both cases can be solved by computing the MBR as in *ContaCT* through the reference. Once all the involved MBRs are computed, the final MBR is the result of obtaining the maximum and minimum coordinates from the previous MBRs.

Notice that the previous algorithm depends on the number of phrases of W. Assuming that k is that number, it would require k MBR operations through the reference. In order to solve it in constant time, we add a $O(z)$-sized structure that stores the minimum and maximum on each axis for each phrase. In Figure 7.6, arrays m'_x and M'_x store the minimum and maximum concerning the horizontal axis within each phrase; that is, at the second position of both arrays, we have the minimum and the maximum of the second phrase, respectively. Analogously, we have the minimum and maximum of the vertical axis stored in m'_y and M'_y. We create an *rmq* (resp. *rMq*) structure for the minima (resp. maxima) over those arrays. Consequently, the MBR for those phrases completely contained in $[t_b, t_e]$ (MBR_c) can be computed in $O(1)$ time by using *rmq* and *rMq*. Then, we compute the MBRs on the remaining phrases, but considering only that interval included within $[t_b, t_e]$. That is, if a phrase covers $[t'_b, t'_e]$, when $t'_b < t_b$ we compute the MBR in $[t_b, t'_e]$ (MBR_b) and if $t_e < t'_e$ we compute it in $[t'_b, t_e]$ (MBR_e). The global MBR is computed by taking the maximum and minimum coordinates from MBR_b, MBR_c, and MBR_e.

For example, in Figure 7.6, we compute the MBR in the time interval $[t_4, t_{11}]$. Firstly, we obtain the involved phrases in that interval by using $rank_1$ at the bitmap l at positions 4 and 11. We can observe as the required information is included within phrases w_1, w_2, and w_3, but we have to know which of these phrases are completely covered by the queried interval. Since the partially covered phrases can only occur at the extremes, we know that w_2 is completely covered, but we need to check the remaining phrases. By using $select_1(l, 1) = 1$ and $select_1(l, 1+1) - 1 = 5$, we know that w_1 is in the range $[t_1, t_5]$, thus that phrase is not completely covered. The last phrase corresponds with the interval $[t_{10}, t_{11}]$, so w_3 is completely covered. Consequently, we have to compute MBR_c of the completely coverd phrases $w_2 w_3$. With the help of rmq and rMq on m'_x and M'_x we obtain the minimum and maximum (7 and 12) for those two phrases $w_2 w_3$. The same procedure through M'_y and m'_y obtains the minimum and maximum (5 and 7) on the vertical axis. Therefore, our MBR_c turns $[7, 5] \times [12, 7]$. Then, we need to look for MBR_b, which partially covers the first phrase w_1. The MBR_b is solved in the reference by using *ContaCT*. The algorithm takes into account that the previous location of the object to that phrase was $F_{id} + (X[1], Y[1]) = (1, 3)$, and the MBR is calculated considering the last two locations of w_1, at indexes 4 and 5 on the reference. As in those two indexes the location of the object is $(5, 5)$ and $(6, 5)$, ContaCT gets $MBR_b = [5, 5] \times [6, 5]$. Finally, MBR_c and MBR_b are merged into one MBR, obtaining $[5, 5] \times [12, 7]$. Notice that we do not have to compute MBR_e because the last phrase is completely contained in the queried interval.

With that algorithm, we only need to run at most three operations, one for those consecutive phrases completely included within $[t_i, t_j]$ and two for the phrases that intersect with the extremes of the time interval. Therefore, we can solve in $O(1)$ time by using $O(z)$ extra space.

7.4.3 Spatio-temporal range queries

7.4.3.1 Time Slice

Once we detect the candidates with the snapshot, the log only needs to compute each candidate's position at the queried time instant. This operation is solved in RCT as we explained in Section 7.4.2.1. By checking if the object is within the queried region, the object is either added to the solution or not.

7.4.3.2 Time Interval

After detecting the candidates on the snapshot, the log has to discern which objects are within the region r_q and the queried interval $[t_b, t_e]$. Due RCT computes the MBR in constant time, for each candidate, it runs a binary search looking for an interval of time in $[t_b, t_e]$ whose MBR is contained within the queried region r_q.

7.4.3.3 K-Nearest Neighbors

Every time an object is on top of the priority queue of candidate Q_c, we have to compute its position at the queried time instant t_q and then we try to add it to the queue of known results Q_r. As we can see, the log only needs to compute the position of the object at t_q. In the case of RCT, it is solved as we presented in Section 7.4.2.1.

Chapter 8

Using real data

All the presented structures consider a raster model representation of the space and deal with object positions at regular time instants. However, the incoming spatial data of objects are coordinates that may not match the considered regular time instants. Therefore, we need a mechanism that transforms those locations to the cells of a grid that represents the space, and they have to be associated with the tracked time instants. Thus, a preprocessing of the data is necessary.

In the previous chapters, we assume that all the objects start emitting their location at the same time instant, and all the time instants have information about the location of the object. Though the data preprocessing can help to accomplish those constraints, there are cases where it is impossible to avoid lack of information, for example, the objects can start sending their locations at different time instants, or there are significant periods of time where the location of the object is unknown. Therefore, our structures need some modifications in order to consider trajectories where those two constraints do not hold.

For these reasons, in this chapter, we explain the prepocessing of the data to achive their normalization, and how to deal with the uncertainty in our structures.

8.1 Data preprocessing

When we have a dataset with trajectories, there are errors of GPS coordinates caused by a bad calibration of the GPS, problems with the network, or other reasons. This kind of information must be fixed in such a way that it does not affect the other transmitted locations. Therefore, we decided to remove all entries from the dataset whose speed between the previously known location is considered unrealistic. That is, the speed is greater than a parameter that we call *speed limit*. This parameter can be adjusted depending on the domain of the data; for example, if we are using trajectories of cars, we can limit reasonably it to 200km/h, instead if we represent people running *speed limit* can be set to 45km/h.

Once those errors have been deleted, the next step deals with the discretization of the coordinates and time instants. To transform the coordinates into grid cells, the algorithm computes the *origin*, the known location with the minimum values for both coordinates. Therefore, we can calculate the distance in both axes for any location with respect that origin, that is d_x, d_y. Since every cell of the raster model is squared and has a fixed side size β, with an algebraic operation, we know that the object is located in the cell $(\lfloor d_x/\beta \rfloor, \lfloor d_y/\beta \rfloor)$. Notice that the value of β can be tuned depending on the domain, in fact, different units of measurement can be used (e.g. feet, degrees, radians).

Finally, we define a parameter t_{span} that determines the gap between each tracked time instant. For example, if t_{span} is 60 seconds, the structure only stores the location of all object after 60 seconds, that is, at time instants 0, 60, 120, and so on. Since every object can emit its location with different frequencies that do not match with the tracked time instants, we need a mechanism that transforms the actual time instants to the regular ones. By using a window of size t_{span} for the first window, we can observe the object's locations during $[0, t_{span} - 1]$, and the regular time instant 0 can be represented by choosing one of those time instants. We repeat this step for each regular time instant i and the interval $[i \times t_{span}, (i + 1) \times t_{span}]$. In our case, we choose the first element of that interval since it is the closest to the tracked time instant.

After the data processing, the spatial and temporal information is discretized into a grid and regular time instants, respectively. Nevertheless, objects can still have periods of time without spatial information. In some cases, when the difference of time between the time instant without data and the previous one with information is smaller than a parameter τ, we can interpolate the location of the object. Notice that τ must be defined by taking into account the domain and it should be a value that avoids a large displacement of the object.

8.2 Missed data

Although the previous process reduces the number of instants without information, the datasets can still have some periods without knowing the location of the objects and trajectories that start at different time instants. For this reason, our structures have to deal with those missed data. Different approaches are applied depending on the kind of log.

8.2.1 Events of missed data

Notice that, in the case of ScdcCT and GraCT, if some of the log information is unknown, there is no way to associate a movement with its corresponding time instant. For example, consider a trajectory with ScdcCT whose log has three entries but evolves the interval of time $[t_1, t_5]$, we notice that there are two time instants without information about the trajectory.

For this purpose, the log manages three events: when an object starts emitting its positions, when it stops, and when it stops emitting signals for periods of time. In fact, the three cases are homogeneously managed by the following events, each identified in the log with a special codeword. For a better explanation of these codewords, let us define two contiguous snapshots as $\mathcal{S}_h$ and S_{h+d}, and the section of the log that tracks the movements between those snapshots as $\mathcal{L}_{id}(h, h+d)$.

- *Disappearances* (D). The occurrence of codeword D in $\mathcal{L}_{id}(h, h+d)$ means that object O_{id} stopped emitting its position at the corresponding time instant of that portion of the log and that O_{id} did not restart emitting (at least) until the time instant of the next snapshot ($\mathcal{S}_{h+d}$).

  To enable backward traversal of the logs, each disappearance must store the time instant and the absolute coordinates of the object at the moment where it disappears.

- *Absolute appearance* (AA). The occurrence of codeword AA in $\mathcal{L}_{id}(h, h+d)$ means that O_{id} started to emit its position in the corresponding time instant of that portion of the log and that O_{id} had not emitted its positions (at least) since the previous snapshot ($\mathcal{S}_h$). AA may signal the first appearance of an object or its reappearance after having disappeared in a preceding section of log.

  To enable forward traversal of the logs, an absolute appearance stores the time instant and the absolute coordinates where the object appears. Note that, even if the object is reappearing, it could do that very far away from the previously known location.

- *Relative disappearance.* The occurrence of a codeword indicating a relative disappearance in $\mathcal{L}_{id}(h, h+d)$ in an entry corresponding to time instant t_i means that O_{id} stopped emitting positions at that time instant and that O_{id} restarted emitting its positions at a time instant between t_i and the time instant of the next snapshot ($\mathcal{S}_{h+d}$). In other words, a relative disappearance occurs when the object disappears and reappears within the same section of log.

  A relative disappearance must store the number of time instants in which O_{id} was not emitting its positions and, depending on the type of relative disappearance, also a movement:

  – *Relative disappearance with movement* (RM) means that O_{id} appears in a different position concerning its last known position. This requires the log to store the relative movement (using the spiral encoding).

  - *Relative disappearance without movement* (RNM) means that O_{id} appears in the same position where the signal was lost. Therefore, there is no need to store a movement.

The extra information is stored in two auxiliary arrays associated with each partition of the log $\mathcal{L}_{id}(h, h+d)$:

- Array $\mathcal{D}_{id}(h, h+d)$ stores the time characteristics of each codeword that represents a disappearance or appearance:
  - in the case of relative disappearances, the time the disappearance lasted;
  - in the case of an absolute appearance or a disappearance, the absolute time instant of that event.
- Array $\mathcal{P}_{id}(h, h+d)$ stores the spatial characteristics of each codeword that represents a disappearance or appearance:
  - in the case of relative disappearances with movement, the relative movement with respect to the last known position;
  - in the case of an absolute appearance or a disappearance, the absolute position of that event.

Regarding the synchronization of the appearances and disappearances with the time instants, during the traversal of the log we can find two cases: reading an D or a relative disappearance (RM or RNM). In the first case, there is no more information about the object until $h+d$, and the next location of that object is stored in a snapshot or with AA, in both cases, we know the corresponding time instant and its location. Therefore, we keep the synchronization between the locations and the time instants. For the second case, when RM or RNM appears, the duration of the disappearance is stored and we only need to add it to the time instant previously read to compute the next time instant with information. If the codeword is RNM, the object is in the same position; otherwise, we have to add the relative movement of RM to the previous position. Thus we obtain the position where the object appears.

For example in Figure 8.1, with ScdcCT the location of the object at time instant t_{19} can be computed by traversing the log and tracking the time instant of each entry. We start at the first entry, which is marked with AA and contains the information $\langle t_4, (0, 5)\rangle$, the time instant, and position where the object appears. Therefore, we know that the object is at positon $(0, 5)$ at t_4, then we traverse the log. Every time we read an entry the time instant increases by one, thus we reach the fourth entry at t_7 and with location $(4, 4)$. Since the next entry is marked as RM, we have to consider the extra information. The elapsed time is three time instants and the displacement $(-2, 2)$, thus we obtain the position of the object $(2, 2)$ at t_{10}. Those steps are repeated until the current time instant is t_{19}, getting the position $(6, 4)$.

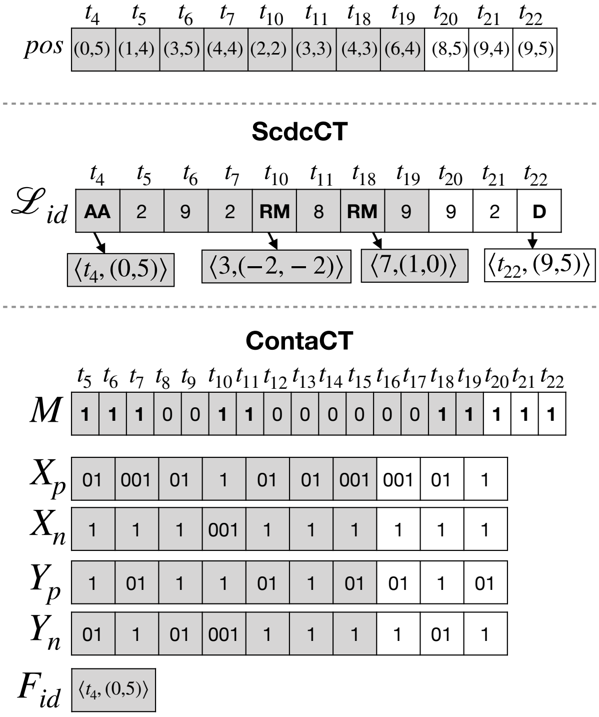

pos
t4 t5 t6 t7 t10 t11 t18 t19 t20 t21 t22
(0,5) (1,4) (3,5) (4,4) (2,2) (3,3) (4,3) (6,4) (8,5) (9,4) (9,5)
ScdcCT
ℒ id
AA 2 9 2 RM 8 RM 9 9 2 D
⟨t4, (0,5)⟩
⟨3,(−2, − 2)⟩
⟨7,(1,0)⟩
⟨t22, (9,5)⟩
ContaCT
t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22
M 1 1 1 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1
Xp 01 001 01 1 01 01 001 001 01 1
Xn 1 1 1 001 1 1 1 1 1 1
Yp 1 01 1 1 01 1 01 01 1 01
Yn 01 1 01 001 1 1 1 1 01 1
Fid ⟨t4, (0,5)⟩

Figure 8.1: Different approaches to represent the lack of information.

8.2.2 Setting marks of missed data

For those logs where the position of an object is solved in constant time (ContaCT and RCT), we use a different approach to the previous one. We add an additional bitmap $M[1,n]$ and the first time instant of each trajectory t_{start}, where n is the number of time instants from t_{start} to the last time instant of the trajectory. Each index i of M corresponds with the time instant $t_{start+i}$, and $M[i] = 1$ iff there is location information for $t_{start+i}$. Consequently, we can map each time instant t_j with the corresponding movement by computing $rank_1(M, t_j - t_{start})$. Therefore, in order to compute the different queries, the algorithm has to map those time instants to movements and solve them with the mapped movements.

In the bottom part of Figure 8.1, we can observe the structure for ContaCT. In order to compute the position at t_{19}, we have to compute the number of known movements up to that time instant. Since the trajectory starts at t_4, the number of movements is $rank_1(M, t_{19} - t_4) = rank_1(M, 15) = 7$. Therefore, t_{19} maps with the seventh movement, and its displacement can be computed with as the number of 0s up to the seventh 1-bit in the positive bitmap minus the number of 0s until the seventh 1-bit in the negative bitmap, for each axis. That is, we obtain an accumulative movement of $(6, -1)$ cells. By adding it to $(0, 5)$, we know that the object is in the cell $(6, 4)$ at t_{19}.

8.2.3 The effect of missed data on selecting the candidates

For spatio-temporal range queries, when using the k^2-tree-based snapshots, recall that we first check in the closest snapshot to the queried time instant (or time interval), the objects with chances of being part of the result of the query (called candidates), by using the spatial index characteristics of the k^2-tree. Though the log manages the appearances and disappearances, we need a mechanism that allows adding to the candidates those objects that have information in the involved section of the log, but are not present at the time instant of the snapshot. If we do not store more information at this kind of snapshot, those objects without information are not considered candidates. For this reason, we store two additional arrays:

- *Dis*, a list of the active objects during the time interval $(t_{h-d}, t_{h-1}]$ that stopped emitting before the time instant represented by $\mathcal{S}_h$.
- *App*, a list of objects that are not present in $\mathcal{S}_h$ and appear (or reappear) before the next snapshot, that is in the interval $[t_{h+1}, t_{h+d})$.

Therefore in time-slice, time-interval, and knn queries the candidates of *Dis* or *App*, depending on whether the traversal is backward or forward, are directly added to the set of candidates.

Chapter 9

Experimental evaluation

In this chapter we compare all the proposed structures. Additionally, we compare them with the MVR-tree, a classical spatio-temporal index. The MVR-tree [TP01b] is one of the best exponents of the classical approaches, where the speed is the main goal, without worrying too much about space. MVR-tree is a spatio-temporal index designed to solve *range* and *nearest neighbour* queries. It can be combined with other structures, as done with MV3R-tree [TP01b], which adds auxiliary 3DR-trees for speeding up basic queries like retrieving the trajectory of an object.

For our experimental evaluation, we implemented our structures ScdcCT, GraCT, ContaCT, and RCT in C++, using components from the SDSL library[1] [GBMP14]. We used the MVR-tree of the spatialindex library.[2] The experiments were conducted on an Intel® Core$^{\mathrm{TM}}$ i7-3820 CPU @ 3.60GHz (4 cores/8 siblings) with 10MB of cache and 64 GB of RAM, running Debian GNU/Linux 9 with kernel 4.9.0-8 (64 bits). We used compiler g++ version 6.3.0 with `-O9` optimization.

9.1 Datasets

During the experimental evaluation, we used four sources of data, three of them containing real-world data and the other one with pseudo-real data. They are briefly described below.

- `Ships`: a real dataset obtained from MarineCadastre.[3] It contains the location of 4,461 vessels travelling inside the UTM Zone 10 during one month of 2017.
- `Planes`: it contains real data corresponding to 2,263 trajectories between 30 European airports from aircrafts of 30 different airlines. Altitude is not

[1] `https://github.com/simongog/sdsl-lite`

[2] `http://libspatialindex.github.io`

[3] `http://marinecadstre.gov/ais`

	Ships	Planes	Taxis	Ciconia
Total objects	4,461	2,263	24	88
Total points	63,093,559	36,741,877	44,682,648	4,390,159
Max x	6,000	22,901	1,215,897	407,367
Max y	647,776	4,688	1,160,360	299,593
Max *time*	44,639	172,547	2,102,639	505,573
Size Plain	1,293.13 MB	738.93 MB	1,024.00 MB	98.71 MB
Size Bin	481.37 MB	315.36 MB	426.12 MB	41.87 MB
Size *p7zip*	38.11 MB	58.30 MB	53.58 MB	7.21 MB

Table 9.1: Datasets and their dimensions.

considered, only latitude and longitude are represented in our dataset. The original data can be obtained from *OpenSky Network*.[4]

- Taxis: a pseudo-real dataset containing trajectories of 433 taxis in New York City during 2013. Since the original dataset only includes the origin and destination of each trip, the trajectories were computed as the shortest path between each origin and destination by taking into account the road network. The original data are available at *NYC Taxis: A Day in the life*.[5]
- Ciconia: a small and non-repetitive real dataset containing the locations of 88 white storks travelling between Europe and North Africa from 2013 to 2019. The original data can be obtained from *MoveBank Data Repository* [FFW16, CFWF19].

All the structures considered here deal with raster data, that is, every position is stored into a discrete grid. Every location associated to a given object from our four datasets must be represented by the corresponding cell of that grid. Depending on the application and its precision requirements, a different size of cell is chosen.[6]. In particular we choose cells of size: 100 × 100 meters in Ships, 1000 × 1000 meters in Planes, and 10 × 10 meters in Taxis and in Ciconia. The time also requires a normalization because each object emits its location with different frequency, and our structures need to synchronize those positions at regular time instants. We used regular intervals of 1 minute for Ships, 15 seconds for Planes and Taxis, and 6 minutes for Ciconia.

Following the ideas presented in Chapter 8, after the discretization, we detected some errors in several locations of our real datasets (Ships, Planes, and Ciconia),

[4] https://opensky-network.org

[5] http://chriswhong.github.io/nyctaxi/

[6] https://en.wikipedia.org/wiki/Decimal_degrees

because some movements would require extremely high speed. To filter these errors, a maximum speed parameter was set for each dataset: 800 km/h in `Planes`, 234 km/h for `Ships`, and 54 km/h in `Ciconia`. As a consequence, those signals that imply exceeding the maximum speed were removed: 0.01%, 0.83%, and 0.42% of the signals were deleted from `Ships`, `Planes` and `Ciconia`, respectively. In addition, an object can emit its signals with different frequency; for example, in `Ships`, the frequency is lower when they are in a port. The asynchronism between the frequency of emission of GPS devices and the regular time instants of the structures, makes it possible that there are some time instants without information about the position of the object. In the cases where the difference between two consecutive signals is less than 15 time instants, we interpolate the locations of those time instants.

Trajectories are usually stored in a plain text file composed of four columns: *object identifier*, *time instant*, *x coordinate*, and *y coordinate*. To obtain a fair comparison, we stored all this information in binary form by using the minimum number of bytes required for each column. For example, in `Ships`, two bytes are used to represent the first column (max value 4,461), two bytes for the time instant column (max value 44,639), two bytes for the x-axis (max value 6,000), and finally, three bytes are used for the y-axis (max value 647,776).

Table 9.1 shows a description of the datasets, their binary and plain text size, and their size after compressing them with *p7zip*. The last row gives us an idea of how compressible the data are: we observe that *p7zip* compresses the data to 5%–20%[7] of its binary representation.

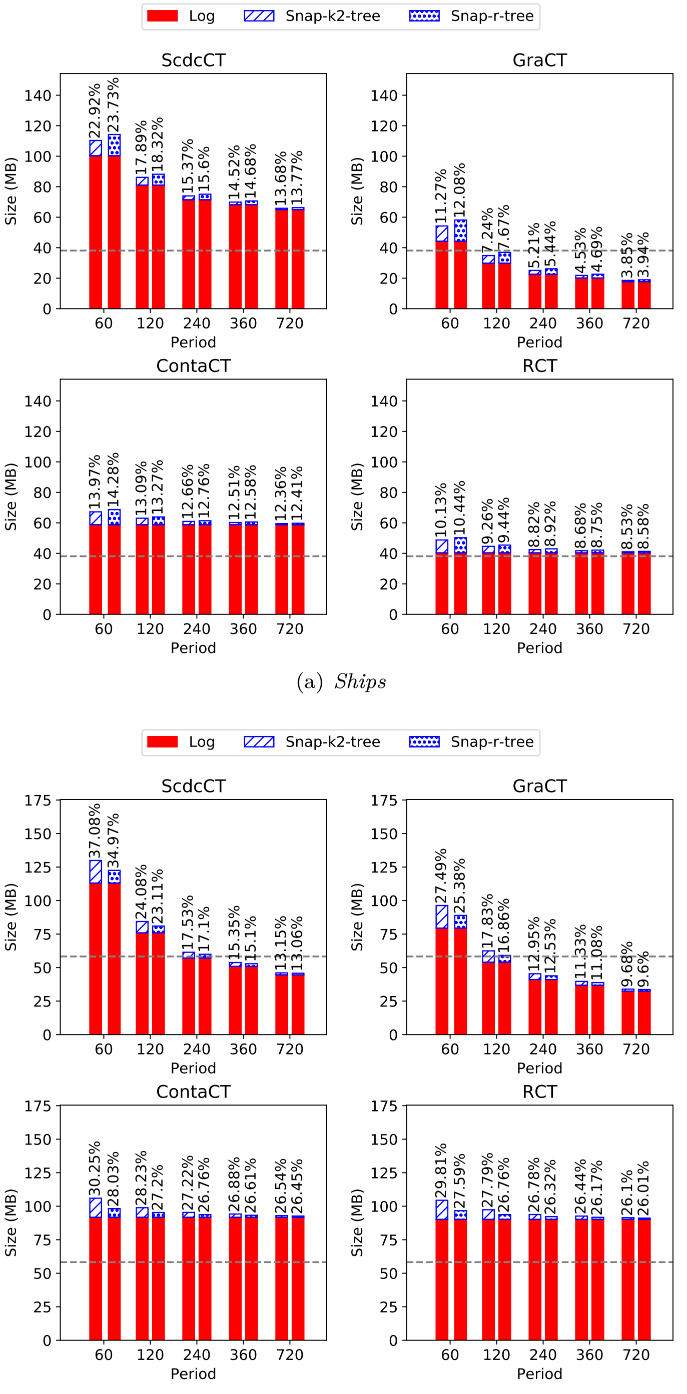
9.2 Compression

Firstly, we analyze the space requirements of all our structures. For each dataset we build the eight possible structures, that is the combination of kinds of snapshots (2) and logs (4). All of them use the same distances between snapshots, i.e, we keep a snapshot every $d = 60, 120, 240, 360$, and 720 time instants.

Figures 9.1 and 9.2 show the compression ratios and the space requirements for each dataset. Note that each chart refers to a specific implementation of the log and the odd/even bars represent those structures that use snapshots based on k^2-tree and R-trees, respectively. Each chart corresponds with the best setup of each technique. Therefore, ContaCT represents D_p and D_n using plain bitmaps and the size of the reference of RCT, in plain format, is 100MB on `Ships`, `Planes`, and `Taxis`. Instead, `Ciconia` uses sparse bitmaps and the size of the reference is 50MB. We can observe that most of the space is occupied by the compressed log (represented by the red part of the vertical bars).

In the case of ScdcCT and GraCT, the log reduces its size when d increases but, as expected, this does not occur in the case of ContaCT and RCT. In GraCT, the

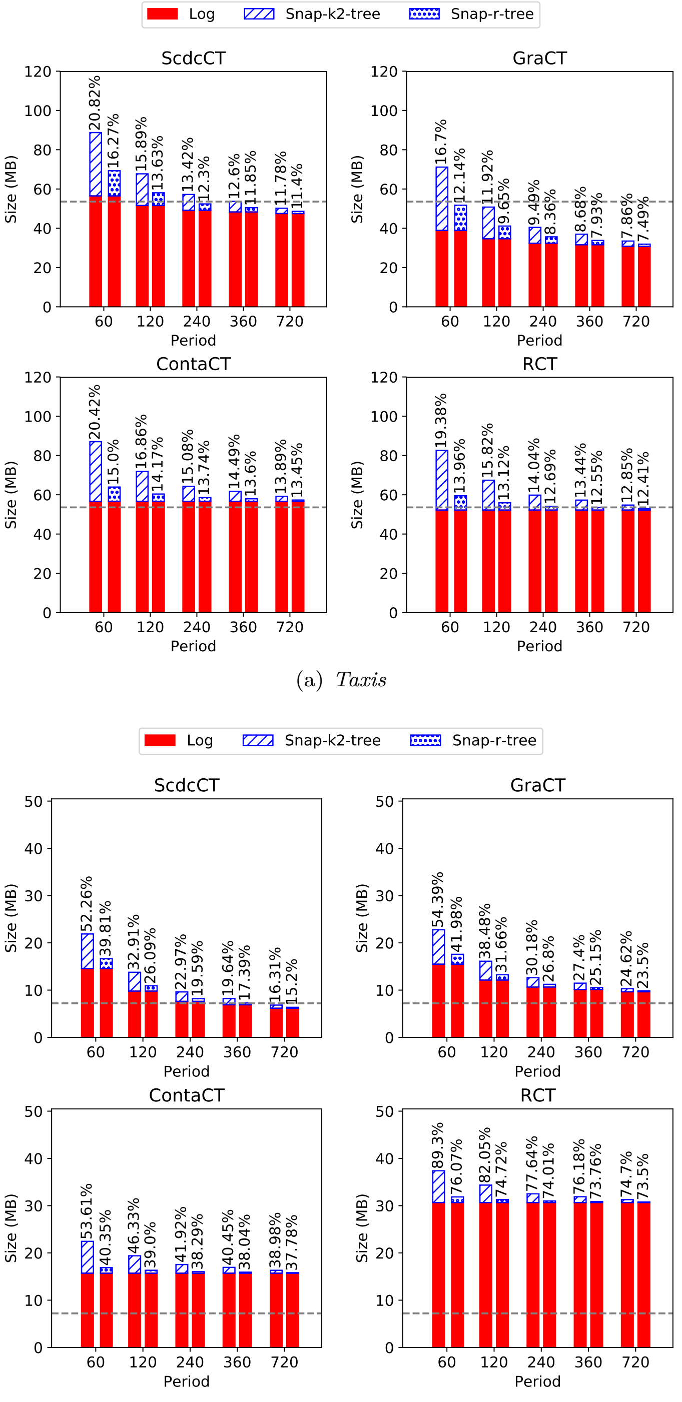
[7] The values are shown as the size of the compressed file as a percentage of the size of the original file.



(a) *Ships*

(b) *Planes*

Figure 9.1: Space requirements of each structure when representing the datasets of *Ships* and *Planes*.



(a) *Taxis*

(b) *Ciconia*

Figure 9.2: Space requirements of each structure when representing the datasets of *Taxis* and *Ciconia*.

quotient of the log space with $d = 720$ versus $d = 60$ is 0.63 and in ScdcCT that quotient is 0.66. As d increases, the length of the log between snapshots is larger. Since ScdcCT and GraCT require to know where each portion of the log starts, the size reduces when d is larger. However, that difference of 3% is occasioned because GraCT's grammar compressor finds more repetitiveness in large sections. ContaCT and RCT do not exploit this redundancy, therefore that previous quotient is 1. As we will see soon, this higher space usage is traded by ContaCT and RCT to provide much faster evaluation of some queries.

Regarding the other main component of our structures, the snapshots based on R-trees tend to use less space than those based on k^2-trees, except in the case of Ships. Comparing the space used for the snapshots with the different kinds of logs, we can observe that they require more space in ScdcCT and GraCT than in ContaCT and RCT. For example in the dataset Planes and assuming $d = 60$, ScdcCT and GraCT require 1.18 times more space on snapshots based on k^2-tree than ContaCT and RCT, and 1.45 times more space on those based on R-trees. Recall that the extra-space is required in order to compute the spatial operation $abs(\mathcal{S}_h, id)$ over ScdcCT and GraCT, which is not needed in ContaCT and RCT.

The compression ratios, computed with respect to the binary representations, are showed above each individual bar. In addition, the gray line signals the space used by *p7zip*. As explained, GraCT exploits the redundancy of trajectory data to obtain better compression, requuiring around 75%–85% of the space needed by *p7zip*, (except on Ciconia, which is not repetitive and makes GraCT use 50% more space than *p7zip*). With $d = 720$, GraCT uses 1.6–2.7 times less space than RCT, 1.8–3.2 times less space than ContaCT, and 1.4–3.5 times less space than ScdcCT. As we expected, RCT is ranked between GraCT and ContaCT. However, it slightly reduces the space of ContaCT in repetitive datasets, that is, it uses 70%–99% of the space of ContaCT. ContaCT is the structure demanding more space, however it obtains competitive compression ratios: the version with $d = 720$ uses 10%–60% of the space of a binary representation, and about twice the space used by *p7zip* (which just compresses the data; it cannot solve any query without decompressing the whole dataset). To compare with another system that uses differential compression (and also does not support queries), we built Trajic [NH15], which used 140.30 MB on Ships, 167.05 MB on Planes, and 16.86 MB on Ciconia.[8] However, the space required for Trajic around 2.3–3.7 times larger than that of *p7zip*. Comparing it with our largest setup (ContaCT), Trajic uses 100% more space on Ships, an 85% more on Planes, and a 6% more on Ciconia.

9.3 Query times

In this section, we focus on comparing the performance at query time of the tested techniques. We study the response times of all the structures with the queries

[8] Trajic crashed when building on Taxis.

presented in Chapter 4. All the structures use the same distances of snapshots of the previous experiment, that is $d = 60, 120, 240, 360$, and 720. ContaCT-SD uses the *sd-array* (see Section 2.2.2) over the sparse bitmaps that represent the positive and negative displacements for each axis. The *'X'* value from RCT-X refers to the size in MB of the reference in plain format. The response times correspond to the average user time after a set of the same kind of queries. Note that, in `Ciconia`, we only show the structures that obtain compression ratios lower than 50%. In our experiments we have executed the following queries over all the tested structures:

Object queries

- *ObjectPosition*: this query obtains the position of a given object at a given time instant t_q. We show average times, obtained from 20,000 queries for random objects and time instants.
- *ObjectTrajectory*: this query returns the trajectory followed by an object during an interval $[t_b, t_e]$, where $t_e - t_b$ is fixed at 2,000 time instants. We averaged over 10,000 queries for random objects and time instants t_b.
- *MBR*: it computes the minimum bounding rectangle that covers a trajectory between two time instants t_b and t_e. We computed 1,000 random queries whose time intervals involve 200 instants.

Spatio-temporal range queries

- *TimeSlice S*: this query obtains the identifiers and positions of the objects lying within a small region (40×40 cells) at a given time instant. We averaged over 1,000 queries for random region positions and time instants.
- *TimeSlice L*: this query obtains the identifiers and positions of the objects lying within a larger region (320×320 cells). We run 1,000 queries for random region positions and time instants, and show average times.
- *TimeInterval S*: this query obtains the objects that were in a small region (40×40 cells) at any time instant between t_b and t_e, where the interval size is $t_e - t_b = 100$ instants. We averaged over 1,000 queries with random region positions and time intervals starting at random instants t_b (hence $t_e = t_b + 99$).
- *TimeInterval L*: this query obtains the objects present in a larger region (320×320 cells) over a longer interval of size $t_e - t_b = 800$ instants. We averaged times obtained from 1,000 queries with random region positions and starting time instants t_b.
- *Knn*: this query obtains the K nearest neighbors to a given position at a given time instant, where K is a random value between 1 and 50. We show average times from 1,000 queries with random objects and time instants.

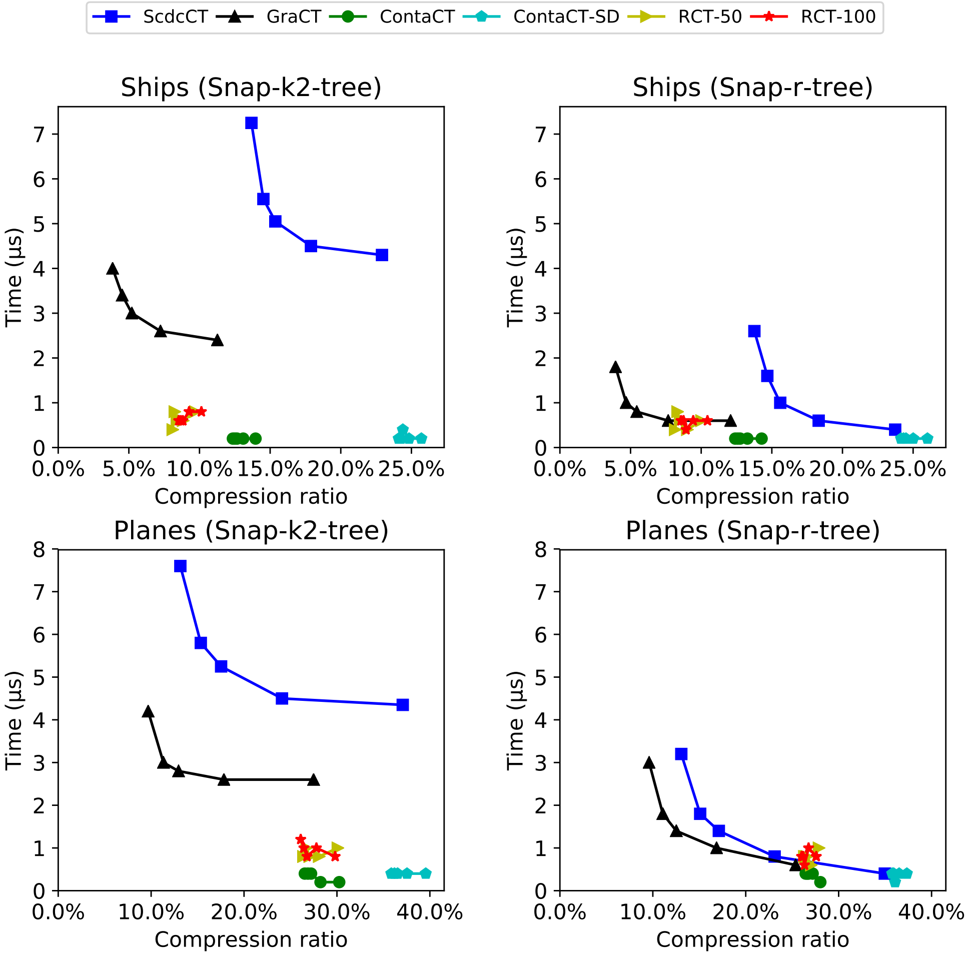


Figure 9.3: Time performance for *ObjectPosition* on `Ships` and `Planes` in microseconds.

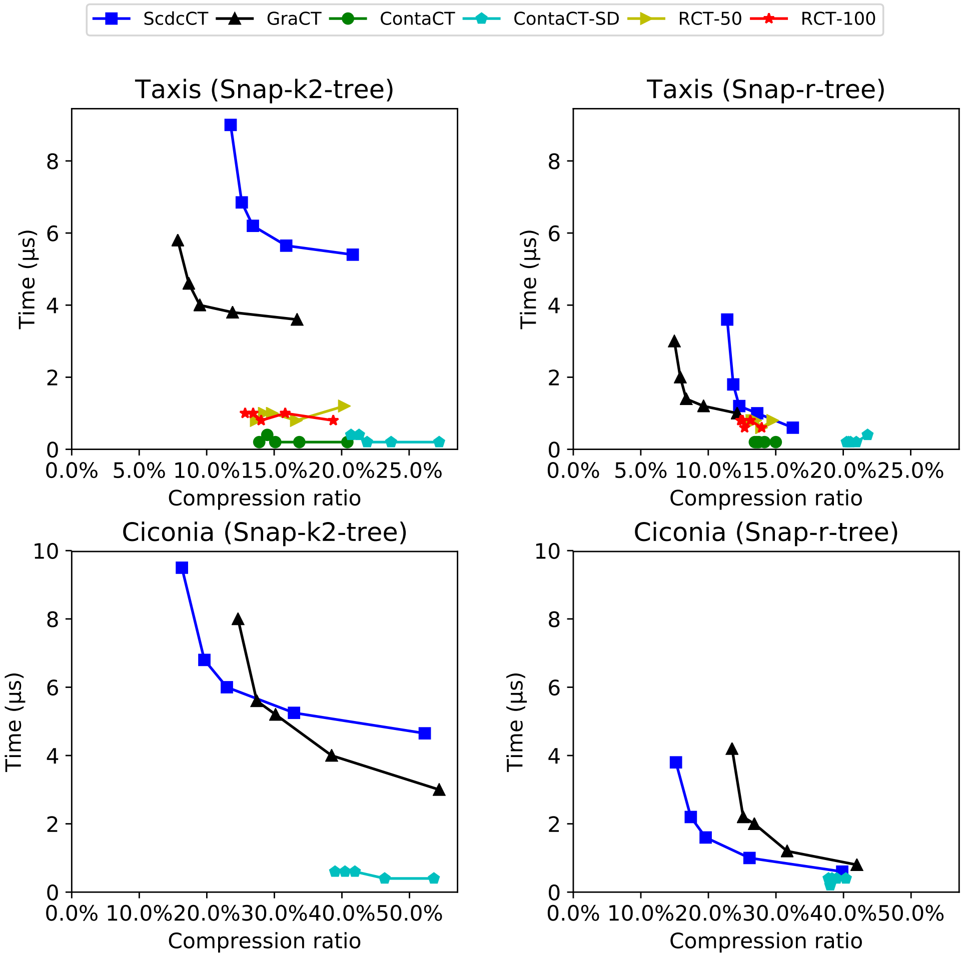


Figure 9.4: Time performance for *ObjectPosition* on `Taxis` and `Ciconia` in microseconds.

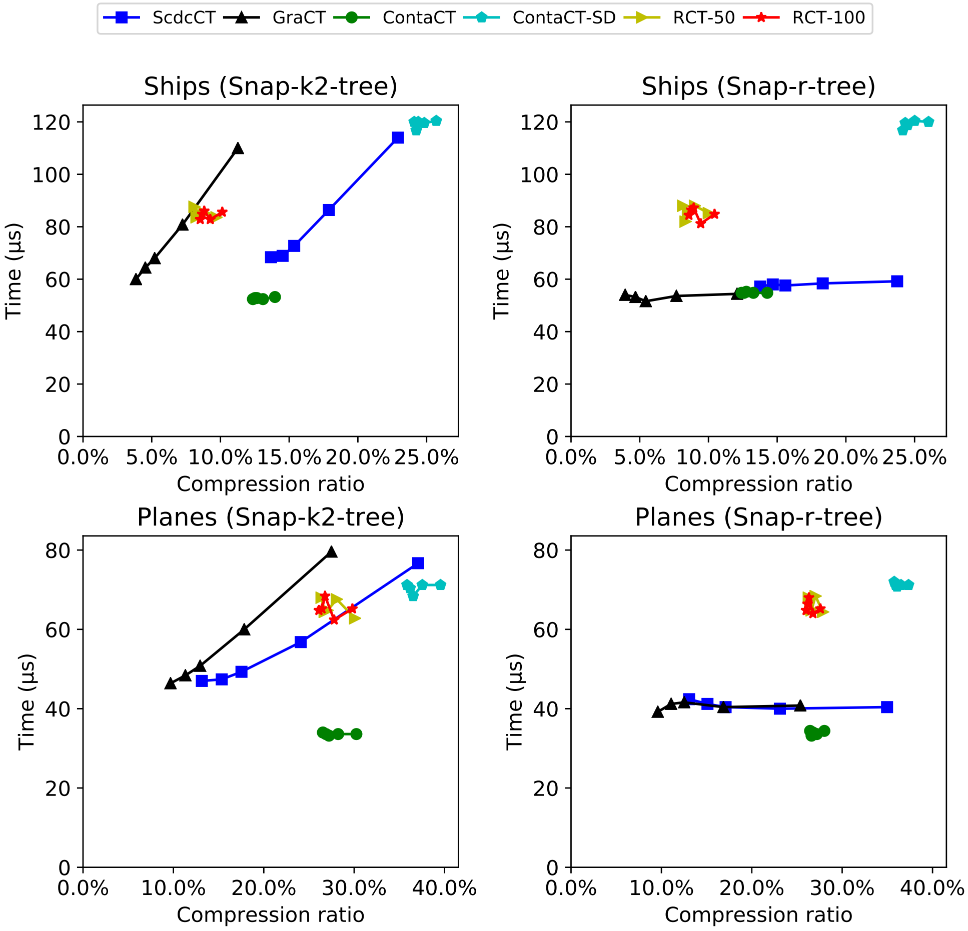


Figure 9.5: Time performance for *ObjectTrajectory* on `Ships` and `Planes` in microseconds.

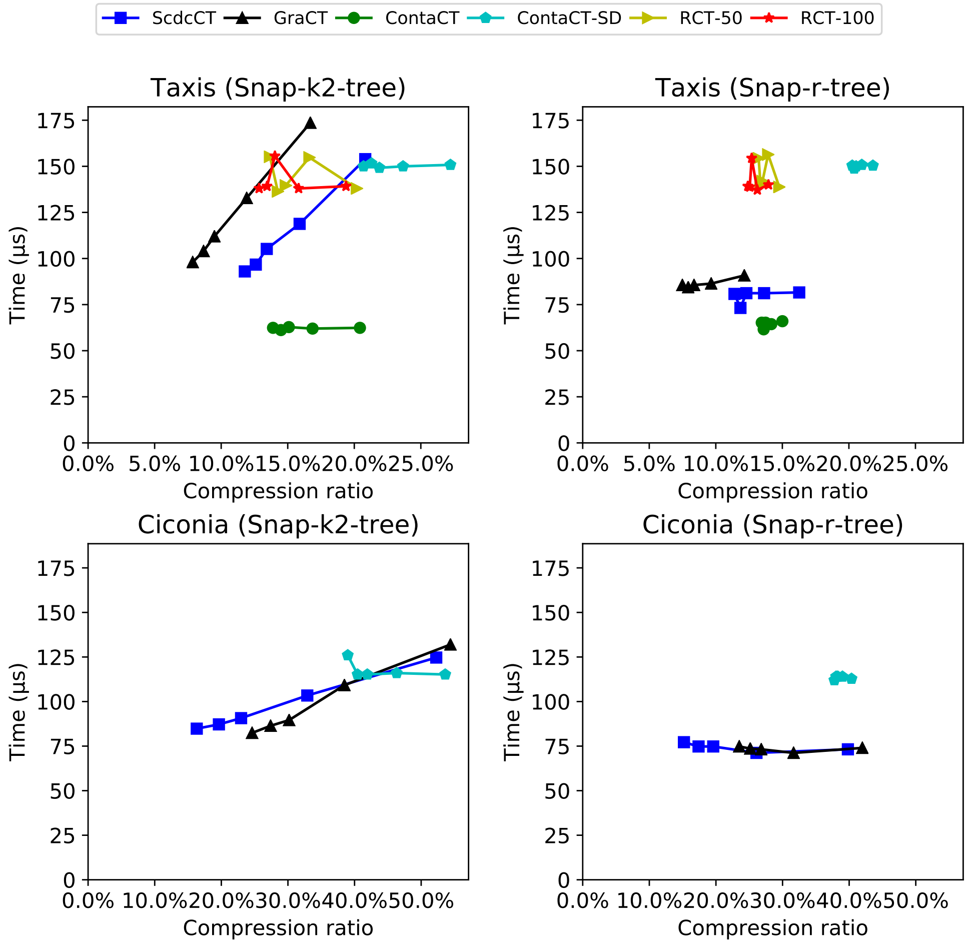


Figure 9.6: Time performance for *ObjectTrajectory* on `Taxis` and `Ciconia` in microseconds.

9.3.1 ObjectPosition

Figures 9.3 and 9.4 show the space-time tradeoffs for *ObjectPosition* queries. The best performance is obtained with ContaCT that computes this query in around 200–600 nanoseconds. Though RCT is based on ContaCT, it requires some additional time in order to detect the phrase that contains the information corresponding to the queried time instant. This makes it around 2–4 times slower than ContaCT. Note that there is no difference in time with the structures that use k^2-tree or R-tree because this query does not require to access the snapshot in ContaCT and RCT.

ScdcCT and GraCT, instead, have to retrieve the position of the object from the snapshot. Since the snapshot representations using a k^2-tree can compute the location at the snapshot in logarithmic time and, in those that use the R-tree can solve it in constant time, this query turns 2–5 times faster on the last kind of snapshots. Once the position is retrieved, the algorithm traverse the log computing the positions. These structures turn out slower as d becomes greater. GraCT benefits from phrases expanding into very long sequences of terminals that appear in repetitive datasets. Due to this, GraCT traverses the log faster than ScdcCT, hence it computes the location 1.1–1.5 times faster than ScdcCT, except in `Ciconia` which is non-repetitive. Such a difference is expected because most of the log is composed of terminals, thus there is no improvement when traversing the log.

The dependence of ScdcCT and GraCT on the value of d is also clearly illustrated in the figure. The times of RCT and ContaCT, instead, remain constant as d changes.

9.3.2 ObjectTrajectory

In trajectory queries, as shown in Figures 9.5 and 9.6, ContaCT also outperforms the other structures in most datasets. However, its variant ContaCT-SD worsens its performance due to the use of sparse techniques when those bitmaps are not actually sparse. For this reason, ContaCT-SD is not the dominant technique in `Ciconia`. During the computation of the trajectory, RCT traverses the involved phrases. For each one of them, it requires to synchronize the cumulative movements until the starting position of the phrase at the reference. That need of synchronizing the phrases, makes RCT around 1.6–2 times slower than ContaCT, when solving this kind of queries.

The only structures that depend on the snapshot to solve this query are ScdcCT and GraCT. The penalty of obtaining the location of the object in the snapshot with a k^2-tree is higher than in the R-tree-based snapshot. Since the trajectories to recover are significantly longer than the snapshot periods, snapshots mostly disrupt the processing of the logs, which explains why times improve with fewer snapshots. Consequently, the trajectory queries are 1.5–11 times slower in ScdcCT and GraCT with the snapshots based on k^2-trees than in those configurations with R-trees. Comparing ScdcCT and GraCT combined with the snapshots based on R-trees, we observe that the difference in response times is small. In the extreme cases, ScdcCT is around 9% slower on `Ships` and GraCT is a 10% slower on `Planes`.

9.3.3 Minimum Bounding Rectangle

Figures 9.7 and 9.8 show the space-time tradeoffs for *MBR* queries. Only those structures with the log of ScdcCT and GraCT need to retrieve the absolute position from the snapshot, and consequently, they obtain different time results depending on the type of snapshot being used. In this query, the combinations of ScdcCT and GraCT with the snasphots based on R-trees are 1.1–1.8 and 1–2 times faster than their respectively variants with k^2-trees.

The best performance in all datasets is obtained with ContaCT, that computes this query in around 1.8–2.4 microseconds. As in *ObjectPosition*, its most immediate competitor is RCT that is around twice slower because of the need of computing the MBR of the phrases that are completely covered by the interval t_b and t_e, and the two MBRs of the phrases that intersect with the bounds of the queried interval. Therefore, RCT yields a good space-time tradeoff. The remaining indexes need to traverse the log in linear time, but GraCT can skip the nonterminals, and compute the MBR as the union of the MBRs of the traversed nonterminals, thus it only needs to decompress those nonterminals at the limits of the time interval. As a consequence it makes GraCT around 5%–30% faster than ScdcCT. In those figures we can observe the dependence of ScdcCT and GraCT on the value of d, whereas the times for RCT and ContaCT remain constant as d increases.

9.3.4 TimeSlice S and TimeSlice L

We start analyzing the results from those setups that use snapshots based on k^2-trees. In Figures 9.9, 9.10, 9.11, and 9.12, we observe in *TimeSlice S* and *TimeSlice L* with $d = 60$, that RCT and ContaCT are the slowest structures, except on `Ciconia`. We can observe two groups of structures that obtain similar times: one is composed of ScdcCT and GraCT and the other contains ContaCT and RCT. The first group obtains better performance than the second one, for example, in `Ships` and `Planes`, ScdcCT and GraCT require around 83% − 89% of the response times of RCT and ContaCT. ScdcCT and GraCT can do better, without necessarily computing the position of each object at time t_q. After processing each value of the log in the way to t_q, they determine if the object still has chances of being within the queried region at t_q. If not, the object is discarded immediately, speeding up these queries. However as d increases, those response times grow faster in ScdcCT and GraCT because of the linear traversal through the log.

On the other hand, the structures based on R-trees are much faster than the previous ones (16–55 times). Note that the R-trees compute the objects that can reach the region at t_q taking into account the trajectory of each individual object, instead of the maximum speed of the dataset. Therefore, the number of considered objects is lower in the snapshots based on R-trees than on k^2-trees, and discarding

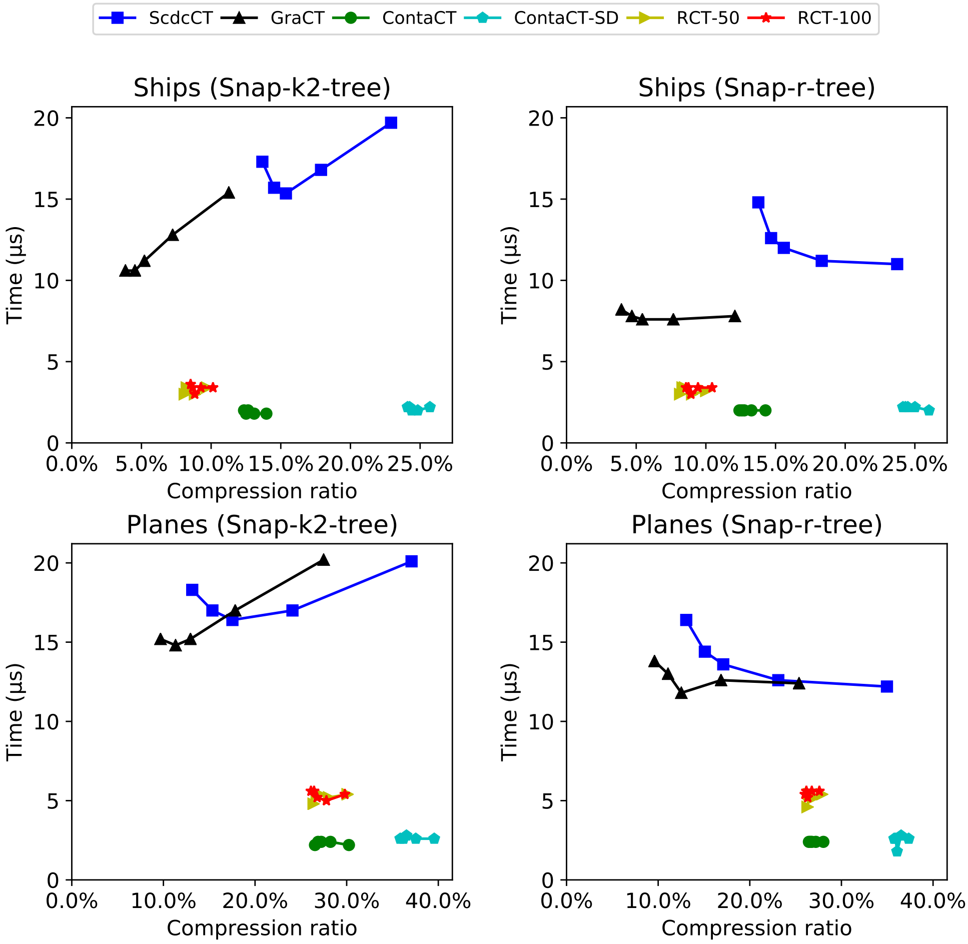


Figure 9.7: Time performance for *MBR* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

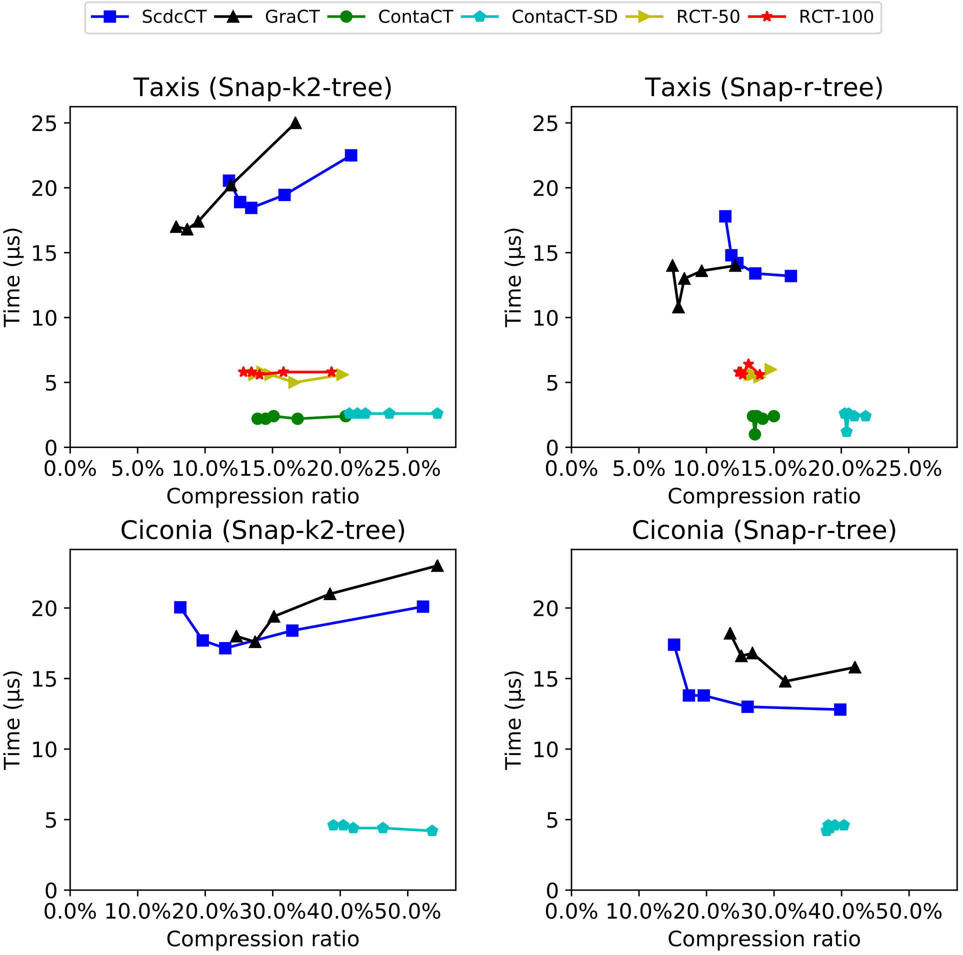


Figure 9.8: Time performance for *MBR* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

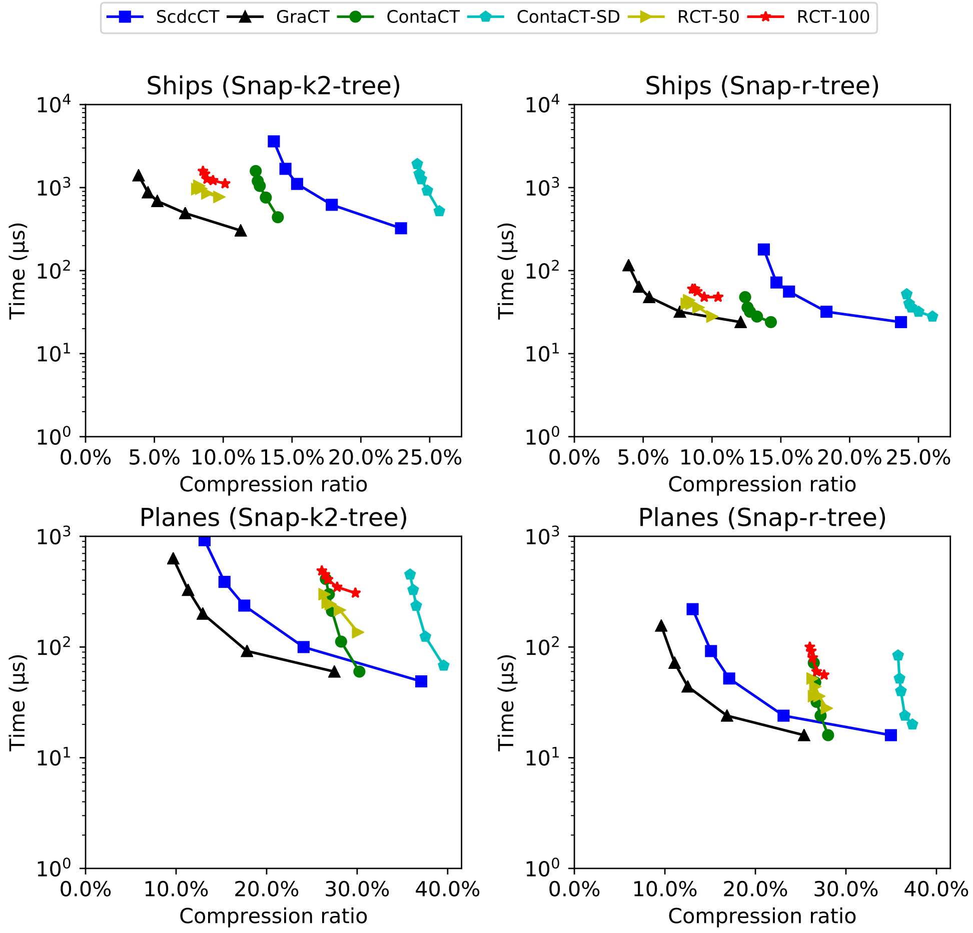


Figure 9.9: Time performance for *TimeSlice S* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

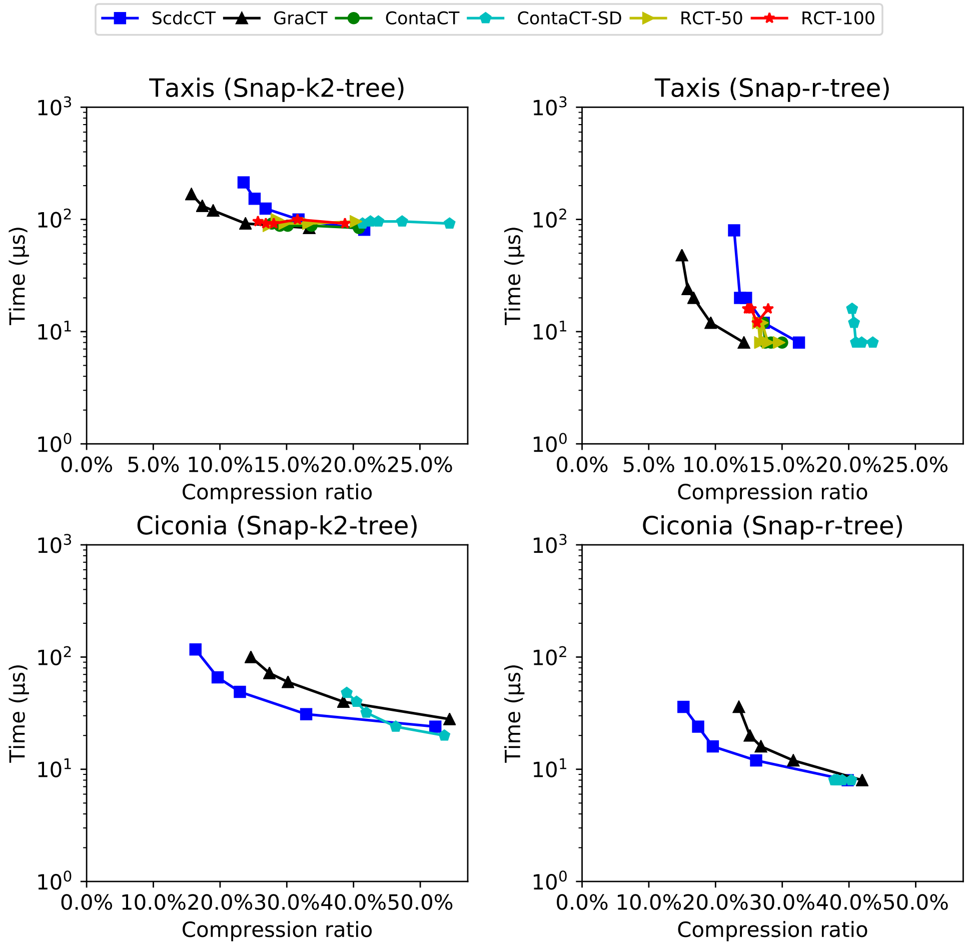


Figure 9.10: Time performance for *TimeSlice S* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

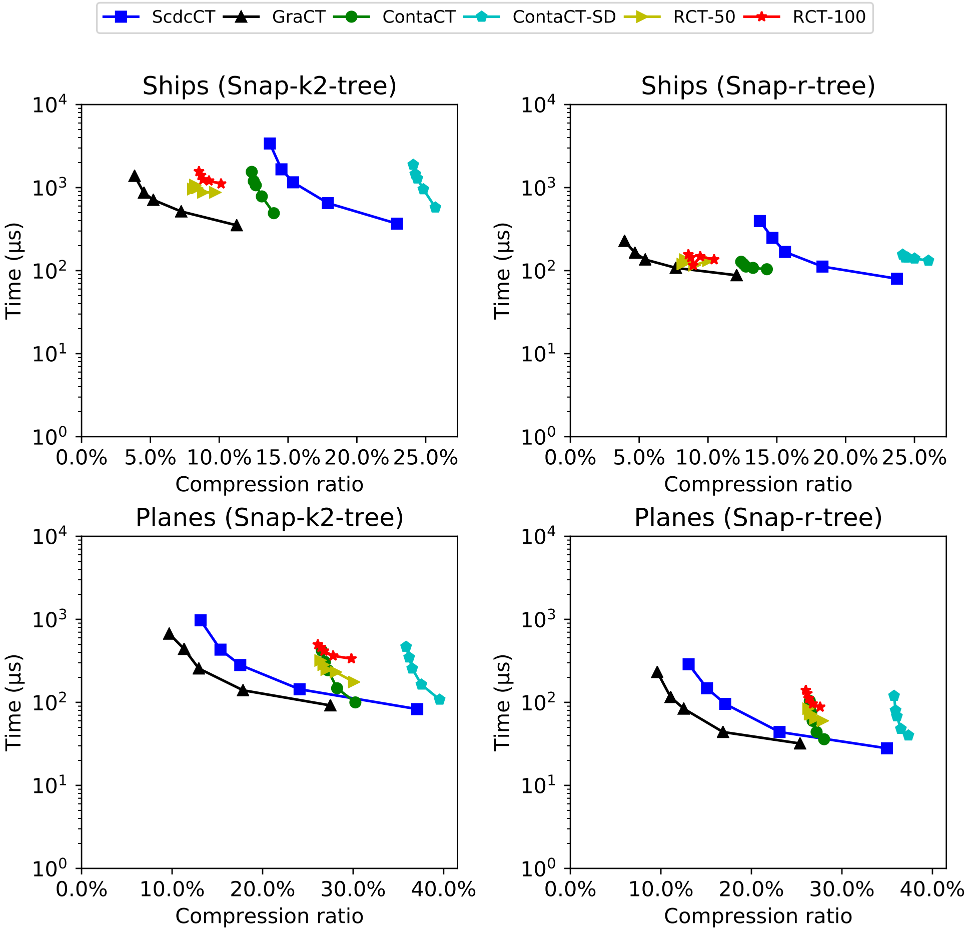


Figure 9.11: Time performance for *TimeSlice L* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

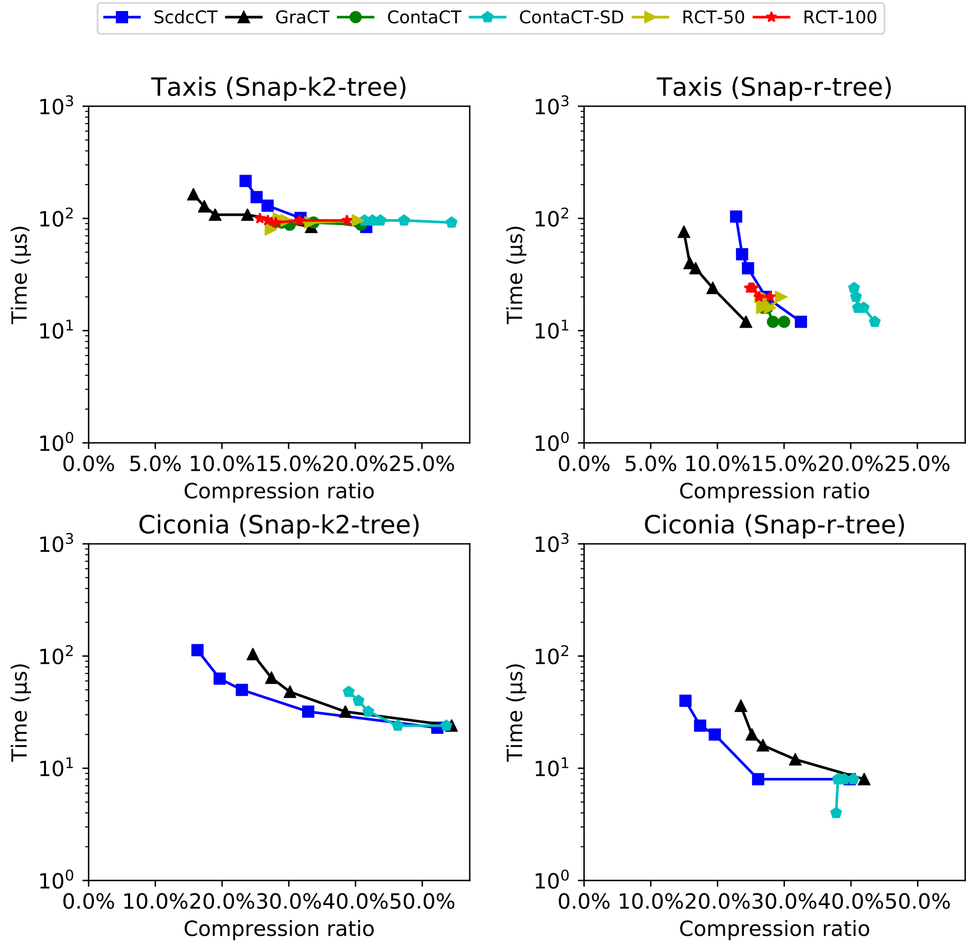


Figure 9.12: Time performance for *TimeSlice L* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

objects in ScdcCT and GraCT is unlikely. As a consequence, all the structures obtain similar times with $d = 60$. However, as d increases, ContaCT and RCT obtain a better performance, for example in `Taxis`, ContaCT and RCT-50 become arouund 1.5–6.5 and 1–5 times faster than ScdcCT, respectively.

9.3.5 TimeInterval S and TimeInterval L

Figures 9.13, 9.14, 9.15, and 9.16 show the average times of time interval queries. We use $\lambda = 20$ for the different structures using ContaCT and RCT to represent the log. As in time slice queries, there is a large difference between the structures that use k^2-trees or R-trees, being 3–15 and 3–33 times slower in GraCT and ScdcCT, respectively. The main reason is that, in those snapshots based on k^2-trees, retrieving an object is much slower than in the R-tree based counterparts. Of course, this becomes more noticeable when the number of retrieved objects increases.

Focusing on those structures with R-trees, we observe that ScdcCT and GraCT grow faster than ContaCT and RCT when d increases. However, with $d = 60$, both GraCT and ContaCT are competing for being the fastest structure. We observe that ContaCT is faster than GraCT in all cases, except on `Planes`, where the maximum-space configuration of GraCT is as fast as ContaCT and still uses 90% of its space. In the other cases, when comparing the fastest configuration of both structures, ContaCT is around 1–1.2 times faster on `Ships`, 1.2–2.5 in `Taxis`, and 1.5–1.8 in `Ciconia`. Since ContaCT can compute any MBR on the fly, it can apply a perfect binary search on the whole interval $[t_b, t_e]$. GraCT, instead, must follow the partitioning given by the grammar, where each nonterminal stores its MBR. It may require traversing several nonterminals to cover the queried interval, and even several snapshots on large intervals. Regarding the remaining indexes, RCT is around 1.7–4 times slower than ContaCT in all datasets. In the case of ScdcCT, although it is 28% faster in the maximum-space configuration on `Planes`, it becomes 1.1–4.3 times slower than ContaCT in the remaining datasets.

9.3.6 K-Nearest Neighbor

Figures 9.17 and 9.18 show the response times of the Knn query for all the structures. All the algorithms for solving this type of query start the process from the snapshot and prioritize those objects that have more chances to be closer to the queried point. As those snapshots based on R-trees take into account the trajectory of the object, and those based on the k^2-trees do not, the objects are better prioritized in the first kind of snapshots. Consequently, the best performance is obtained with those structures that use snapshots based on R-trees that can solve k-nearest neighbor queries around 1.3–15.5 times faster.

Focusing on the structures that use snapshots based on R-trees, in this kind of queries, ContaCT becomes the best choice in terms of performance. When $d = 60$ it is as fast as GraCT and ScdcCT. In addition, when d increases, the response times

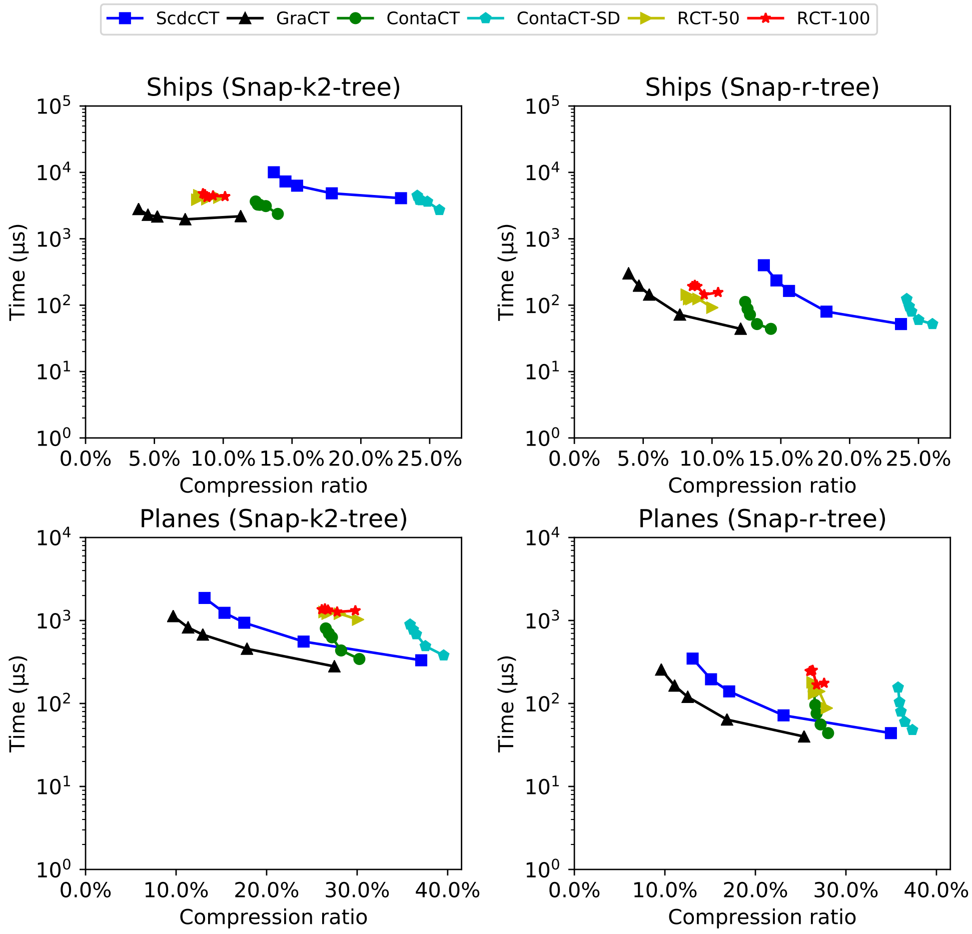


Figure 9.13: Time performance for *TimeInterval S* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

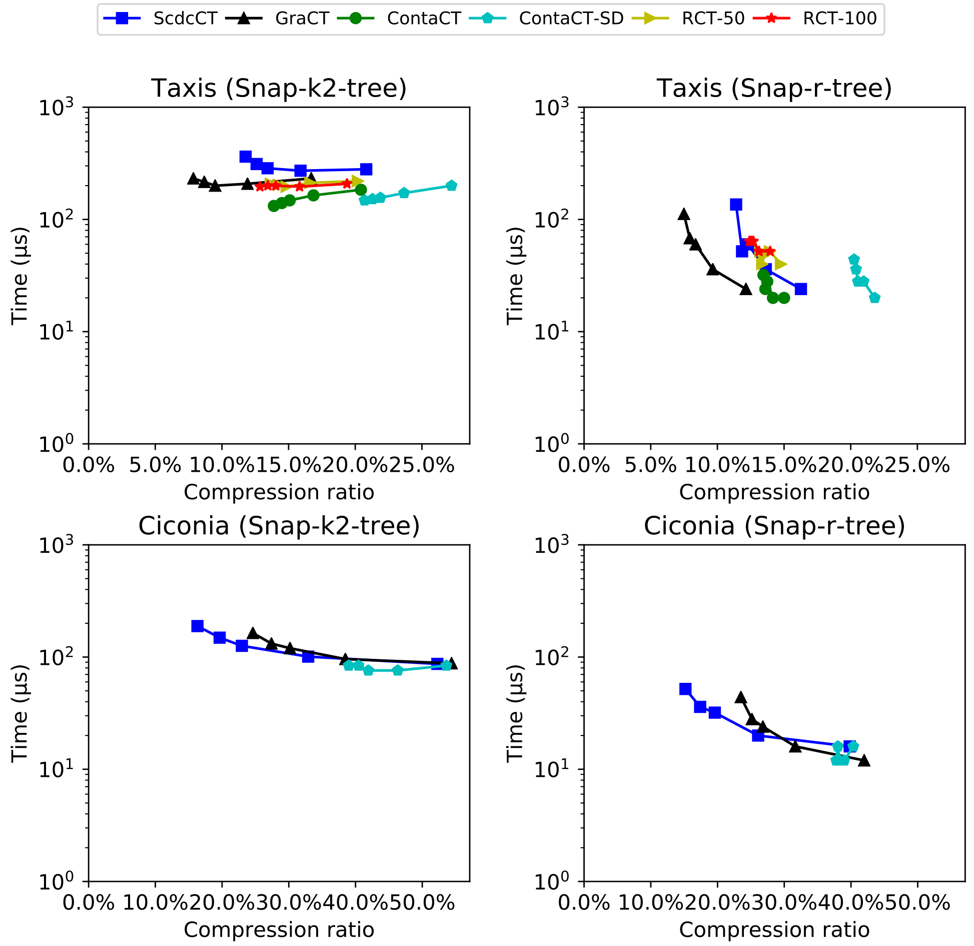


Figure 9.14: Time performance for *TimeInterval S* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

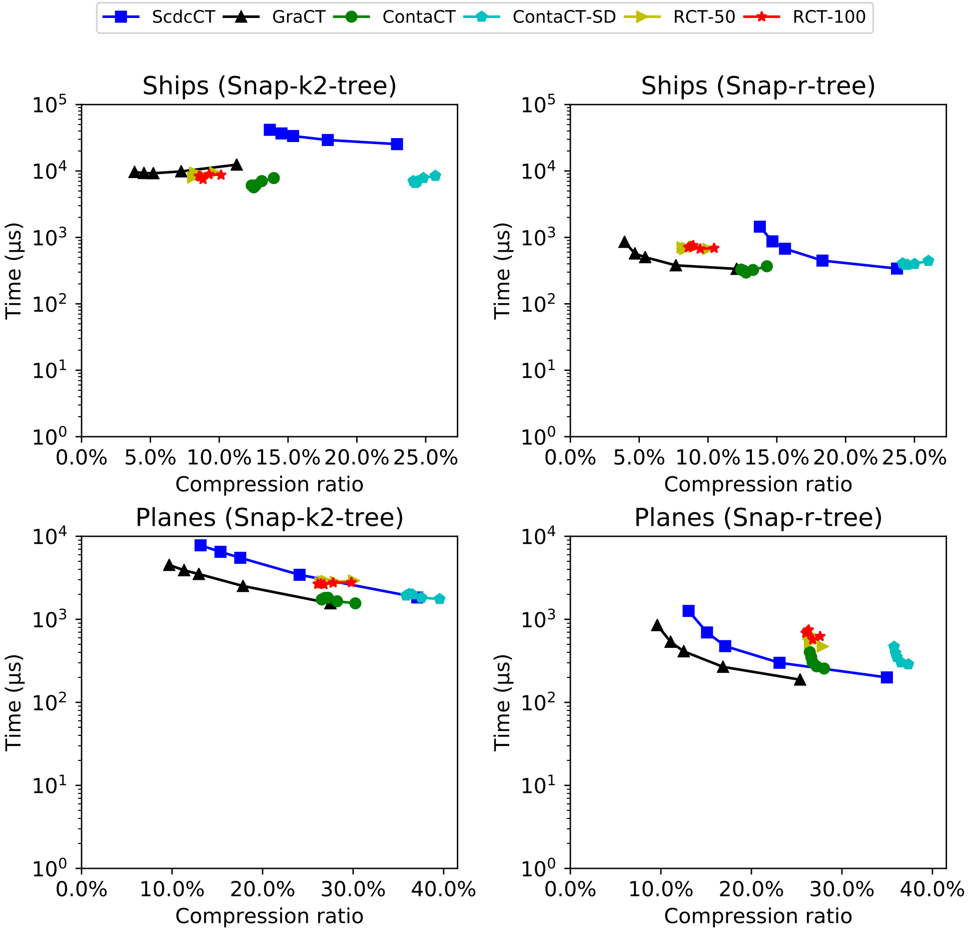


Figure 9.15: Time performance for *TimeInterval L* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

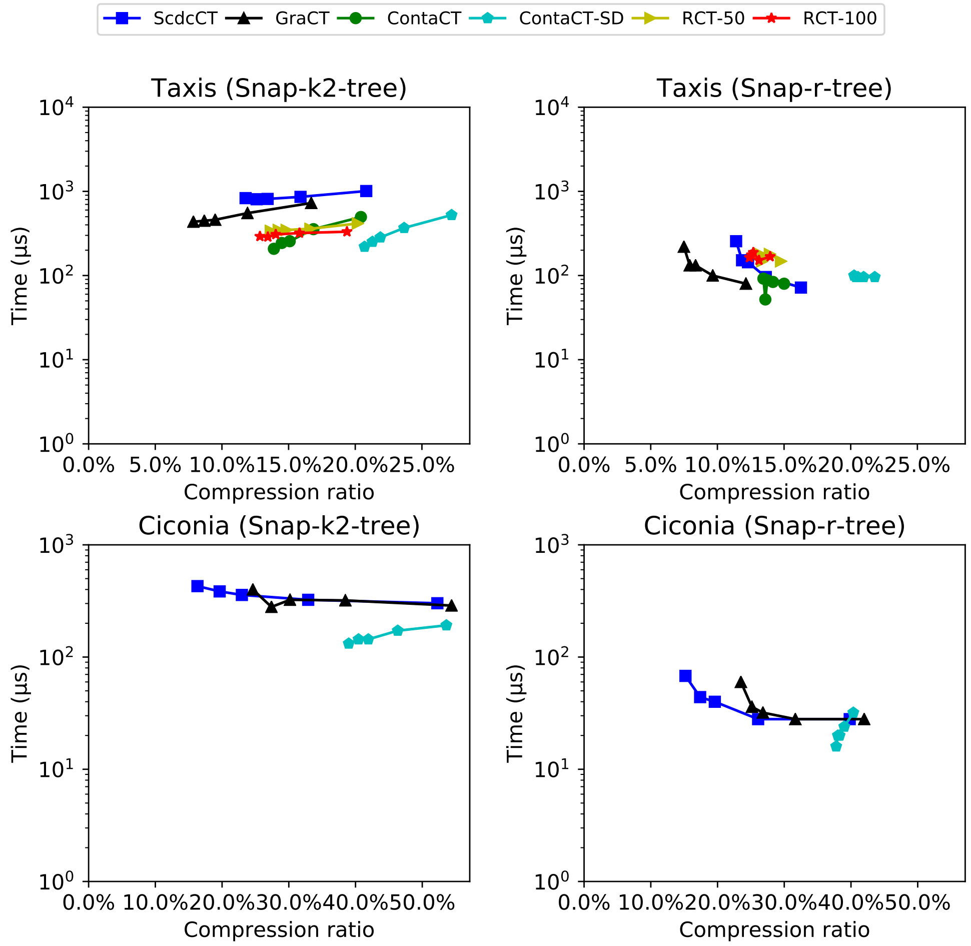


Figure 9.16: Time performance for *TimeInterval L* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

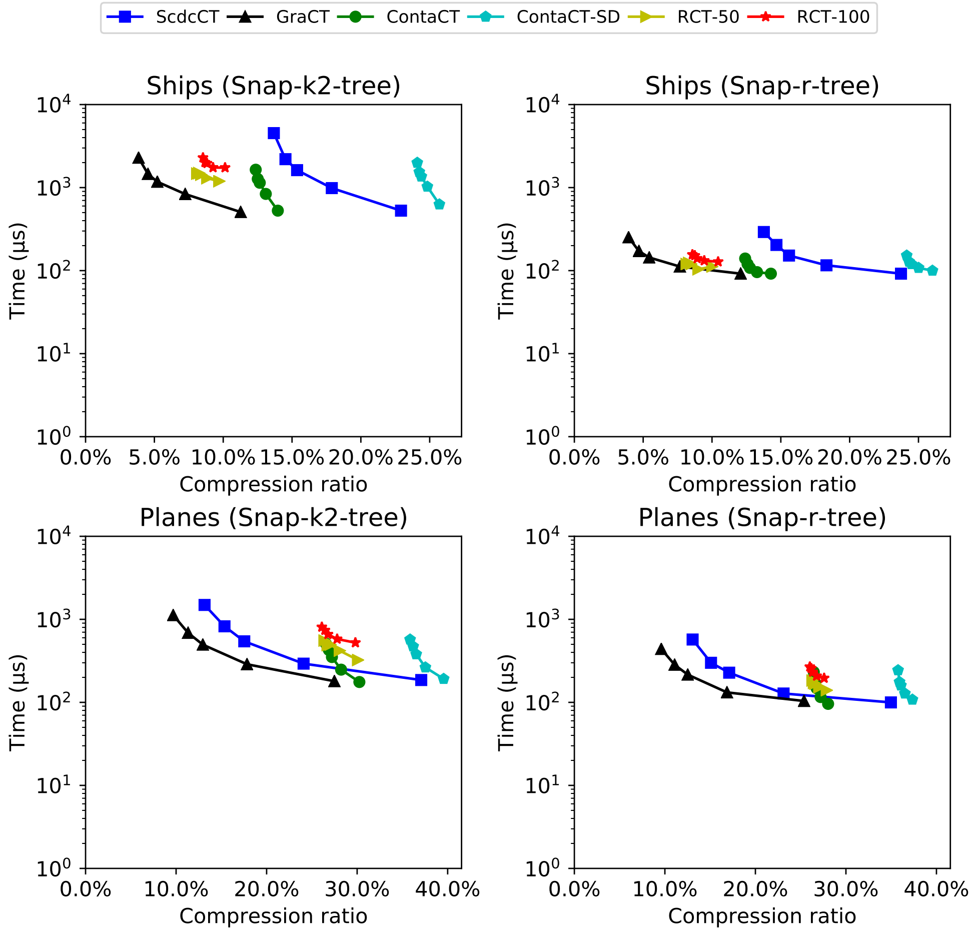


Figure 9.17: Time performance for *Knn* on `Ships` and `Planes` in microseconds. Notice the log scale in the vertical axis.

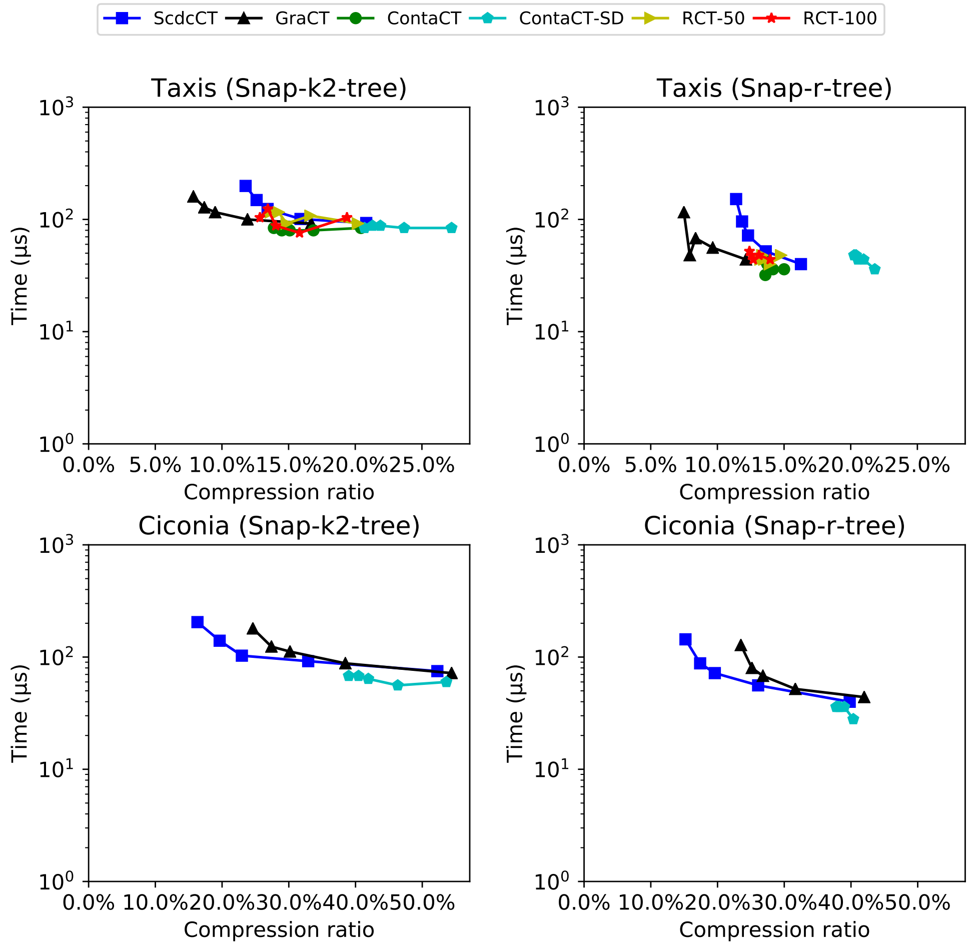


Figure 9.18: Time performance for *Knn* on `Taxis` and `Ciconia` in microseconds. Notice the log scale in the vertical axis.

of ContaCT worsen slower than those of GraCT and ScdcCT. That is expected because as d increases, GraCT and ScdcCT require scanning a larger section of the log in order to compute the location of each object. Since ContaCT can solve the position of the object in constant time, *Knn* is solved around 1.1–1.6 times faster, comparing the fastest configurations. RCT-50 and RCT-100 keep the constant time computation of the location at t_q, however, they still require synchronizing with the phrase that contains t_q. For this reason RCT-50 and RCT-100 are slower than ContaCT (20%–60%) being the second fastest structure in both `Ships` and `Taxis`.

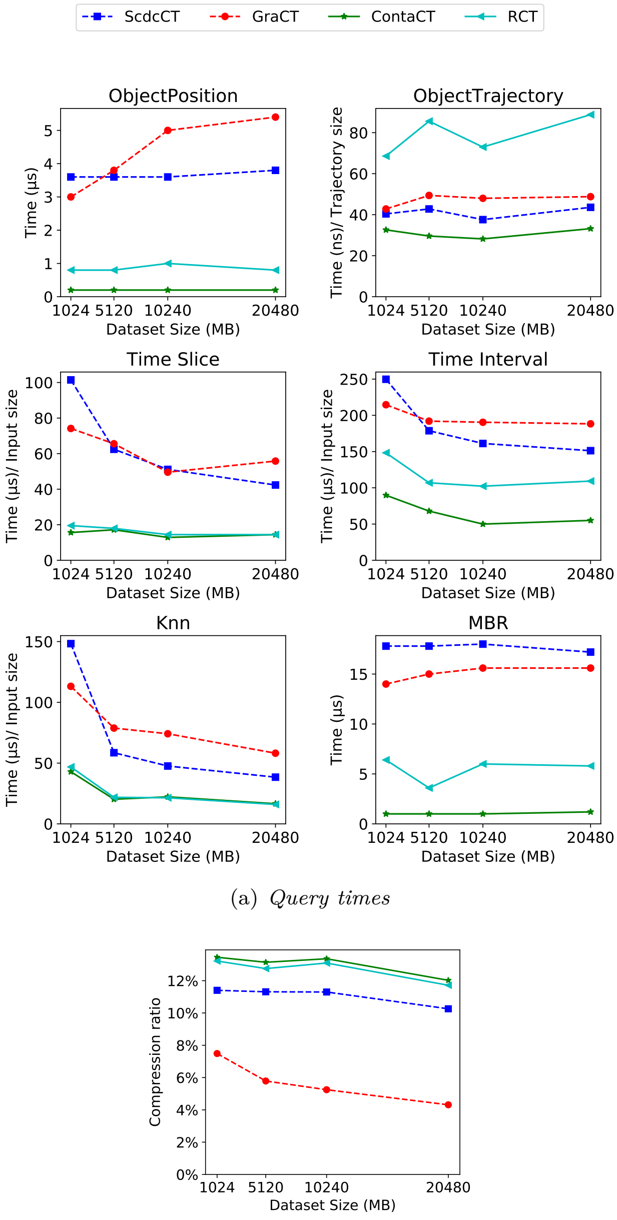
9.4 Scalability

We have studied the scalability of our structure in terms of query times and compression ratios by generating new datasets from the same source of `Taxis`. The size of those new datasets are 5,120 MB, 10,240 MB, and 20,480 MB in plain form. We chose the indexes with the best time performance on `Taxis`, that is, the log representations ScdcCT, GraCT, ContaCT, and RCT-50 combined with snapshots based on R-trees. We built those indexes on the resulting datasets with distance $d = 720$ between snapshots and compared all of them.

Figure 9.19(b) shows the compression ratios of those structures. We can observe as all the structures, except GraCT, slightly improve their compression ratios with respect to the smallest dataset of `Taxis`. Instead GraCT reduces the compression ratio to a 57.33% between the smallest to the largest dataset. That is expected, since the size of the dataset increases, the trajectories also grows, which makes it possible to find more repetitiveness between them and, consequently Re-Pair yields better compression. Although RCT-50 can exploit this redundancy of movements, the extra-space required for computing the queries in an efficient way makes it impossible to improve the compression ratios as fast as in GraCT. RCT-50 only improves the compression ratio by around 12%. On the other hand, the remaining structures do not exploit the repetitiveness, which explains the compression ratio does only improve by around 8%–10%.

Figure 9.19(a) shows the time performance of the structures with the different types of queries. In *ObjectPosition*, as expected, GraCT turns much slower (80% more) when the size of the dataset increases. Note that, in large datasets, GraCT can traverse faster the log, but it needs to decompress a deeper grammar when it reaches the queried time instant. ScdcCT only decreases its performance 5%, because the scanning of the log is limited by the distance between the snapshots $d = 720$. Instead ContaCT and RCT-50 solve it in constant time. A similar behavior can be observed in *MBR* queries where GraCT is a 15% slower and the remaining structures keep the response times rather constant.

Regarding *ObjectTrajectory* queries, we compute the quotient between the time and the size of the results obtained from the query, i.e. the number of points involved in the trajectory. RCT-50 is clearly the most irregular and slowest structure, because



(a) *Query times*

(b) *Compression ratios*

Figure 9.19: Evolution of query times and compression ratios as the dataset grows.

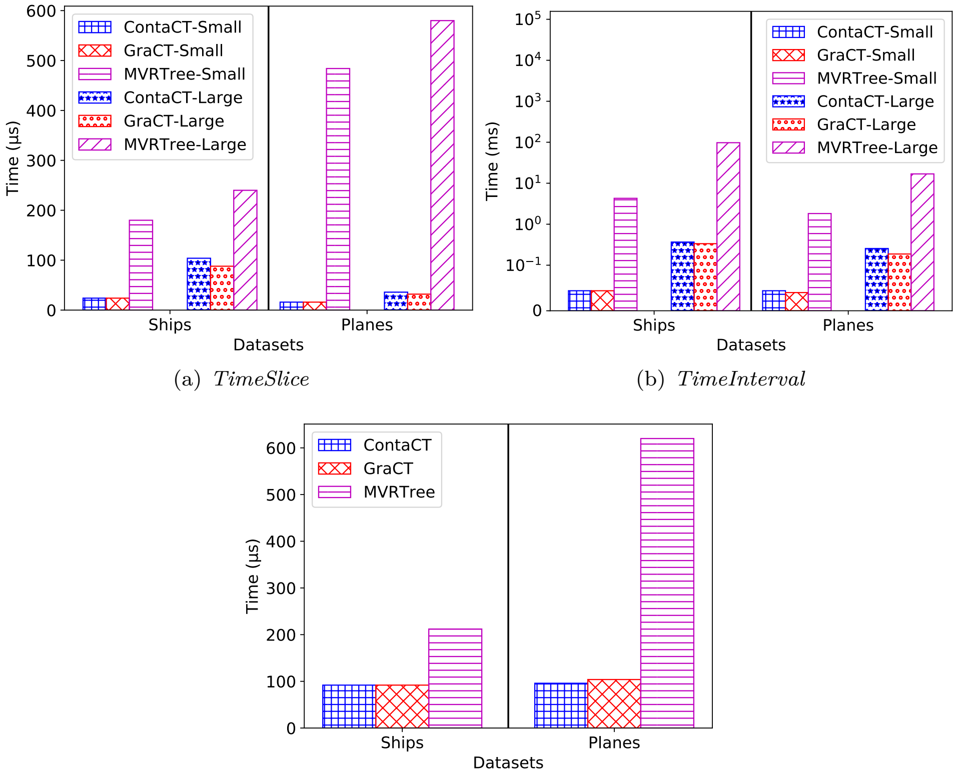
of its dependency on the number of phrases that split the trajectory. Retrieving a point of a trajectory shows a logarithm increase in GraCT as the height of the parse tree of its grammar increases. ContaCT retains constant time for the queries where it has $O(1)$ time complexity per point. Since computing a point in ScdcCT requires to decode a codeword, the times for each returned point take roughly the same amount of time.

In the remaining queries of Figure 9.19(a) (*TimeSlice*, *TimeInterval*, and *Knn*), the time is divided by the dataset size in MB. In all the structures, the queriy times tend to grow linearly with the data size. Notice that, as the dataset grows, the number of object candidates increases. GraCT is the most penalized technique in *TimeSlice* and *Knn* because of decompressing larger parse trees. In ScdcCT the number of read entries from the log is limited to d in all the datasets, thus the time reduces faster from 1GB to 5GB. Instead, in ContaCT and RCT-50 the times per input size are between 15–22 microseconds in both queries for the datasets of 5GB, 10GB, and 20GB. Regarding *TimeInterval*, Figure 9.19(a) shows a similar behavior for GraCT and ScdcCT, but there is a larger difference between ContaCT and RCT-50. Those latter structures use the same binary search through the MBRs to solve *TimeInterval*, but, as we pointed above, computing a MBR with RCT-50 is 4–7 times slower than in ContaCT. This overhead is extrapolated to *TimeInterval* making the response times of RCT-50 around 66%–98% slower than ContaCT.

9.5 Comparison with a spatio-temporal index

The structures that, in general, obtain the best compression ratios (GraCT) and the one with the best performance (ContaCT) with the snapshots based on R-trees are compared with a spatio-temporal index, the MVR-tree [TP01b]. As we explained in Section 3.1, the MVR-tree is composed of several R-trees, called *versions*. Each version corresponds with an interval of time. Since two consecutive versions have similar (or identical) nodes, the MVR-tree reduces the space by sharing common nodes between consecutive versions. This structure is focused on solving *TimeSlice*, *TimeInterval*, and *Knn* queries by traversing the R-trees or versions involved in the queried time interval. That traversal consists in going from the root down to the nodes by following those nodes that intersect the spatial point or region of the query. These structures do not support *ObjectPosition* and *ObjectTrajectory*, because it would require to traverse all the nodes of the versions involved on the query until obtaining the desired object.

In our experimental setup, we built MVR-trees on our two repetitive and real datasets, `Ships` and `Planes`, with sizes of 17.16 GB and 9.31 GB, respectively. Although MVR-tree was designed for residing on disk, for a reliable comparison, we configured it to run entirely in main memory. ContaCT and GraCT were configured with the same settings used in the previous experiments, fixing $d = 60$. Note that even using this maximum-space configuration (i.e. faster) of ContaCT, it still uses



(a) *TimeSlice*

(b) *TimeInterval*

(c) *Knn*

Figure 9.20: Query time comparison of ContaCT and GraCT that use snapshots based on R-tree with the MVR-tree, running in main memory.

260 times less space on `Ships`, and 90 times less space on `Planes`, than MVR-tree.

Figure 9.20 shows the times for the queries supported by the MVR-tree, comparing it with the maximum-space configuration of ContaCT and GraCT using the snapshots based on R-trees. *TimeSlice* and *Knn* queries are the most efficient queries for the MVR-tree, because it needs to traverse only one R-tree. However, the speed of querying the snapshots with an R-tree and the ability of ContaCT and GraCT to efficiently traverse the log allow both structures to outperform the MVR-tree in both types of queries. In `Ships`, our structures are 2–8 and 2–3 times faster in *TimeSlice* and *Knn*, respectively. With respect to `Planes`, the MVR-tree is surpassed by our structures, which are around 16–30 and 6–7 times faster in *TimeSlice* and *Knn*.

TimeInterval queries involve a larger interval of time, and thus MVR-tree needs to traverse multiple versions. These are the queries where ContaCT and GraCT

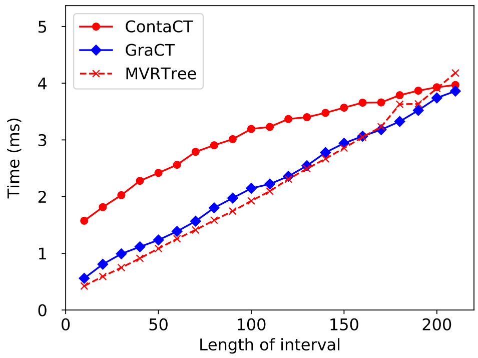


Figure 9.21: Growing *TimeInterval* queries on `Ships` where GraCT and ContaCT use snapshots based on k^2-tree.

outperform MVR-tree more sharply. Our structures can compute this query in around 40 and 400 microseconds with small and large regions, respectively. Instead, the MVR-tree needs 1–4 and 15–95 milliseconds to solve these queries, that is, the MVR-tree is around two orders of magnitude slower.

Regarding the structures that use k^2-trees as a part of the snapshot, we observed that *TimeSlice* and *Knn* queries are slower than in MVR-tree, but *TimeInterval* is faster. Notice that the difference between *TimeSlice* and *TimeInterval* is the span of the queried interval of time. In order to study the turning point of *TimeInterval* queries with small regions on `Ships`, we run queries varying the span of their time interval (see Figure 9.21). Despite both structures slow down as the length of the interval grows, the times of MVR-tree increase faster. Indeed, both ContaCT and GraCT with the snapshots based on k^2-trees outperform MVR-tree when the length of the time interval surpasses 170 and 210 time instants. Therefore, both compact data structures are less dependent on the span of the queried time interval, outperforming MVR-tree on long intervals.

9.6 Conclusions

In the experimental evaluation we can observe that the MVR-tree is slower than our combination of logs and snapshots based on R-trees. We can also see that if we uuuse snapshots based on k^2-trees, our structures are faster on *TimeInterval* queries when the span of the queried interval of time is large. Therefore, our structures obtain competitive and even better time performance than the MVR-tree and requiring around 90–260 times less space.

Although the snapshots based on k^2-trees were the first variant we designed, comparing the structures that use snapshots based on either k^2-trees or R-trees, we can observe that the latter ones are clearly faster to perform all the queries. The two main causes are that the snapshots based on R-trees can obtain the absolute position of an object in constant time, and provide a more accurate selection of the candidates in spatio-temporal range queries. Thus, we highly recommend using the snapshots based on R-trees.

Regarding the logs, GraCT obtains the best compression ratios except in `Ciconia`, which is not repetitive. In that case, the best compression is obtained with ScdcCT, which usually is the second structure that requires less space. Therefore, GraCT and ScdcCT are the most compressible logs. On the other side, we can observe that RCT uses the same or slightly less amount of space of ContaCT, which is the most-space consuming structure.

In time performance, ContaCT excels on object queries, where it obtains the best performance. RCT is the second best structure in *ObjectPosition* and *MBR*, where it is around by twice slower than RCT. GraCT is the third structure in those two queries, closely followed by ScdcCT. With respect to *ObjectTrajectory*, GraCT and ScdcCT outperform RCT because of the need for RCT to synchronize every phrase involved in the queried interval of time. On the other hand, the results of spatio-temporal range queries are similar for all the structures when $d = 60$. In case of increasing the value of d, the results get worse in ScdcCT and GraCT faster than in ContaCT, which is considered the best option in time performance when d is greater than 60.

Choosing the best log structure depends on the requirements of the domain. To summarize, the use of ContaCT and GraCT obtain the best time- and space-efficient structures, respectively. An intermediate option that provide a good space-time tradeoff is RCT. Instead, ScdcCT produces a good baseline that gets a good compression and time performance on those data that are not highly repetitive.

Chapter 10

Summary of contributions

During the development of this thesis, different works related with the *compact data structures* were done. Most of those works are based on the representation of moving object trajectories, which is the main topic of this thesis. However, other works that cover different lines of the compact data structures area were done and published in different conferences. Note that they are not included in the main discussion of this thesis because of coherence reasons. In the following sections, we reflect our contributions of three main domains: moving objects, two-dimensional block trees, and successor and predecessor problem.

10.1 Moving objects

10.1.1 Motivation

In the last years, the number of devices that track GPS information has increased considerably; we can find them in smartwatches, smartphones, cars, planes, ships, etc. Consequently, the amount of information about trajectories has increased exponentially. As these data grow, more applications have an interest in storing and exploiting these data, e.g., with data mining purposes. However, the large amount of data makes it difficult to manage this kind of applications, so finding new techniques for storing and accessing data efficiently is needed.

The classical research line is focused on spatio-temporal databases, that is, databases that store the data in the disk and keep an index on main memory to speed up the queries. However, they do not take advantage of the memory hierarchy. Therefore, we focus on exploiting that hierarchy by using compact data structures to represent moving object trajectories. As the space required of those structures is small, the entire structure can reside in the main memory and becomes faster in operations to access the information. Consequently, in some cases, answering queries on the compact data structure is even faster than performing that query over the plain representation.

In state of the art there are no compact data structures designed to store information about trajectories of moving objects and solve the queries of interest in that domain. For this reason we focus on the design and development of compact data structures of moving objects.

10.1.2 Description

We have designed different compact data structures to represent moving object trajectories, but all of them have two common goals:

- Storing in a reduced space the huge amount of data about spatio-temporal information, which describe moving object trajectories. Reducing the size of those structures allows us to take advantage of the memory hierarchy.
- Exploiting the stored information by supporting queries of two types: (i) object queries, which given an object and an interval of time retrieve information about the location or trajectory of an object during a period of time; (ii) spatio-temporal range queries, which given a region of space and an interval of time compute the objects that are within that region during the given interval of time. Solving both types of queries using the same structure was not supported for any structure of the state of the art.

To achieve those two goals, we assume a raster representation of the space and a discretization of the time, where the parameters that decide about the levels of normalization in space and time can be adjustable depending on the domain. Assuming that normalization, the trajectories are represented with different structures, but all of them, as a common nexus, use the same two elements:

- Snapshots: they work as a spatial index that are periodically distributed along time (e.g. every 1 minute). They can store the absolute position of the object or the area which covers the trajectory where the object is moving between two snapshots (MBR). Depending on the implementation, they store the absolute position or the MBR. In the case of the snapshots based on k^2-trees the structure stores the absolute position, and with the snapshots based on R-trees, they store the MBRs. The snapshots are used to speed up spatio-temporal range queries.
- Logs: each log corresponds with an individual object and stores the relative movements of that object, that is, the displacements on the space every two time instants. With the help of this structure we can retrieve the cumulative movement between two time instants, i.e. they allow us to retrieve the actual trajectory of the object. We have designed and implemented four different log structures: ScdcCT, GraCT, ContaCT, and RCT. Each one uses a different

compression strategy: statistical compression, grammar compression, bitmaps and relatitve compression, respectively.

The combination of those two elements, which includes two variants for snapshots and four for logs, gives us a total of eight different structures that permit the representation of moving object trajectories. All of them were experimentally evaluated, and they show good compression ratios and competitive time performance when compared with classical spatio-temporal indexes. In addition, our structures allow us to support the two kind of queries at a same structure, something impossible for the indexes in state of the art. Apart from that, we can obtain different space-time tradeoffs and a better adaptation to the application domain by choosing the implementations that compose our structure.

10.1.3 Conclusions

We have built eight structures for representing moving objects trajectories, and we have achieved the two proposed goals: storing the data in reduced space and exploiting that information by supporting object and spatio-temporal range queries.

After evaluating all the eight structures we have observed that our structures obtain competitive times in spatio-temporal range queries when we compare them with a spatio-temporal index like the MVR-tree. Indeed, comparing them we conclude the following:

- We have presented two different representations of snapshots, one based on k^2-trees, and another one based on R-trees. In the experimental evaluation, we show that those structures that use the second type of snapshot obtains the best performance in all the queries. For example, the structure with snapshots based on R-trees can solve *TimeSlice* queries 16–55 times faster, than the other type of snapshot.

- We have proposed four representations of the logs. The first structure is ScdcCT, which compresses the movements by using a statistical zero-order compressor that assigns shorter codewords to small movements. ScdcCT can be considered a good option to compress trajectories when they are not highly repetitive. GraCT is the second type of log, and it was designed to exploit the repetitiveness of movements between trajectories. Therefore, it turns into the log that obtains the best compression ratios and competitive times with the other structures. The third log is ContaCT, which is compressed by using a partial-sums structure, making it possible to compute the position of an object in constant time. It is the option that obtains the best time performance. The last variant of the log is RCT, which is based on relative compression and tries to take the advantages of the most compressible structure, GraCT, and the time performance of ContaCT.

10.1.4 Future work

Although these structures were completely implemented and tested, there are some extensions that can improve this thesis's contribution. Future research lines include:

- We plan to extend the supported queries of our structures. As we show, we can solve all the queries of a classical spatio-temporal index, but in the state of the art, there are other sets of queries focused on data mining. Supporting queries like moving together patterns, which obtain those objects that are moving through the same locations at the same time; or trajectories clustering, which look for the representative patterns of a set of trajectories. The use of the MBR query can be the basis to solve those queries.

- Since RCT is not as close as we expected to GraCT, we plan to study different ways of building the reference in order to obtain better compression ratios. One interesting point is to build the reference from the grammar tree obtained from compressing the union of trajectories with any grammar compression technique.

- Notice that using a raster model, we are assuming some loss of precision. In some domains, that loss is not important, for example, we do not need to know the location with a precision of meters in trajectories of planes. However, in some cases storing the location of the objects in full-precision is required, thus we plan to use this type of indexes as an in-memory cached index. That index can solve the queries with a loss of precision, and a disk-based structure refines the final result.

- We observed that the trajectories could be considered as temporal series where the value of each time instant is a location. Therefore, we believe that these structures can be extended to other domains like the stock-market.

10.2 Two-Dimensional Block Trees

10.2.1 Motivation

In many applications, image collections contain identical sub-images, for example two-dimensional slices of three-dimensional scans, video frames, and periodical sky surveys. This is an important source of redundancy that can be exploited for compression. Such an approach is found in two-dimensional Lempel-Ziv[LZ86] (2D-LZ), which stores only the first occurrence of each sub-image on the dictionary and the others are represented as pointers to the reference. However, 2D-LZ does not support efficient random access to individual images or arbitrary regions thereof. This is a relevant problem when storing large image collections in compressed form. Some proposals [PW96, AF00] provide direct access by splitting the image into different partitions, and solving range queries by decompressing only those parts that intersect with the queried region. This induces, however, a tradeoff between extraction time and compression ratio, driven by the size of those partitions.

Other data like matrices, maps, and graphs, are also modeled as images and may contain similar areas. A particular case of repetitive two-dimensional data are *Web graphs*, which are directed graphs of pages pointing to other pages on the Web. The adjacency matrix of a Web graph can be seen as a bilevel image, where the link between pages a and b is represented with a 1 at position (a, b), which has a 0 otherwise. Web graphs are sparse, so this matrix has large zones of 0s and a few 1s.

Since the adjacency matrix is huge and needs efficient random access, the design of compact data structures to represent Web graphs is a relevant topic. A well-known such structure is the k^2-tree [BLN13]. The k^2-tree is very efficient at representing large zones of 0s of the adjacency matrix and supporting direct and reverse neighbor queries. While there are other representations that exploit other properties of Web graphs (locality, similarity of adjacency lists, etc.) [HN14, GB11, BV04], the k^2-tree offers the best space-time tradeoff when considering both direct and reverse neighbor queries. However, the k^2-tree does not directly exploit repetitiveness.

10.2.2 Description

A recent compact data structure called Block Tree (BT) [BGG$^+$15] compresses repetitive collections of (one-dimensional) strings. It obtains compression ratios close to Lempel-Ziv [ZL77] while supporting efficient direct access to any substring. The BT overcomes the inability of Lempel-Ziv in providing direct access by imposing a regular structure to the targets of string copies. The BT reduces the size of repetitive collections by orders of magnitude.

In order to apply the features of BT to images and matrices, we extend the BT to two dimensions. The result is called *Two-Dimensional Block Tree (2D-BT)*. It is designed to compress two-dimensional elements like matrices, images, or graphs. We presented a general 2D-BT structure and a specific 2D-BT variant to compress Web graphs. This is a hybrid with the k^2-tree that exploits the clustering of 0s and, at the same time, the repetitiveness of the adjacency matrix.

Given a matrix M of size $|M| = n^2$ over an alphabet $[1..\sigma]$, the matrix is subdivided into k^2 submatrices of size n^2/k^2. Each of these submatrices is called a block and represents a node of the 2D-BT. The nodes can be classified into internal or leaves. Consider any submatrix order, such as the row-major one used by the k^2-tree. Then nodes whose submatrix overlaps the first occurrence of a block (including themselves) are internal nodes; the others are leaves. The submatrix of any leaf node is said to be the *target* of a copy, whose *source* is its first occurrence. A source may overlap up to four adjacent blocks. Each leaf stores a pointer *ptr* to the top-left block that includes its source and two offsets O_x and O_y, one by axis, where the source starts in that block. Once the first level is built, we split the internal nodes into k^2 new nodes, and add them as children of the corresponding internal node. This step is repeated recursively until storing a pointer and its offsets is more expensive than storing the submatrix of a node. At this point, the submatrix content is stored verbatim. The 2D-BT has a maximum height of $height = \log_k n$.

To handle, in particular, Web graphs, we specialize this general 2D-BT structure so as to exploit clustering and sparseness, not only repetitiveness. We regard the adjacency matrix as a binary image. We define a new kind of leaf called *empty node*, which represents a block of all 0s. Therefore, leaves in this 2D-BT may be empty nodes, pointers to sources, or last-level nodes storing individual cells.

10.2.3 Conclusions

We have proposed a new structure that extends Block Trees to two dimensions, and combined them with k^2-trees to handle in particular Web graphs. In the experimental evaluation, where we compare the Two-Dimensional Block Tree with the k^2-tree on web graphs, we conclude the following:

- We obtained up to 50% of the space of k^2-trees, the best structure that allows navigating the graph in both directions.

- The price is that we are 3–6 times slower. This price can be irrelevant when the lower space allows fitting the whole graph in a faster memory (e.g., RAM vs disk).

10.2.4 Future work

Our most immediate future work is to improve the construction, in order to handle full Web graph adjacency matrices. The current construction takes too much space and time because of using 2D signatures stored in a large matrix. We plan to replace the 2D signature based scheme by a randomized method based on sampling positions in the submatrix.

We also plan to explore other application areas where the values of the matrix display a good deal of repetitiveness like three-dimensional scans, bi-level images or periodical sky surveys.

10.3 Successor and predecessor problem

10.3.1 Motivation

One of the main computational tasks in a search engine is to look for those documents that contain a set of words. In order to speed up that search, those engines use inverted lists. Each inverted list corresponds to a word and stores as an increasing sequence the document identifiers of the documents where that word occurs. Most of the time, the query received by a search engine carries more than one word, to know where all the words appear together, the search engine needs to intersect various inverted lists. The optimal intersection of two lists can be easily solved by iterating over both of them in alternate form [CM10] (merge-wise intersection). In each iteration, the search engine looks for a value in the second list, v_2, equal to or higher than the value from the first list, v_1. If they are identical, v_1 is part of the solution and iterates to the next value in the first list. Otherwise, the iterator of the first list skips those values lower than v_2. Therefore, it needs an efficient mechanism that can find an equal or higher value in the other list, which is known as the successor problem.

Let us formalize the successor and predecessor problem, considering a set of integers $S = \{x_1 < x_2 < \cdots < x_m\}$, the successor ($succ(x) = x_i$) of a given value x returns the minimum value $x_i \geq x$ of S. Analogously, the predecessor of x ($pred(x) = x_i$) returns the maximum value $x_i \leq x$ of S. Assuming $n = x_m$ and $m = |S|$, both problems can be modeled by using a bitmap $B[1, n]$ which contains m 1s located at positions x_i for all $1 \leq i \leq m$, and solved in $O(1)$ time with the two classical operations on bitmaps: *rank* and *select* [Jac89, Mun96, Cla96].

In some scenarios, the bitmap B can contain the set bits clustered together in k runs; hence B contains k runs of 1s and $k \pm 1$ runs of 0s. There is a structure, *oz-vector* [Nav16], which compresses the bitmaps exploiting its runs. The *oz-vector* transforms the input bitmap into two sparse bitmaps O and Z, which mark the lengths of the runs of 1s and 0s, respectively. Since those bitmaps are sparse, they are very compressible, and the *oz-vector* can obtain good compression ratios in practice. However, for solving *succ* and *pred*, it requires $O(\log k)$ time.

10.3.2 Description

We have proposed a new structure, *zombit-vector*, which compresses bitmaps with runs and supports *succ* and *pred* in $O(1)$ time. The main idea is to divide the input into blocks in such a way that most of the blocks are uniform (all 0s or all 1s). With this approach, our structure only needs to store the information contained by non-uniform blocks.

Given a bitmap B of size $|B| = n$ with k runs of 1s and $k \pm 1$ runs of 0s, *zombit-vector* defines a size of block β which splits B into $\lceil \frac{n}{\beta} \rceil$ partitions, obtaining a set of blocks $\{X_1, X_2, \ldots, X_{\lceil \frac{n}{\beta} \rceil}\}$. Each block X_i can be classified into three different sets of blocks depending on its data: uniform blocks full of 0s ($\mathcal{Z}$), uniform blocks full of 1s ($\mathcal{O}$), and mixed blocks, those which contain both bits ($\mathcal{M}$). As a consequence, the structure contains $u = |\{\mathcal{Z} \cup \mathcal{O}\}|$ uniform and $t = \lceil \frac{n}{\beta} \rceil - u$ mixed blocks. The classification of each block can be represented by using two plain bitmaps: U and O. The bitmap $U[1, \lceil \frac{n}{\beta} \rceil]$ marks which X_i is uniform by setting $U[i] = 1$ when $X_i \in \{\mathcal{Z} \cup \mathcal{O}\}$. Then, we use the bitmap $O[1, \lceil \frac{n}{\beta} \rceil]$ to represent which block contains at least one 1-bit, it means $O[i] = 1$ when $X_i \in \{\mathcal{O} \cup \mathcal{M}\}$. Additionally to this classification, we need to store the data of every mixed block. For that purpose, we use a bitmap $M[1, t \times \beta]$ which appends the information of each mixed block together, preserving the order in B. In total, we need $O(k\beta + \frac{n}{\beta})$ bits, and since the optimal $\beta = \sqrt{\frac{n}{k}}$, we can reduce the space to $O(\sqrt{kn})$ bits.

10.3.3 Conclusions

We have proposed a structure, *zombit*, which compresses bitmaps with large runs and can solve *access*, *rank* and, *successor/predecessor* queries in $O(1)$ time. In the experimental evaluation, where we evaluate the compression and time performance over bitmaps with different length of runs, we conclude the following:

- We obtained a compression ratio of $0.04\% - 26.33\%$, when the length of runs is larger than 100 and we can handle successor queries $3 - 12$ times faster than our immediate competitor. Consequently, *zombit* gets a good trade-off in terms of space and time on bitmaps with runs.
- A variant of our structure to obtain better compression was introduced, but in practice, it is $5-12$ times slower than *zombit*, and it reduces to $68.25\%-87.06\%$ the space of our first proposal.

10.3.4 Future work

As future work, since we do not beat the space of our competitors in shorter runs, we will focus on improving the compression in that scenario. We plan to solve *select* operations on *zombit* efficiently by using $o(n)$ extra-space.

Also, we will explore other areas where the successor and predecessor problem is relevant.

Appendix A

Publications and other research results

Publications

Journals

- Brisaboa, N. R.; Gagie, T.; Gómez-Brandón, A.; Navarro, G.; Paramá, J. R.: **An index for moving objects with constant-time access to their compressed trajectories.** *Submitted to International Journal of Geographical Information Science.*

- Brisaboa, N. R.; Gómez-Brandón, A.; Navarro, G.; Paramá, J. R.: **GraCT: A Grammar-based Compressed Index for Trajectory Data.** *In Information Sciences, 483, Elsevier, New York (Estados Unidos), 2019, pp. 106-135.*

International conferences

- Gómez-Brandón, A.: **Bitvectors with runs and the successor/predecessor problem.** *In Proceedings of the 2020 Data Compression Conference (DCC 2020) IEEE Computer Society, Snowbird, Utah (United States), 2020, pp. 133-142.*

- Brisaboa, N. R.; Fariña, A.; Gómez-Brandón, A.; Navarro, G.; Varela Rodeiro, T.: **Dv2v: A Dynamic Variable-to-Variable Compressor.** *In Proceedings of the 2019 Data Compression Conference (DCC 2019), IEEE Computer Society, Snowbird, Utah (United States), 2019, pp. 83-92.*

- Brisaboa, N. R.; Gagie, T.; Gómez-Brandón, A.; Navarro, G.: **Two-Dimensional Block Trees.** *In Proceedings of the 2018 Data Compression*

Conference (DCC 2018), IEEE Computer Society, Snowbird, Utah (United States), 2018, pp. 227-236.

- Brisaboa, N. R.; Gómez-Brandón, A.; Martínez Prieto, M.A.; Paramá, J. R.: **3DGraCT: A Grammar based Compressed representation of 3D Trajectories.** *In Proceddings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE 2018) - LNCS 11147, Springer, Lima (Perú), 2018, pp. 102-116.*

- Brisaboa, N. R.; Gagie, T.; Gómez-Brandón, A.; Navarro, G.; Paramá, J. R.: **Efficient Compression and Indexing of Trajectories.** *In Proceddings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE 2017), LNCS 10508, Springer, Palermo (Italy), 2017, pp. 103-115.*

- Brisaboa, N. R.; Gómez-Brandón, A.; Navarro, G.; Paramá, J. R.: **GraCT: A Grammar based Compressed representation of Trajectories.** I*n Proceddings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016) - LNCS 9954, Springer , Beppu (Japan), 2016, pp. 218-230.*

- de Bernardo, Guillermo; Casares, R.; Gómez-Brandón, A.; Paramá, J. R.: **A new method to index and store spatio-temporal data.** *In Proceddings of the 20th Pacific Asia Conference on Information Systems (PACIS 2016), AIS Electronic Library (AISeL), Chiayi (Taiwan), 2016.*

International research stays

- *6th March, 2016 - 6th June, 2016.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

- *12th October, 2016 - 11th November , 2016.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

- *9th April, 2018 - 8th July, 2018.* Research stay at University of Melbourne, School of Computing and Information Systems (Melbourne, Australia).

- *11th October, 2019 - 12th December, 2019.* Research stay at University of Kyushu, Department of Informatics (Kyushu, Japan).

- *6th March, 2020 - 13th March, 2020.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

Appendix B

Resumen del trabajo realizado

En este capítulo se presenta un resumen del trabajo realizado durante la tesis. En la sección B.1 se presenta una breve introducción y la motivación para la realización de esta tesis. Además, se resume brevemente la área donde se desenvuelve la tesis, indicando cada uno de los principales problemas que tratamos de solucionar mediante nuestras contribuciones. En la sección B.2 se exponen y discuten cada una de las estructuras y algoritmos desarrollados. En la sección B.3 se presentan las conclusiones a las que se llegaron tras el desarrollo de la tesis. Finalmente, este capítulo se cierra con la sección B.4, donde se abordan diferentes líneas de investigación para mejorar y ampliar en un futuro nuestras contribuciones aquí expuestas.

B.1 Introducción

Durante los últimos años, con la introducción de sensores GPS en todo tipo de dispositivos personales, (móviles o relojes) y en medios de transporte (coches, aviones o buses) la cantidad de información sobre trayectorias de objetos móviles ha crecido exponencialmente. Al mismo tiempo que estos datos crecen, nacen nuevas aplicaciones que necesitan almacenar el histórico de cada una de las trayectorias y analizarlas de una forma eficiente, por lo que son considerados parte de un nuevo campo denominado Big Data. De estos problemas de almacenamiento y explotación de abundantes cantidades de datos surgen nuevos retos, ya que las estructuras de datos y los algoritmos convencionales no están diseñados para tratar el gran volumen de información. El objetivo principal de esta tesis es proporcionar nuevas estructuras de datos y algoritmos que puedan reducir el tamaño necesario para guardar estos datos y, sin la necesidad de incrementar ese tamaño, poder explotar su información de manera eficiente.

En la actualidad, para la explotación de los datos de trayectorias móviles, existen las bases de datos espacio-temporales. Estas bases de datos surgieron para almacenar datos sobre posiciones de objetos en diferentes instantes de tiempo y poder consultarlos de forma eficiente. Tanto estas bases de datos como los sistemas de información geográfica han sido muy estudiados y son hoy una tecnología madura tanto a nivel académico como industrial. Pero, la investigación en estas bases de datos ha renacido por la proliferación de dispositivos con GPS. Por otro lado, con el avance de la tecnología y la aparición de nuevos sistemas que se centraban en la distribución y paralelización de los procesos, aparecen un nuevo tipo de estructuras y algoritmos con un enfoque completamente diferente al convencional. Entre ellas destaca una nueva línea de investigación que se centra en mantener los datos en memoria principal de forma comprimida y acceder a ellos sin la necesidad de descomprimirlos, es lo dominado, estructuras de datos compactas.

Las estructuras de datos compactos surgieron dentro del campo de la compresión de datos, cuyo principal objetivo era reducir el tamaño destes datos y el ancho de banda usado para su transmisión. Sin embargo, esas técnicas de compresión tradicionales, no permiten consultar esos datos cuando están comprimidos. Descomprimir esos datos es costoso, tanto en tiempo como en espacio de almacenamiento. Las estructuras compactas surgen para cambiar la idea clásica de compresión, ya que permiten el acceso y la ejecución de consultas complejas directamente sobre los datos comprimidos, sin la necesidad de recuperar los datos originales mediante un proceso de descompresión. Al reducir su tamaño, estas estructuras están pensadas para residir en memoria principal, aprovechando la ventaja de acceso que ésta le ofrece, mucho más rápido que en almacenamiento secundario. Como consecuencia, muchas de las estructuras de datos compactas son más rápidas que su representación sin compresión.

B.1.1 Motivación

La principal motivación de esta tesis es el diseño, implementación y evaluación experimental de estructuras de datos compactas para la representación de trayectorias de objetos móviles. En el mundo de la representación de trayectorias, podemos distinguir dos tipos: trayectorias en espacios abiertos y trayectorias restringidas a redes, como puede ser la red de carreteras. EN nuestro caso nos centramos en el primer tipo de representación, por lo que la trayectoria es considerada el camino realizado por un objeto a lo largo del tiempo. Debido a los requisitos de almacenamiento y las limitaciones de los dispositivos GPS que adquieren esas posiciones, el movimiento continuo de esos objectos se aproxima guardando la localización del objeto en intervalos regulares de tiempo.

En la actualidad, el medio para almacenar las trayectorias de objetos móviles son las bases de datos espacio-temporales. Estas bases de datos guardan todos los datos en almacenamiento secundario y sobre ellos se construyen índices que permiten acelerar la explotación de su información. Aunque hay estructuras que mantienen el índice en disco junto a los datos, lo más común es que los índices se mantengan en memoria principal. Estas estructuras se conocen como índices espacio-temporales y se centran en resolver consultas del tipo: rango espacio temporal y de objetos cercanos a un punto dado. La primera se centra en obtener todos los objetos que están dentro de una región en un intervalo de tiempo y la última obtiene los objetos más próximos a un punto del espacio. Sin embargo, estos índices no puede recuperar la trayectorias completas de los objetos de forma eficiente y requiere el uso de espacio adicional.

Ninguna de las técnicas anteriormente presentadas considera el uso de compresión. Sin embargo, existen algunas técnicas de compresión usadas usadas en trayectorias, pero que no permiten su indexación, como puede ser la simplificación de trayectorias o compresión de diferencias. La simplificación de trayectorias es una técnica que solamente guarda aquellos puntos de la trayectoria que considera necesario y descarta el resto, es decir, es una técnica de compresión con pérdida. Por lo tanto, existe una pérdida de información al comparar la trayectoria descomprimida con la trayectoria real. La técnica más común para comprimir las trayectoria es la basada en diferencias, es decir, cada nueva posición es almacenada como la diferencia con la posición previa. Esta idea explota el hecho de que de dos posiciones consecutivas de una trayectoria se espera que estén cerca y que las diferencias sean pequeñas. Cuando las diferencias son pequeñas, menos cantidad de bits son necesarios guardar, mejorando la compresión. Recuperar la trayectoria completa se puede realizar eficientemente recorriendo todas las diferencias, pero para acceder a un elemento de la trayectoria sin reducir el rendimiento necesita guardar información adicional.

Este estado actual, tanto en indexación como en compresión de trayectoria, nos presenta dos retos principales: debemos reducir el tamaño requerido para almacenar colecciones de objetos y, al mismo tiempo, proporcionar un índice que soporte tanto recuperar las trayectorias originales como realizar las consultas soportadas por los índices espacio-temporales. Hasta donde sabemos, no existen antecedentes de estructuras de datos compactas diseñadas específicamente para tratar trayectorias de objetos móviles.

Ante este escenario los objetivos de esta tesis son los que siguen:

- Diseñar estructuras de datos compactas que reduzcan el tamaño de la representación de los objetos y los algoritmos necesarios para poder recuperar las trayectorias originales sin ningún tipo de pérdida de información.

- Diseñar los algoritmos necesarios para poder resolver de una forma eficiente las consultas relacionadas con los índices espacio temporales: (i) time-slice, computar los objetos dentro de una región en un instante de tiempo; (ii) time-interval obtener los objetos dentro de una región en algún instante de tiempo de un intervalo de tiempo; (iii) knn, obtener los objetos más próximos a un punto del espacio.

- Evaluación de las estructuras y los algoritmos para resolver las consultas. En esa evaluación uno de los objetivos es ver como escalan las estructuras según se incrementa el tamaño de la colección de datos.

- Mostrar que las estructuras de datos compactas aportan un nuevo enfoque al mundo de representación de trayectorias, que les hace ser competitivas con los índices espacio-temporales clásicos.

B.2 Contribuciones

En esta tesis nos hemos centrado en la representación compacta y eficiente de colecciones de datos de trayectorias de objetos que se mueven en un espacio abierto sin restricciones. El conjunto de estas nuevas estructuras compactas y algoritmos buscan completar los objetivos presentados en el apartado anterior. Esta sección resumen las contribuciones más importantes de esta tesis.

Todas las estructuras presentadas representan las trayectorias en un modelo raster, es decir, las coordenadas que indican la localización donde está un objeto en un instante de tiempo se representan como celdas que dividen el espacio en partes iguales. El tamaño de estas celdas es ajustado en función del dominio al que pertenecen las trayectorias. Nótese que al reducir el tamaño de las celdas la representación de las trayectorias es más fina. Por lo tanto se entiende que una trayectoria es una lista de celdas e instantes de tiempo, que indican por donde ha pasado el objeto a lo largo del tiempo. Para su representación, todas las estructuras presentadas parten de los dos componentes básicos:

- *Snapshot.* Este componente se encarga de guardar la información espacial sobre los objetos a intervalos regulares de tiempo, lo que permite que se resuelvan una serie de operadores espaciales que nos dan una orientación por donde ha estado moviendose el objeto.

- *Logs.* Cada log pertenece a un objeto y guarda todos los puntos por donde éste ha pasado. Cada uno de esos puntos es representado como el desplazamiento entre el nuevo punto y el anterior. En los logs se poden resolver lo que denominamos operaciones de log.

Para cada uno de estos componentes, esta tesis propone diferentes estructuras, cada una con sus ventajas y desventajas. En el caso de la snapshot, se han diseñado dos estructuras:

- *Snapshots basadas en k^2-trees.* Este tipo de estructura considera el modelo raster y transforma la matriz que representa el espacio en una matriz binaria, donde las celdas que contienen un objeto son marcadas con un 1 y el resto con un 0. Consecuentemente se obtiene una matriz donde la mayoría de las celdas están vacías. Esta matriz es comprimida con un k^2-tree y con ayuda de estructuras compactas auxiliares, se guarda que objetos están en cada una de las celdas marcadas con un 1.

- *Snapshots basadas en R-trees.* A diferencia con el tipo de snapshot anterior, en estas snapshots se guarda para cada objeto, el rectángulo mínimo que cubre su trayectoria entre dos snapshots consecutivas. Para guardar esa información de forma compacta, se utiliza una variante compacta de un índice espacial denominado R-tree.

Con respecto al log, se han diseñado cuatro estructuras, donde cada una de ellas explota diferente características para la compresión de los mismos.

- *ScdcCT*, es una representación del log que explota el hecho de que los movimientos pequeños son más frecuentes que los desplazamientos grandes. Para ello usa un compresor denominado (s,c)-Dense Codes, que asigna códigos más cortos a los valores más pequeños.

- *GraCT* utiliza una compresión basada en gramáticas llamada Re-Pair. Es decir, considera el log como una sequencia de valores que comprime en una sequencia final donde no hay ningún par idéntico repetido. Básicamente busca patrones de movimiento idénticos y cada uno de esos patrones son representados con un único código. Esto permite explotar la repetividad de movimientos entre todas las trayectorias, aumentando la compresión. Para agilizar las consultas, se guarda información adicional para cada uno de estos códigos.

- *ContaCT* a diferencia de las dos estructuras de log anteriores, esta estructura está más centrada en poder realizar las consultas sobre el log de forma eficiente que en la compresión. Los movimientos del log son representados usando varios bitmaps, que usando sumas parciales, pueden calcular en tiempo constante el desplazamiento de un objeto para un intervalo dado.

- *RCT*, en este caso todos los movimientos del log son representados usando compresión relativa. Hay un log artifical de referencia que contiene los movimientos más comunes de la colección y el resto de logs se codifican con respecto a esa referencia. Su principal objetivo es poder mantener la eficiencia de ContaCT y acercarse a la compresión de GraCT.

Por lo tanto, en total existen ocho posibles estructuras de datos compactas para la representación de trayectorias, fruto de la combinación de las estructuras diseñadas para cada componente. Cada una de esas combinaciones, gracias a las operaciones espaciales y de log, soportan tanto operaciones relacionadas con la trayectoria (*ObjectPosition ObjectTrajectory*, *MBR*) como operaciones relacionadas con los índices espacio-temporales (*TimeSlice*, *TimeInterval*, *Knn*). Las consultas soportadas son las siguientes:

- *ObjectPosition*: calcular la localización de un objeto en un instante de tiempo dado
- *ObjectTrajectory*: se cansulta la trayectoria de un objeto completa o bien limitada por un intervalo de tiempo.
- *MBR*: esta operación calcula el mínimo rectángulo que envuelve la trayectoria seguida por un objeto entre dos instantes de tiempo.
- *TimeSlice*: dada una región espacial y un instante de tiempo, se calculan los objetos que están dentro de esa región.
- *TimeInterval*: a partir de un intervalo de tiempo y una región espacial se computan los objetos que han estado
- *Knn*: dada una localización en el espacio, un instante de tiempo y un parámetro k, se obtienen los k-ésimos objetos más cercanos a esa localización del espacio en el instante de tiempo dado.

Es importante notar que muchas veces la información de que contienen las colecciones de datos es errónea o debido a circunstancias externas existen períodos se falta de información. Aunque los errores pueden ser fáciles de detectar y se pueden eliminar, la falta de información es algo que la estructura debe tratar. Por esta razón nuestras estructuras usan diferentes estrategias dependiendo del tipo de log para manejar este tipo de incertidumbre. En el caso de ScdcCT y GraCT se añaden nuevos códigos que representan períodos sin información y la duración de los mismos. Sin embargo en ContaCT y RCT se añade una marca individual para cada instante de tiempo, que señala si la localización del objeto en ese instante de tiempo es conocida o no.

Para evaluar las ocho posibles estructuras, se han usado cuatro colecciones de datos y sobre ellos se han construido los diferentes índices con distintos parámetros. En esas comparaciones pudimos observar que aquellas estructuras cuyas snapshots se basan en R-trees obtienen una compresión similar a las basadas en k^2-trees, pero son más eficientes a la hora de resolver las consultas. En general, si comparamos el espacio de las distintas estructuras de log, aquellas configuraciones que usan GraCT obtienen le mejor compresión y en el otro extremo se encuentra ContaCT. ScdcCT suele estar cerca de GraCT cuando la distancia entre snapshots es corta, pero a medida que ésta crece, ScdcCT comprime peor. RCT mejora un poco el espacio de ContaCT, pero no llega a acercarse. En cuanto al tiempo, ContaCT muestra su superioridad y GraCT se ve penalizado por la necesidad de descomprimir ciertas partes del log. Para probar la escalabilidad de estas estructuras a medida que los datos van creciendo, evaluamos su funcionamiento con distintos tamaños de una mismo colección de datos. En esa experimentación pudimos observar que aquellas estructuras formados por los logs RCT y ContaCT son las más rápidas y, como esperábamos, el tiempo requerido para calcular la posición de los objetos se mantiene constante a medida que la colección de datos va creciendo. En otro experimento de la evaluación experimental se comparan las estructuras contra un índice espacio temporal, configurado para residir en memoria. Los resultados demuestran la capacidad de estas estructuras para resolver las consultas más rápido que dicho índice, usando mucha menos cantidad de espacio.

B.3 Conclusiones

Como se puede observar, en esta tesis se han diseñado las primeras estructuras compactas para la representación de trayectorias en espacios abiertos, es decir, movimientos sin ningún tipo de restricción. Añadiendo más valor al estado de arte actual de la representación de trayectorias de objetos móviles.

Todas estas estructuras se enfocan en mantener la estructura comprimida en memoria de tal forma que se pueda acceder a súa información sin la necesidad de descomprimirla. Para ello, cada una cuenta con dos componentes: snapshots y logs. En cuanto a las snapshots, está claramente demostrado que aquellas basadas en R-tree consiguen una mejora sustancial en tiempo y ocupan lo mismo que las basadas en k^2-tree.

En cuanto a las técnicas usadas para representar el log, cada una de ellas goza de unas ventajas y desventajas, que permite al usuario decantarse por una o por otra en función del dominio y ámbito a tratar. Por ejemplo, según nuestra evaluación experimental, si lo que se prima es la compresión y la colección es altamente repetitiva, GraCT sería la mejor opción. En caso de tener que tratar con movimientos cortos no repetitivos y prime la compresión, ScdcCT podría ser una buena opción. Por otro lado, si lo que se busca es la velocidad para responder a las consultas ContaCT y RCT serían las dos mejores opciones. Además, para cada una de las estructuras se realizó un estudio de escalabilidad, que demostró la buen hacer de todas ellas ante colecciones de datos grandes.

En la comparación de las estructuras con el índice espacio temporal clásico, se puede observar que las estructuras propuestas en esta tesis pueden obtener mucho mejor rendimiento ocupando mucho menos espacio. Mostrando así que las estructuras de datos compactas aportan un enfoque tan válido, como el de los índices espacio-temporales, al mundo de la representación de objetos móviles.

B.4 Trabajo futuro

En esta sección proponemos varias consideraciones que pueden ser interesantes para un futuro de cara a mejorar la aplicabilidad y el rendimiento de nuestras contribuciones. Entre ellas podemos destacar las siguientes líneas de investigación:

- Aumentar las consultas soportadas. Aunque las consultas que las estructuras propuestas pueden resolver son las más comunes de la representación de trayectorias, hay otras consultas que podrían ser de interés. Por ejemplo, detectar movimientos de objetos que viajan juntos, buscar patrones representativos de trayectorias o las trayectorias más parecidas entre si, como estas tres operaciones tratan de buscar similitudes entre movimientos, se podrían implementar de forma eficiente a partir de varias consultas de MBR.

- Mejorar la compresión de RCT. Como se puede observar en la evaluación experimental, RCT reduce el tamaño de la estructura si la comparamos con el ContaCT, pero esta mejora es mínima. La causa principal de este problema es que la referencia no es lo suficientemente buena para generar una buena compresión. Una propuesta interesante para mejorar dicha referencia es construirla a partir de una gramática, lo que añadiría a la referencia las partes más habituales entre las trayectorias.

- Precisión total. Las estructuras presentadas se basan en un modelo ráster, es decir, el espacio es dividido en celdas de un tamaño fijo. A pesar de que ese tamaño de celda se puede ajustar en función de la aplicación, se puede producir una falta de precisión no deseada. Una buena propuesta para estas estructuras sería utilizarlas como una estructura en memoria principal, que permita resolver las consultas con un cierto error de precisión y que esos resultados se refinen mediante el uso de índices basados en disco que indexen la información con la precisión total.

- Aplicar estas estructuras a otros ámbitos. Aunque estas estructuras representan la trayectoria de objetos, esto no es más que una serie temporal en la que los valores varían en función de donde está el objeto. Una opción interesante, sería evaluar estas estructuras en otros ámbitos relacionados con las series temporales, como puede ser el mercado bursátil.

Bibliography

[Abr63] N. Abramson. *Information Theory and Coding.* McGraw-Hill, 1963.

[AF00] Eugene Ageenko and Pasi Fränti. Lossless compression of large binary images in digital spatial libraries. *Computers & Graphics*, 24(1):91–98, 2000.

[BDM$^+$05] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BFCP$^+$05] Michael A Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[BFNP05] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Efficiently decodable and searchable natural language adaptive compression. page 234, 2005.

[BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[BFNP10] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Dynamic lightweight text compression. *ACM Transactions on Information Systems*, 28(3):article 10, 2010.

[BGG$^+$15] Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Proc. Data Compression Conference (DCC)*, pages 83–92, 2015.

[BHMR07] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th*

Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 680–689, 2007.

[BINP03] N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá. An efficient compression code for text databases. In *Proc. 25th European Conference on Information Retrieval Research (ECIR)*, pages 468–481, 2003.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

[BLN13] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013.

[BLN14] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

[BLNS13] Nieves R Brisaboa, Miguel R Luaces, Gonzalo Navarro, and Diego Seco. Space-efficient representations of rectangle datasets supporting orthogonal range querying. *Information Systems*, 38(5):635–655, 2013.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BMNS08] Viorica Botea, Daniel Mallett, Mario A. Nascimento, and Jörg Sander. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[BV04] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. 13th International Conference on World Wide Web (WWW)*, pages 595–602, 2004.

[CEP03] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with SETI. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2003.

[CFWF19] Yachang Cheng, Wolfgang Fiedler, Martin Wikelski, and Andrea Flack. "closer-to-home" strategy benefits juvenile survival in a long-distance migratory bird. *Ecology and evolution*, 9(16):8945–8952, 2019.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Ontario, Canadá, 1996.

[CLL05] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.

[CM10] J Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1):1, 2010.

[CMWM10] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Proc. 26th IEEE International Conference on Data Engineering (ICDE)*, pages 109–120, 2010.

[dMNZBY00] E. Silva de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[DP73] D. H. Douglas and T. K. Peuker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[DRR06] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *Proc. 5th International Workshop on Experimetal Algorithms (WEA)*, pages 134–145, 2006.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[Fan71] Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

[FFW16] A Flack, W Fiedler, and M Wikelski. Data from: Wind estimation based on thermal soaring of birds, 2016.

[FH11] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[FM08] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–184, 2008.

[FN17] H. Ferrada and G. Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.

[GB11] Szymon Grabowski and Wojciech Bieniecki. Merging adjacency lists for efficient Web graph compression. In *Man-Machine Interactions 2*, pages 385–392. Springer, 2011.

[GBMP14] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[GBT84] Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. Scaling and related techniques for geometry problems. In *Pro. 16th ACM symposium on Theory of computing*, pages 135–143. ACM, 1984.

[GGG$^+$07] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. 15th Annual European Symposium on Algorithms (ESA)*, pages 371–382, 2007.

[GLW08] Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Movement patterns in spatio-temporal data. *Encyclopedia of GIS*, 726:732, 2008.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GNR$^+$05] G. Gutiérrez, G. Navarro, A. Rodríguez, A. González, and J. Orellana. A spatio-temporal access method based on snapshots and events. In *Proc. 13th ACM International Symposium on Advances in Geographic Information Systems (GIS)*, pages 115–124, 2005.

[GRR04] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[GRRR04] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 159–172, 2004.

[GRRR06] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[HAE$^+$15] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, volume 9473. International Society for Optics and Photonics, 2015.

[HMR07] M. He, J. Ian Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 509–520, 2007.

[HN14] C. Hernández and G. Navarro. Compressed representations for Web and social graphs. *Knowledge and Information Systems*, 40(2):279–313, 2014.

[Huf52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proc. I.R.E. (Institue of Radio Engineers Inc.)*, volume 40, pages 1098–1101, 1952.

[HWZ$^+$14] S. Huang, B. Wang, J. Zhu, G. Wang, and G. Yu. R-HBase: A multi-dimensional indexing framework for cloud computing environment. In *Proc. IEEE International Conference on Data Mining Workshop*, pages 569–574, 2014.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JSS07] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[KCHP01] E. Keogh, S. Chu, D. Hart, and M. Pazzani. An online algorithm for segmenting time series. In *Proc. 2001 IEEE International Conference on Data Mining (ICDM)*, pages 289–296, 2001.

[KPZ10] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 201–206, 2010.

[KY00] John C Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[KYNC00] John C Kieffer, En-Hui Yang, Gregory J Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.

[LM00] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722 –1732, 2000.

[LMZ+17] Xuelian Lin, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. One-pass error bounded trajectory simplification. *Proceedings of the VLDB Endowment*, 10(7):841–852, March 2017.

[LPMW16] Kewen Liao, Matthias Petri, Alistair Moffat, and Anthony Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proceedings of the 25th International Conference on World Wide Web*, pages 807–816. International World Wide Web Conferences Steering Committee, 2016.

[LS89] David Lomet and Betty Salzberg. *Access methods for multiversion data*, volume 18. ACM, 1989.

[LY08] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4(3):28:1–28:13, 2008.

[LZ86] Abraham Lempel and Jacob Ziv. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, 32(1):2–8, 1986.

[MdB04] Nirvana Meratnia and Rolf A. de By. Spatiotemporal compression techniques for moving point objects. In *Proc. 9th International Conference on Extending Database Technology, (EDBT)*, pages 765–782, 2004.

[MNW95] A. Moffat, R. Neal, and I.H. Witten. Arithmetic coding revisited. *Proc. Data Compression Conference (DCC)*, 16(3):256–294, 1995.

[MOH+14] Jonathan Muckell, Paul W. Olsen, Jeong-Hyon Hwang, Catherine T. Lawson, and S. S. Ravi. Compression of trajectory data: A comprehensive evaluation and new approach. *GeoInformatica*, 18(3):435–460, 2014.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MR04] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. 31th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1006–1015, 2004.

[MRR01] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[MRRR12] J Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[Mun96] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[MYQZ09] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query processing of massive trajectory data based on mapreduce. In *Proc. 1st International Workshop on Cloud Data Management (CloudDB)*, pages 9–16, 2009.

[Nav16] Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 570 pages.

[NDAEA13] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. MD-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, 31(2):289–319, 2013.

[NH15] A. Nibali and Z. He. Trajic: An effective compression system for trajectory data. *IEEE Transactions on Knowledge and Data Engineering*, 27(11):3138–3151, 2015.

[NM96] Craig G Nevill-Manning. *Inferring sequential structure.* PhD thesis, University of Waikato, 1996.

[NMW97a] Craig G Nevill-Manning and Ian H Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2_and_3):103–116, 1997.

[NMW97b] Craig G Nevill-Manning and Ian H Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings - Practical on-line search algorithms for text and biological sequences.* Cambridge University Press, 2002.

[NR07] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering*, 19(5):663–678, 2007.

[NS98] Mario A. Nascimento and Jefferson R. O. Silva. Towards historical R-trees. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 235–240, 1998.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc.of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[Pag99] R. Pagh. Low redundancy in static dictionaries with O(1) worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 595–604, 1999.

[PJT00] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. 26th International Conference on Very Large Data Bases (VLDB)*, pages 395–406, 2000.

[PPS06] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *Proc. 18th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 275–284, 2006.

[PW96] Renato Pajarola and Peter Widmayer. Spatial indexing into compressed raster images: how to answer range queries without decompression. In *Proc. International Workshop on Multimedia Database Management Systems (MDBMS)*, pages 94–100, 1996.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:370–423,623–656, 1948.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.

[SRL09] Falko Schmid, Kai-Florian Richter, and Patrick Laube. Semantic trajectory compression. In *Proc. 11th International Symposium on Spatial and Temporal Databases (SSTD)*, pages 411–416, 2009.

[TCS$^+$06] Goce Trajcevski, Hu Cao, Peter Scheuermann, Ouri Wolfson, and Dennis Vaccaro. On-line data reduction and the quality of history in moving objects databases. In *Proc. 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 19–26, 2006.

[TLCF16] Na Ta, Guoliang Li, Bole Chen, and Jianhua Feng. Semantic-aware trajectory compression with urban road network. In *Proc. 17th International Conference (WAIM), Part I*, pages 124–136, 2016.

[TLN12] Haoyu Tan, Wuman Luo, and Lionel M. Ni. CloST: A hadoop-based storage system for big spatio-temporal data analytics. In *Proc. 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2139–2143, 2012.

[Tob70] Waldo R Tobler. A computer movie simulating urban growth in the detroit region. *Economic geography*, 46(sup1):234–240, 1970.

[TP01a] Yufei Tao and Dimitris Papadias. Efficient historical R-trees. In *Proc. International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 223–232, 2001.

[TP01b] Yufei Tao and Dimitris Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. 27th International Conference on Very Large Data Bases (VLDB)*, pages 431–440, 2001.

[VTS98] Michalis Vazirgiannis, Yannis Theodoridis, and Timos K. Sellis. Spatio-temporal composition and indexing for large multimedia applications. *ACM Multimedia Systems Journal*, 6(4):284–298, 1998.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WNC87] I. Witten, R. Neal, and J. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30:520–541, 1987.

[Wor05] Michael F. Worboys. Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science*, 19(1):1–28, 2005.

[WZ99] H. E. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, mar 1999.

[WZX+14] Haozhou Wang, Kai Zheng, Jiajie Xu, Bolong Zheng, Xiaofang Zhou, and Shazia Sadiq. SharkDB: An in-memory column-oriented trajectory storage. In *Proc. 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, pages 1409–1418, 2014.

[WZXM08] Longhao Wang, Yu Zheng, Xing Xie, and Wei-Ying Ma. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. In *Proc. International Conference on Mobile Data Management (MDM)*, pages 1–8, 2008.

[XHL90] Xiaomei Xu, Jiawei Han, and Wei Lu. RT-tree: An improved R-tree index structure for spatiotemporal databases. In *Proc. 4th International Symposium on Spatial Data Handling*, volume 2, pages 1040–1049, 1990.

[YHC18] Shengxun Yang, Zhen He, and Yi-Ping Phoebe Chen. GCOTraj: A storage approach for historical trajectory data sets using grid cells ordering. *Information Sciences*, 459:1 – 19, 2018.

[YK00] En-Hui Yang and John C Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. *IEEE Transactions on Information Theory*, 46(3):755–777, 2000.

[ZJM+17] Zhigang Zhang, Cheqing Jin, Jiali Mao, Xiaolin Yang, and Aoying Zhou. TrajSpark: A scalable and efficient in-memory management system for big trajectory data. In *Proc. 1st International Joint Conference APWeb-WAIM, Part I*, pages 11–26, 2017.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

[ZZ11] Yu Zheng and Xiaofang Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.

[ZZS+05] Panfeng Zhou, Donghui Zhang, Betty Salzberg, Gene Cooperman, and George Kollios. Close pair queries in moving object databases. In *Proc. 13th Annual ACM International Workshop on Geographic Information Systems (GIS)*, pages 2–11, 2005.