COMPACT DATA STRUCTURES AND QUERY PROCESSING FOR TEMPORAL GRAPHS

por

Diego Felipe Caro Alarcón

Profesoras Guía:
M. Andrea Rodríguez Tastets
Nieves Rodríguez Brisaboa

Tesis presentada
para la obtención del título de

DOCTOR EN CIENCIAS DE LA COMPUTACIÓN

Departamento de Ingeniería Informática y Ciencias de la Computación

de la

UNIVERSIDAD DE CONCEPCIÓN

Concepción, Chile
Junio, 2015

A mi amigo Moncho.

(=^·^=)

Tengo tiempo para saber
si lo que sueño concluye en algo.
No te apures ya más, loco
porque es entonces cuando las horas
bajan, el día es vidrio sin sol
bajan, la noche te oculta la voz
Y además vos querés sol
Despacio también podés hallar la luna.
Viejo roble del camino,
tus hojas siempre se agitan algo
Nena, nena, que bien te ves
Cuando en tus ojos no importa si las horas
bajan, el día se sienta a morir
bajan, la noche se nubla sin fin
Y además vos sos el sol
Despacio también podés ser la luna.

L.A. Spinetta

Agradecimientos

En primer lugar quisiera agradecer a mis tutoras. A Andrea, gracias por la paciencia, la confianza, y por todas las horas invertidas en mi formación. A Nieves, gracias por toda la ayuda la distancia y mientras estuve en Coruña.

Gracias también a mis compañeros de posgrado Pepe, Naty y Erick por todas esas horas de discusión y ayuda desinteresada. Gracias también a Diego por todas sus sugerencias y por alentarme a probar nuevas ideas.

Gracias a todos en el LBD, en especial a Fari, Sandra, Leti y Guillermo por ayudarme con la experimentación de esta tesis, y por hacer mi estadía más grata mientras estuve en Coruña.

Y por sobre todo, gracias infinitas a Coty-co, gracias por estar siempre ahi!.

Abstract

Temporal graphs represent vertices and binary relations that change along time. A temporal graph could be represented as several static graphs (or snapshots), one per each time point in the lifetime of the graph. The main problem of this representation is the space usage when edges remain unchanged for long periods of time, as consecutive snapshots tend to be very similar between them. Consequently, most of the space correspond to a replication of previous snapshots.

In this thesis, we explore alternative representations of temporal graphs that are efficient in space and in time to answer adjacency operations. On one hand, a representation based on a *log of changes* stores the time instant when edges appear or disappear. A *multidimensional binary matrix*, on the other hand, stores data of edges and their valid time intervals as cells in a 4D matrix, which reduces space while maintaining a good retrieval time.

As a baseline of comparison, we present two first strategies: the Time-interval Log per Edge (EdgeLog) and the Adjacency Log of Events (EveLog), both using compression techniques over an inverted index that represent the logs. We use a Compact Suffix Array to represent temporal graphs as a sequence of 4-tuples, where adjacency operations are answered as a pattern matching problem.

We introduce two new strategies to represent temporal graphs using compact data structures. Compact Adjacency Sequence (CAS) represents changes on adjacent vertices as a sequence stored in a Wavelet Tree, and the Compact Events ordered by Time (CET) represents the edges that change in each time instant using an Interleaved Wavelet Tree, a new compact data structure specifically designed in this work that is able to represent a sequence of multidimensional symbols (that is, tuples of symbols encoded together).

We finally propose to represent temporal graphs as cells in a 4D binary matrix: two dimensions to represent extreme vertices of an edge and two dimensions to represent the temporal interval when the edge exists. This strategy generalizes the idea of the adjacency matrix for storing static graphs. The proposed structure called Compressed k^d-tree (ck^d-tree) is capable to deal with unclustered data with a good use of space. The ck^d-tree uses asymptotically the same space than the information-theoretical lower bound for storing cells in a 4D binary matrix, without considering any regularity. Techniques that group leaves into buckets and compress nodes with few children show to improve the performance in time of the ck^d-tree.

We experimentally evaluate all the structures and compare them with previous alternatives in the state-of-the-art based on snapshots and logs, showing that our proposals can represent large temporal graphs making efficient use of space, while keeping good time performance for a wide range of useful queries. We conclude that the use of compact data structures open the possibility for the design of interesting representations of temporal graphs that fit the needs of different application domains.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Temporal graphs model real networks that exhibit a dynamic behavior where the interactions between elements of the network change over time. For example, consider an online social network where friends are added or removed along time, or a network of mobile communications where connections represent calls between mobile phones.

The use of temporal graphs can support the discovery of special events or interesting time-varying patterns, and getting historical information within specific time intervals [CBL08]. For example, a system may need to analyze changes on the topology of communication networks to prevent bottlenecks [BCGJ11], to study the evolution of transportation networks to provide better public transportation, to use the evolution of the Web graphs over a period of time to reorder the results of a query [AZAL04], to analyze the evolution of social communities to detect terrorism networks [TL10], or to use changes on object locations to link visual objects in video data [LOH05]. It can also help to develop new drugs for medical treatment [Hol08], help to know how the information is diffused in networks [GRLK12], or to support social network analysis for business applications [BCGJ11]. The Linked Data initiative has also made available huge amount of data in the form of RDF triples, which are in essence edges of a RDF graph where vertices are RDF resources. Example of such data can be data about people and their time-sensitive attributes such as their position in an organization, address, and marriage state. These examples all share the interest for querying not only the current but also historical state of the network.

Taking into account the historical dynamism of temporal graphs also allows us to exploit information about temporal correlations and causality, which would be unfeasible through a static (or classical) network analysis [NTM+13, HS12]. Classical measures over static graphs (e.g., centrality, betweenness, and so on) use the assumption that vertices are always connected, which is a naive assumption for real networks. As a consequence, these measures overestimate links availability, which could lead to wrong conclusions about the network.

A temporal graph could be represented as several static graphs (or *snapshots*), one per each time point in the lifetime of the graph. The main problem with this snapshot-based representation is that the space used is several times larger than the space used to store active edges in the temporal graph. Indeed, depending on the time granularity (i.e., seconds, minutes, and so on), edges may remain unchanged a long time interval, generating duplicated information with a consequently increase of space usage. Another alternative is to represent temporal graphs as a *temporal log* composed by the history of activations/deactivations of edges along time. Although this alternative is less space consuming, it requires a sequential traversal over the history of changes to obtain the activation state of an edge.

The work in this thesis explores different alternatives of compressed data structures for temporal graphs based on logs of historical changes and multidimensional representations. We start by proposing a compressed data structure for current temporal graph representations based on storing a history of changes by edge [BXFJ03], which will act as a baseline representation. Then we modify a compressed full text index that recovers the state of edges

by doing a pattern matching operation over the sequence. We design a sequence-based encoding that does not require sequential scanning of the temporal history to obtain the state of edges. We also propose a novel representation that considers temporal graphs as whole space-time data in a multidimensional environment. This multidimensional representation does not require to traverse the temporal logs to recover the current state of edges.

1.1 Motivation

Traditionally, compression has been seen as the task to reduce the space of data. The advantages are clear, less time to transfer the data and more space available. However, the reduction of the space is accompanied with a reduction in accessibility; once the data is compressed, you have to process it to recover the original data. Sometimes this can be even worse, as the retrieval of a small portion of the original data can take as long as recovering all the original data.

With the increasing need of storage, management, and retrieval large amounts of data, the compression task has been also extended to space-efficient data structures. In this context, the main objective of *compressed data structures* is to provide a small space and a collection of operations over the original data, by only processing a negligible portion of its compressed version.

As Gog claims [Gog11], the economical advantage of compressed data structures arises in elastic computing services, where one pays by the resources used per hour. For instance, an application A using m_A MB of memory and solving a task in t_A hours is more economical than an application B that uses $2m_A$ MB and does not complete the task in less than $t_A/2$ hours. This is because the cost of memory usually increases linearly (See Table 1.1).

Instance Name	Memory (GB)	Price per hour (USD)
r3.large	15	\$0.175
r3.2xlarge	61	\$0.700
r3.4xlarge	122	\$1.400
r3.8xlarge	244	\$2.800

Table 1.1: Pricing of Amazon Elastic Cloud Computing (EC2) service in May 2015. Source [Ama15].

The seminal work by Jacobson [Jac89b, Jac89a] introduces the idea of succinct data structures, which main goal is to reduce the space used by static data structures, retaining fast access time without decompressing the representation[1]. He provides, among other contributions, an asymptotically optimal space efficient representation of a static binary tree of n nodes in $2n + o(n)$ bits, which is a fraction of the classical pointer-based representation that traditionally requires $2n \log n$ bits. This small encoding allows the manipulation of large trees in much less memory than if we were using the 64 bits per pointer that nowadays computers provide [SN09].

Typically, data structures act as indexes over data. An alternative to obtain compressed data structures is to represent the index by using a succinct data structure. This was one of the first strategies followed by Grossi and Vitter [GV00, GV05] to reduce the space of the

[1] A static data structure does not provide methods for updating the data structure.

Suffix Array [MM93]. They were able to reduce the suffix array of a text of length n from $O(n \log n)$ bits to $O(n)$ bits. However, the text was still uncompressed.

More recent research in this area has developed self-indexed compressed data structures, where the data and the index are compressed. The Compressed Suffix Array of Sadakane [Sad00, Sad03] is able to compress both, the text and its suffix array. Contrary to the Suffix Array, the original text is no longer required to answer operations, as it is available in a compressed version. The main advantages of this strategy is that the total space of the *self-index* depends on how compressible the data is.

Compressed data structures have been developed for full text indexes [NM07, FGNV09], large graphs [BLN09, BdBN12, CN10a, HN13, BLN14], geographic data [BLNS13], binary relations [BCN13], among others. However, despite the increasing interest in compact data structures[2], no much work has been done so far about compact data structures for temporal graphs.

The use of large temporal graphs has been already detected as a challenging problem. Gudivada, Baeza-Yates and Raghavan [GBYR15] have already noticed that the processing of temporal graphs and dynamic networks impose new challenges to the MapReduce framework, which is traditionally used in commercial applications of cloud computing. In this sense, the use of compressed data structures could take a fundamental role in big data problems that are not amenable to environments like MapReduce.

1.1.1 Goal and hypothesis

The main goal of this thesis is to design new compressed data structures for temporal graphs. We focus here on static representations of temporal graphs with directed edges between vertices. We will consider the answering of adjacency operations for retrieving the activation state of edges and direct/reverse neighbors of vertices over a time instant and a time interval. We will also obtain the set of active edges in a time instant, as well as the set of edges that were activated or deactivated in a time instant or a time interval.

Major research questions that drive the development of this thesis are:

- What are the current representations of temporal graphs? How can they be integrated into compressed indexes?
- How can the adjacency operations on temporal graphs be solved without making a sequential or binary search over a log of activation or deactivation of edges?
- Is it possible to index the data of a temporal graph by time and vertices?
- How can we guarantee the same access time to direct and reverse neighbors in a temporal graph?
- What are the properties of temporal graphs that affect the design of compact data structures? How do they affect the compact data structures in terms of space consumption?
- Is it possible to use general-purpose multidimensional data structures for temporal graphs?

[2]We use the term compact data structure as a synonym of compressed data structure. The latter term was born with the idea of compressing the suffix array.

The answers to these questions yield to the definition of the following data structures: EdgeLog, EveLog, TG-CSA, CAS, CET, ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree. As a consequence, the hypothesis of this work is:

Compact data structures for temporal graphs based on logs of changes and multidimensional representation use less space and similar access time than structures based on snapshot representations.

This hypothesis is supported by the evaluation of all compressed data structures in Chapter 7.

1.1.2 Major results

In this thesis seven compressed compact data structures was developed. The EdgeLog and the EveLog are *temporal logs* that are compressed by using inverted indexes. The adjacency operations need to decompress and traverse the log to recover the state of edges. These structures act as our compressed baselines due their simple compression schema.

The TG-CSA is a modification of the Compressed Suffix Array [Sad02, Sad03], which encodes temporal graphs as a sequence formed by the concatenation of tuples indicating the time interval when an edge was active. The operations are answered following the binary search used for text pattern matching.

The CAS and CET are sequence-based representations of the temporal log. The main advantage is that they do not require to traverse the log for answering adjacency operations. The state of edges is recovered by counting how many times each edge appears in the subsequence related to the vertex and the query time. If it appears an odd number of times, it means that the last state of the edge is active, otherwise it is inactive. In CAS, the sequence is indexed by the source vertex and then by the temporal log related to the source vertex. The sequence is encoded in a Wavelet Tree, a compact data structure capable of returning the frequency of the symbols in logarithmic time.

The main issue with CAS is that the retrieval of reverse neighbors is slower than the retrieval of direct neighbors. We solved this issue in CET, where the temporal log is primary indexed by time, and the sequence is composed by the edges that changed its activation state at each time instant. To answer adjacency operations we propose an extension of the Wavelet Tree called Interleaved Wavelet Tree, which represents the edges in the sequence as 2D symbols. Adjacency operations are also answered by recovering the frequency of symbols, but fixing the first or second component (i.e., the source or target vertex of each edge in the sequence).

With ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree we represent a temporal graph as cells in a 4D binary matrix. As in TG-CSA, the dimensions represent the time interval when an edge (source and target vertices) was active. The adjacency operations are answered trough orthogonal range searches. The main advantage of ck$^{\mathsf{d}}$-tree is that it guarantees an space close to the information-theoretic lower bound for representing a 4D binary matrix with m active cells.

All the data structures developed in this thesis were evaluated against synthetic and real temporal graphs. They are freely available as prototypes in C++[3]. All the implementations were conducted using open source libraries of succinct and compressed data structures:

[3] `http://github.com/diegocaro/temporalgraphs`

the Compressed Data Structure Library (libcds) [CN08] and the Succinct Data Structure Library (sdsl-lite) [GBMP14a].

1.1.3 Organization of the remaining chapters

The organization of the thesis is as follows.

Chapter 2 presents the fundamental concepts related to succinct and compressed data structures that are relevant for the development of the thesis. It also includes important compressed self-indexes that we modified for compressed representations of temporal graphs.

Chapter 3 introduces a formal definition of temporal graphs, and a set of adjacency operations that we consider for the development of the data structures. It also introduces a classification of temporal graphs regarding the main characteristics of activation/deactivations of edges along time. Finally, it reviews some general schemas related with the design of data structures for managing temporal data.

Chapter 4 presents three basic compressed data structures based on well-known compressed indexes. Two of them are related to inverted-index technology that represents temporal graphs as a log of changes. The third structure is a modification of the Compressed Suffix Array that encodes the temporal graph as a sequence of 4-tuples. In this structure, adjacency operations are answered like a pattern-matching problem.

Chapter 5 introduces two compressed data structures based on the Wavelet Tree that do not require a sequential search over the temporal log to check the activation state of edges. It also presents an extension of the Wavelet Tree to deal with sequences of multidimensional symbols.

Chapter 6 reviews previous work that encodes temporal graphs into binary matrices. It then proposes a new representation that encodes temporal graphs as cells in a 4D binary matrix. It also introduces a novel compressed representation, which ensures a space close to the minimum required in multidimensional binary matrices that do not share clustering properties.

Chapter 7 gives an extensive experimental evaluation of space and time for all the structures presented in the thesis using real and synthetic data sets.

Chapter 8 finishes the document with conclusions and future research directions.

Chapter 2

Previous concepts

The research in compression has been an active research area for many years, but it has recently experienced a great development due to the increasing need of storage, management, and retrieval of the large amounts of available data. In this context, not only more efficient compression techniques have been developed, but also these techniques allow searching the compressed data, without decompression, in a more efficient way than if the data was in its uncompressed format. Notice, therefore, that compression does not only save storage space but, more importantly, it saves disk access time because more data could fit in main memory, saving transmission and processing time.

Recent research in this area has developed self-indexed compact data structures such as the Compressed Suffix Array [Sad00], the Wavelet Tree [GGV03], and the k^2-tree [BLN14, BLN09]. These data structures not only represent data in a compressed form, as the classical compression techniques, but they also provide indexing capabilities speeding up the search into the compressed data.

In what follows, we introduce some concepts and a brief description of the state-of-the-art compression techniques and compressed data structures used in this work.

2.1 Information-theoretic lower bound and entropy

Two import concepts to measure space efficiency of data structures are the information-theoretic lower bound and the entropy. The information-theoretic lower bound for storing an object of a universe of size u is $\log u$ bits[1], which corresponds to the minimum number of bits to distinguish two objects of that universe [SN09]. For example, the minimum space necessary for distinguishing two symbols (i.e., two characters) of an alphabet of 256 symbols is $\log 256 = 8$ bits. Similarly, a string of length n over an alphabet of size σ will require $n \log \sigma$ bits (as there are σ^n different strings). Jacobson [Jac89a] uses this measure to provide an asymptotically optimal encoding for binary trees: because there are $\frac{1}{n+1}\binom{2n}{n}$ different binary trees of n nodes, the minimum space to differentiate each of these trees is $2n + o(n)$ bits.

Notice that the information-theoretic lower bound does not consider any regularity on the elements of the universe (i.e., it assumes a uniform distribution over the objects in the universe [SB15]). However, this assumption is very unlikely in many domains such as natural languages, where certain symbols are more frequent than others (vocals, for example).

The entropy introduced by Shannon [Sha48] allows us to measure the minimal amount of bits required to transmit a string taking into account the probability distribution of the symbols in the message. Let $\mathcal{X}$ be a random discrete variable over n outcomes (i.e., the size of the alphabet) and p_i the probability of occurrence of the i-th outcome, the entropy of $\mathcal{X}$ is defined as:

$$H(\mathcal{X}) = -\sum_{i=1}^{n} p_i \log p_i.$$

[1] In this thesis we will consider $\log x$ as $\log_2 x$, unless we explicitly use another base.

This measure has many important properties. It is the average number of bits required to transmit an outcome following the probability distribution.

A more practical measure to obtain the entropy of a sequence (or a text) S of length n over an alphabet Σ is the zero-order empirical entropy[2], defined as

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c},$$

where n_c is the frequency of the symbol c in S. The expression $nH_0(S)$ defines the lower bound space of an optimal encoder that requires $\log \frac{n}{n_c}$ bits to encode a symbol c. In the worst case, as in the Shannon entropy, if all the symbols in S have the same frequency, we have $H_0(S) = \log |\Sigma|$. Thus, $nH_0(S) \leq n \log |\Sigma|$.

The zero-order entropy is the maximum compression that can be achieved, considering that optimal encoders, such as Huffman [Huf52, CT12], always assign the same codeword to each symbol [Man01]. We can achieve better compression ratios with codewords that consider the context of size k where a symbol occurs. This notion is formalized as high-order empirical entropies.

Let w be a sequence of size k over the alphabet Σ, and w_S a sequence composed by the concatenation of the single symbols followed by an occurrence of w in S. For example, for $S = abaabbabc$, if $w = ab$, then $w_S = abc$. The k-order empirical entropy of S is defined as

$$H_k(S) = \sum_{w \in \Sigma^k} \frac{|w_S|}{n} H_0(w_S).$$

The value $nH_k(S)$ represents the lower bound space for representing S using codewords that depend on the preceding context of size k [Man01]. Notice that for any $k \geq 0$, we got $H_{k+1}(S) \leq H_k(S) \leq H_0(S)$.

The notion of entropy is well defined for strings (sequences) and has been used as a measure of the efficiency for many compressors (e.g., the Lempel-Ziv family [ZL77, ZL78] [KM97], the Repair grammar compressor [LM00] [NR08], among others). It has also been used for measuring the space efficiency of compressed data structures related to strings [GGV03, Sad03, SG06, NM07, FMMN07]. However, it is not well defined for other domain such as temporal graphs. Some attempts have been made for measuring the entropy on static graphs [Sim95, DM11, MD12], but as they are defined in terms of the distribution degree, it is unclear the relation of the entropy with the current compressed encodings for graphs (Section 2.4.1).

The information-theoretic lower bound has been extensively used as a measure of the space efficiency of succinct data structures on combinatorial object [RS06] such as trees [Jac89a, MR01, SN09], planar graphs [Jac89a, MR97, BAhM12], and binary relations [FGN14, BCN13]. In this thesis we use the information-theoretic space to give a lower bound for storing temporal graphs, without considering regularities of any kind. In some cases, we will reduce the temporal graph to a sequence. In that case, we will use the zero-order entropy to measure the space of the compressed data structures.

[2]Replacing the probabilities of the outcomes in $\mathcal{X}$ by the frequency of the symbols in S, $H(\mathcal{X}) = H_0(S)$.

2.2 Compression techniques

In this section we introduce some compression techniques used to represent text as sequences of integers in small space. We briefly review some of the state-of-the-art compressors for representing inverted indexes, and a fast and close to optimal encoder for integers.

Inverted indexes are a simple but powerful method for full text search in large collections of documents [WMB99, ZM06, MRS08]. It is composed of a dictionary of words (or lexicon) in the collection. For each word, there is a list of documents identifiers (docID) indicating the set of documents where the word appears. In the literature, this list is sometimes called inverted list or posting list. The inverted index is like the word index found in the back of a textbook, where for each term there is a list of pages where the term appears.

The docIDs can be stored as raw integers, but the space used by the index rapidly increases in large document collections. However, as posting list are sets, we can use a simple transformation to compress them as a list of d-gaps. The transformation works as follows. Let $\langle i_1, i_2, \ldots, i_n \rangle$ be an ordered sequence of integers of a universe u, with $i_j \leq i_{j+1}$. The sequence can be represented as a list of d-gaps of the form $\langle i_1, i_2 - i_1, i_3 - i_2, \ldots, i_n - i_{n-1} \rangle$. Then, the sequence can be compressed with a variable length encoder. The assumption behind this strategy is that many of the d-gaps will be small integers, so, many of these values will be encoded with few bits.

The information-theoretic lower bound for the d-gaps is $\log \binom{u}{n}$ bits, because there are $\binom{u}{n}$ ways to pickup n integers from the universe u. When $n \ll u$, this space is approximately $n \log \left(\frac{u}{n} + 1.44\right) = n \log \frac{u}{n} + O(n)$ bits [CM07]. Encoders such as Huffman [Huf52], variable byte encoding [MRS08], PForDelta [ZHNB06], the S9/S16 family [AM04, ZLS08], among others [WMB99, MRS08, CM07], are close to this bound.

Variable length encoders can be classified into bitwise and bytewise encoders [SWYZ02]. Bitwise encoders adapt the length of codewords to a *bit* level (e.g., Huffman codes). Bytewise encoders adapt the length of codewords in terms of *bytes*. This difference has an important impact on the time efficiency of the decompression stage. In what follows, we review some of the well-known bytewise integer compressors for inverted indexes.

2.2.1 Variable byte encoding

Variable byte encoding (vbyte) [MRS08, pp.96-98] is the simplest technique for storing integers. It is based on representing integers in a variable number of bytes in a stream. In each byte, the 7 least significant bits are used for encoding the integer, while the most significant bit is used as a flag to indicate if the codeword ends with this byte (1) or not (0). The 7 bits act as a payload of the integer, carrying 7 bits from the integer on each byte. An integer n will be encoded using $\lceil \log_{128} n \rceil$ bytes, and converted into digits in base 128. For example, the integer $n = 666 = 5 \times 128^1 + 26 \times 128^0$ is encoded using 2 bytes: **0**0000101 **1**0011010. The main advantage of variable byte encoding is its easy implementation and fast decoding. Its major drawback, on the other hand, is the minimum space required for storing small integers (i.e., one byte).

2.2.2 PForDelta

PForDelta [ZHNB06] encodes integers in groups of size multiple of 32 (usually 128). Each group is encoded using a fixed number of bits per integer. The number of bits is chosen by

obtaining the smallest b such that a great percent (90% for example) of the integers in the group are less than 2^b. Then each integer is moved to a slot of b bits. As some of them do not fit in b bits (as they are greater than 2^b), they are encoded as exceptions using 32 bits, and the slot is used as a pointer to the next exception.

Contrary to vbyte, PForDelta works by processing a segment (an array of bytes) instead of reading values from a stream of bytes. Each segment is composed by three sections: a header, $32k$ slots of b-bits to store the group of integers, and a zone of exceptions. The header contains the number of bits b chosen to codify the integers, the number of exceptions, and a pointer to the first exception. This schema, along with other optimizations, takes advantage of the CPU cache in nowadays CPUs, providing a decompression speed better than vbyte [ZHNB06, ZLS08].

Although PForDelta improves the space to store small integers (we can choose a small b value), the minimum number of integers to encode is 32, so, the minimum space used by a short list of integers is 4 bytes. This can be improved by using a bitwise encoder such as Golomb-Rice.

2.2.3 Golomb-Rice

Golomb-Rice encoding [WMB99, pp. 293-294] is based on the idea of representing an integer as a quotient and a remainder. This is an old idea initially proposed in Golomb codes [Gol66]. In a Golomb code, an integer n is encoded as a value $q = \lfloor n/b \rfloor$ in unary coding[3], and a remainder $r = n \mod b$. The value b is carefully chosen by taking into account the average of the values in the input. In Golomb-Rice codes [RP71], the parameter b is restricted to a power of two. For example, with $b = 2^2 = 4$, the integer $n = 10$ is represented with $q = \lfloor 10/4 \rfloor = 2$ and $r = 10 \mod 4 = 2$, and coded as 110 10. This power-of-two constraint allows a better performance on computing the division and the modulo operations in the decoding stage. The final space of each codeword will depend on the parameter b, which affects the average length of the quotient and the fixed length of the remainder $\lceil \log b \rceil$.

In contrast to vbyte and PForDelta, the main advantage of the Golomb-Rice is that it provides a space efficient method for encoding short lists of small integers.

2.2.4 End-Tagged Dense Codes

End-Tagged Dense Codes (ETDC) is a statistical byte-oriented compression technique [BFNP07]. It is based on vbyte encoding for representing texts as a sequence of words instead of a sequence of symbols. More frequent words in the text get shorter codes, and less frequent get larger codes. The codewords are assigned as with the vbyte encoding, but with a variable number of bits: $b - 1$ bits are used as a payload and one bit as a flag. With $b = 8$, ETDC generates codewords as with the vbyte (7 bits as payload). The first step is to build a vocabulary of words, ordered by their frequency in the text. Then the codewords are filed with the rank of each word in the vocabulary. For $b = 8$, the first 2^7 more frequent words use one byte. The next $(2^7)^2 = 2^{14}$ items in the vocabulary are encoded with two bytes, and so on. The decoding is as with the vbyte, but taking into account the conversion of the codeword to the corresponding word in the vocabulary. As with the vbyte, ETDC is very efficient for decoding. It also allows fast string search algorithms, such as

[3] A natural number n is represented using Unary coding by storing n zeros followed by a 1.

Boyer-Moore [BM77], because codewords can be easily skipped using the flag bit. In natural language text [BFNP07], there is a small space overhead ($\approx 2.5\%$) over optimal Huffman codes using bytewise codewords [dMNZBY00].

Discussion Although some bitwise encoders effectively guarantee a space close to the lower bound (such as Huffman codes), they require more time for decoding codewords than other less space efficient bytewise encoders. This is due to the bit shifting, masking, and other operations that are necessary for decoding each codeword [SWYZ02]. Bytewise compressors (such as variable byte encoding [MRS08]) are less space efficient, but are faster because less operations are required to decode each integer. We will use the bytewise PForDelta and Golomb-Rice encoding (in short lists) for compressing the baseline structures for temporal graphs based on inverted indexes (and d-gaps). We discard other bytewise encoders such as the S9/S16 family, as they have a worse decompression time [ZLS08]. When the list of integers is not ordered, we will use the ETDC, with a vocabulary composed by the most frequent integers. In this way, most frequent integers will use short codewords.

2.3 Succinct data structures

The seminal work by Jacobson [Jac89b, Jac89a] on succinct data structures starts with a short phrase: *small is beautiful*. Indeed, one of the main drawbacks of in-memory data structures is that the space used by pointers rapidly increases with the size of the input (Section 2.1). The main aim here is to represent data in a space close to the information-theoretic lower bound, retaining a fast access to them.

Since Jacobson's PhD Thesis, several works have been presented in this area, including trees [Jac89a, MR01, SN09], planar graphs [Jac89a, MR97, BAhM12], and binary relations [FGN14, BCN13]. In what follows we present some succinct data structures and operations that are the building blocks for creating more complex compressed data structures.

2.3.1 Rank and select over uncompressed bitmaps

One of the building blocks for succinct data structures are operations to count and access bit sequences (often referred as bitmaps or bit vectors). Given a bit sequence $B[1, n]$ of size n, it is possible to define three operations [Jac89a]:

- $\mathsf{rank}_b(B, i)$ returns the number of occurrences of b in $B[1, i]$. Consider the bitmap $B = 0010\ 0101$ of size $n = 8$. Then, $\mathsf{rank}_0(B, 4) = 3$, as there are three zeros up to position four. Similarly, $\mathsf{rank}_1(B, 8) = 3$, as the sequence only contains three ones.

- $\mathsf{select}_b(B, j)$ returns the position of the j-th bit b in B. For example, consider a bitmap $B = 1010\ 0111$, then $\mathsf{select}_0(B, 3) = 5$, because the third zero is found in the fifth position of B. The operation $\mathsf{select}_1(B, 1) = 1$, because the first one is in the first bit of B.

- $\mathsf{access}(B, i)$ recovers the i-th bit in B. Consider a bitmap $B = 0010\ 0101$, then $\mathsf{access}(B, 2) = 0$, because the second bit in B is set to zero.

When the subscript b is missing in rank and select, we will consider by default that the operation is set for counting 1 bits. Notice that $\mathsf{rank}_0(B,i)$ can be redefined as $n - \mathsf{rank}_1(B,i) - 1$.

The motivation to study rank and select operations over bits comes from the ability to represent ordered sets in a small space, which can be useful for representing other data structures. A naive approach to answer rank operations can be the sequential search that counts the b bit in the bit sequence until the i-th position, and analogously for select. Although this does not require extra space, it is very slow. Another strategy could be to precompute all the values of rank, by storing the position of each active bit but again, it is too heavy (in terms of space).

The first solution proposed by Jacobson [Jac89b, Jac89a] answers rank in constant time and uses $o(n)$ extra space over the bitmap, where n is the length of the bitmap. It is based on storing a two-level directory with precomputed values of rank every certain number of bits. The precomputed values are stored in a table. The first level directory divides the bitmap in blocks of size j bits. This creates a rank table with $\lfloor n/j \rfloor$ precomputed values. At the first level, each entry of the table needs $\log n$ bits (as n is the size of the bitmap and the maximum possible answer). In the secondary level, each of these blocks is treated as an independent bitmap $B'[1,j]$. These secondary level bitmaps are divided into blocks of size k, which generates a rank table with $\lfloor j/k \rfloor$ entries of size $\log j$. To find $\mathsf{rank}(B,i)$ we first obtain the precomputed rank value related to the block at position $p_1 = \lfloor i/j \rfloor$ in the first directory table. This gives us the rank value up to position $j \times p_1$. Then, we add the precomputed values in the secondary directory, in the block at position $p_2 = \lfloor (i \mod j)/k \rfloor$. In total, this gives us the rank up to position $j \times p_1 + k \times p_2$, except for those remaining bits in the block p_2 at the secondary directory. The remaining bits (at most k) can be retrieved directly from the p_2 block. With a careful selection of j and k, the extra space used by the precomputed rank tables are $O(n \log\log n / \log n) = o(n)$ bits. With this representation, select can be answered by doing a binary search in $O(\log n)$, by using the $\mathsf{rank}(B, i/2)$ values. Later works by Clark [Cla96] and Munro [Mun96] were able to answer the three operations (select inclusive) in constant time, also using $o(n)$ bits of extra space.

In next section we will review how to reduce the space of the bitmap representation, by taking into account the properties of the distributions of ones in the bitmap.

2.3.2 Compressed bitmap representations

In last section we revised uncompressed structures for maintaining rank and select operations. We say uncompressed because they are stored as arrays of integers that also require $o(n)$ bits to answer the operations. This can be improved by considering the distribution of ones in the bitmap. Indeed, the information-theoretic lower bound of a bitmap of size n with m ones is $\log \binom{n}{m} \leq n \log \frac{n}{m}$ bits, which is less than the n bits of the uncompressed version, unless $m \approx n/2$.

The theoretical work by Pagh [Pag99] explores this idea by dividing the bitmaps into blocks of fixed size b. Each block is represented by two numbers, a number m_i indicating the number of ones that contains the i-th block, followed by a number o_i in $\{1, ..., \binom{b}{m_i}\}$ that indicates which of the $\binom{b}{m_i}$ bitmaps with m_i ones is the i-th block. The proposal also includes a compression schema based on clustering adjacent blocks into intervals of variable length. Although the work by Pagh effectively reduces the space, it does not provide constant time

for select operations.

The work by Raman, Raman, and Rao [RRR02, RRS07] (RRR) squeezes the space used to represent the sequence and the index to $nH_0(B) + o(n)$, while keeping the operations in constant time. The value $H_0(B)$ [WMB99] is the zero order entropy of B, which corresponds to the lower bound on the number of bits needed to codify B, given the probability of occurrence of the bits with value 0 or 1. To be more specific, it is defined as

$$H_0(B) = \frac{n-m}{n} \log \frac{n}{n-m} + \frac{m}{n} \log \frac{n}{m},$$

where m is the number of bits with value 1 , $n - m$ is the number of bits with value zero, and n is the length of the bitmap. Notice that $H_0(B) \leq 1$ for bitmaps.

The strategy of RRR is similar to the strategy of Pagh. They divide the sequence into blocks of size $b = \frac{\log n}{2}$, and each block is represented by the numbers m_i and o_i. The m_i values are stored using $\log b$ bits, while the o_i values require $\log \binom{b}{m_i}$ bits. Both values can be represented using variable length encoding. As in Jacobson's and Clarks' work, a multilevel directory is used to achieve constant time for computing rank/select operations. In this way, the operations are computed by finding the precomputed values of rank/select in the directories, to then process the m_i and o_i in order to obtain the bitmap of the i-th block.

The work by Okanohara and Sadakane [SO07] proposes different methods for dealing with sparse and dense bitmaps. We will review the *sdarray* method that has been specially designed to deal with sparse bitmaps and perform select in constant time. In the *sdarray* method, the bitmap is splitted into two bitmaps L and H, obtained through a decomposition of the most significant bits of the integers indicating the positions of the ones in the bitmap. Given a bitmap B of size n with m ones at positions $x_1, x_2, ..., x_m$ (i.e., $x_i = \mathsf{select}(B, i)$), each x_i is divided into z upper (i.e., most significant) bits, and w lower (i.e., least significant) bits. The lower bits are explicitly stored into an array L of size m, where each entry holds w bits. The higher bits are encoded using d-gaps in unary into a bitmap H of size $m + 2^z$, because there are m ones with 2^z zeros in the d-gaps. Then $\mathsf{select}(B, i)$ over the original bitmap can be performed by operating on H and L, by doing $(\mathsf{select}(i, H) - i) \times 2^w + L[i]$. The first part of the addition will give the higher part of each bit set to 1 in position x_i, while $L[i]$ recovers the lower bits.

Practical implementations Practical implementations of the Jacobson's strategy achieve small extra space (i.e., 5% of the bitmap size) by replacing the second level directory by a fast popcount operation[4]. This strategy keeps constant time for rank, but with the disadvantage of logarithmic time for select, only noticeable for large bitmaps [GGMN05, SO07]. Instead of using an extra table for storing the first level directory, the work in [GP13] interleaves the precomputed rank values in order to take advance of modern CPU cache architectures. The practical implementations of RRR bitmaps have been studied in [CN08, NP12, GP13]. Main optimization has been done by fixing the block size and the sample rate on the directories. The sdarray method has been implemented by its authors [SO07]. The implementations of the raw and compressed bitmaps are available in the Compact Data Structures Library (libcds) and the Succinct Data Structure Library (sdsl-lite) [GBMP14a].

[4] A popcount (population count) operation returns the number of non-zero entries ('1' bits) in a word.

2.3.3 Rank and select over symbol sequences

The rank/select operations can be extended to deal with sequences of symbols. Given a sequence $S[1,n]$ over an alphabet $\Sigma = [1,\sigma]$, the following operations are of interest:

- $\mathsf{rank}_c(S,i)$ returns the number of times that the symbol $c \in \Sigma$ appears in $S[1,i]$.
- $\mathsf{select}_c(S,j)$ returns the position of the j-th occurrence of $c \in \Sigma$ in S.
- $\mathsf{access}(S,i)$ returns the symbol in $S[i]$.

The multilevel directory used for binary sequences can be applied in sequences, but it requires much more memory than the $o(n)$ bits, because it would need to track the precomputed rank values for σ symbols. Another technique is to store a bitmap for each symbol in the alphabet Σ [NM07]. Then, the ones in the bitmap B_j indicate the positions where the symbol σ_j occurs in S. The advantage of this this strategy is that rank and select can be answered in constant time just using bitmaps. However, it requires to hold $\sigma \times n$ bits of space. We could use compressed bitmaps, for which the access operation still needs $O(\sigma)$ access to check which one of the σ bitmaps has an active bit. In what follows we review the Wavelet Tree, a succinct data structure capable of answering the rank/select/access operations in $O(\log \sigma)$ time using $O(n \log \sigma)$ bits of space.

2.3.4 Wavelet Tree

The Wavelet Tree [GGV03] is an elegant solution proposed for general sequences that reduces the rank/select/access operations over sequences with more than two symbols to operations on bit sequences.

The Wavelet Tree is a balanced binary tree, whose leaves are labeled with symbols in Σ and whose internal nodes handle a range $[a_v, b_v] \subseteq \Sigma$. The root node handles the range $[1,\sigma]$ and the leaf nodes handle a symbol. Each node v is a bit sequence $B_v[1,n_v]$ that represents a subsequence $S_v[1,n_v]$ with the symbols in the range $[a_v,b_v]$. Each bit sequence $B_v[1,n_v]$ contains n_v bits, where its i-th value is defined as follows: if $S_v[i] \leq \frac{b_v - a_v}{2}$, then $B_v[i] = 0$; otherwise $B_v[i] = 1$. Then, a symbol $S_v[i] \in [a_v,b_v]$ with $B_v[i] = 0$ ($B_v[i] = 1$) will be represented in the left child (right child) of v in the next level of the tree. An example of a Wavelet Tree is shown in Figure 2.1(a), where the data structure only stores the bitmaps B_v shown in bold.

access/rank/select **on Wavelet Tree** The access operation traverses a path of the tree from the root to a leaf, following the i-th value in the bitmap B_v of each node in the path. At the root node, if $B_v[i]$ is 0 (1), it descends through the left (right) child. The position i at the child node is updated to $\mathsf{rank}_0(B_v,i)$ ($\mathsf{rank}_1(B_v,i)$). This is done recursively until it reaches a leaf node, where the symbol $S[i]$ is the answer. The $\mathsf{rank}_c(S,i)$ operation is done similarly, but descending through the left (right) child corresponding to the symbol c. The i position in the bitmap B_v at each node corresponds to $\mathsf{rank}_0(B_v,i)$ ($\mathsf{rank}_1(B_v,i)$) if c belongs to the left (right) child. At the leaf node, the answer is i. The $\mathsf{select}_c(S,j)$ operation follows an upward traversal of rank, starting from the leaf node representing c. If the leaf corresponds to the left (right) child of its parent, the position in the parent node v is $\mathsf{select}_0(B_v,j)$ ($\mathsf{select}_1(B_v,j)$). At the root node the answer is j.

The Wavelet Tree solves the operations rank, select, and access in $O(\log \sigma)$ if constant solutions for bit sequences are chosen. In its basic form, this structure uses $n \log \sigma + o(n \log \sigma)$ bits when using uncompressed bitmaps, plus $O(\sigma \log n)$ bits to store the topology of the tree [Nav12]. The space can be reduced to $nH_0(S) + o(n \log \sigma)$ using compressed bit sequences [GGV03, RRS07], plus the topology of the tree. The value $H_0(S)$ corresponds to the entropy of the sequence S. Notice that $H_0(S) \leq \log \sigma$.

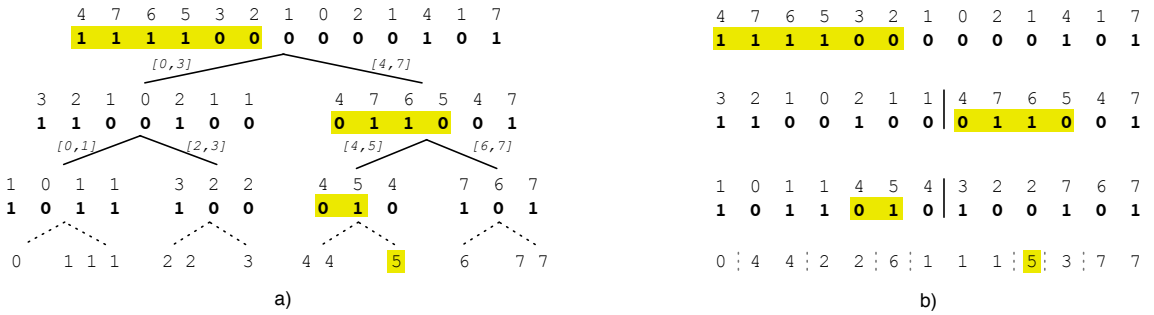

Figure 2.1: Wavelet tree reprsentations over a sequence $S = \langle 4, 7, 6, 5, 3, 2, 1, 0, 2, 1, 4, 1, 7 \rangle$: a) The Wavelet Tree, where the topology of the tree is explicitly stored as pointers. *b*) The Wavelet Matrix, where vertical lines are the z_l divisions and the topology of the tree is inferred during running time (image extracted from [CN12]). The ranges of bitmaps retrieved by operation $\mathsf{rank}_5(S, 6)$ are highlighted with a darker background.

Besides the basic operations proposed for sequences, the Wavelet Tree is capable of supporting other types of queries, using the same space. For example, Makinen and Navarro [MN06] connected the Wavelet Tree to a classical data structure for range search [Cha86, Cha88]. Also, Gagie *et al.* [GNP12] developed an algorithm to answer the rangeNextValue query [CIK+12]. These operations are defined as follows:

- rangeCount$(S, [i, j], [a, b])$ returns the number of distinct symbols in $S[i, j]$ that fall in $[a, b] \subseteq \Sigma$.
- rangeReport$(S, [i, j], [a, b])$ returns the number of occurrences per each symbol in $S[i, j]$ that fall in $[a, b] \subseteq \Sigma$.
- rangeNextValue$(S, [i, j], c)$ returns the smallest symbol $S[r]$ such that $S[r] \geq c$ and $r \in [i, j]$.

Operations rangeCount and rangeNextValue are solved in $O(\log \sigma)$, while rangeReport is solved in $O((1 + k) \log \sigma)$, where k is the number of symbols reported. Gagie *et al.* [GNP12] made some extensions to the algorithm for range queries. They defined rangeNextValuePos that returns the position of the symbol returned by rangeNextValue, and rangePrevValue that corresponds to the inverse of rangeNextValue. These new operations are also computed in logarithmic time following the same strategy of rangeNextValue proposed by [GNP12].

rangeReport on Wavelet Tree The rangeReport$(S, [i, j], [a, b])$ operation works like the rank operation, but instead of traversing down through the left or right child, it descends

to child nodes whose symbols' ranges intersect the range $[a, b]$. Range $[i, j]$ is re calculated on each descending step, accordingly to the alphabet of the left and right child. It starts at the bitmap of the root node $B_v[i, j]$, mapping the range going to the left child $[i_l, j_l]$, with $i_l = \mathsf{rank}_0(B_v, i)$ and $j_l = \mathsf{rank}_0(B_v, j)$. The range $[i_r, j_r]$ for the right child is updated, with $i_r = \mathsf{rank}_1(B_v, i)$ and $j_r = \mathsf{rank}_1(B_v, j)$. This continues recursively until the interval $[i, j]$ becomes empty or the range of the node does not intersect $[a, b]$. At leaf node, the frequency of a symbol $c \in [a, b]$ is $j - i$.

rangeNextValue **on** Wavelet **Tree** The rangeNextValue$(S, [i, j], c)$ operation traverses down the tree looking for those nodes that handle symbols that are $\geq c$. At each descending step, it recalculates the range $[i, j]$ according to whether or not c belongs to the left/right child. At the root node $B_v[i, j]$, if c belongs to the left (right) child, it descends and updates the range $[i, j]$ to $[\mathsf{rank}_0(B_v, i-1)+1, \mathsf{rank}_0(B_v, j-1)+1]$, respectively (i.e., the range becomes $[\mathsf{rank}_1(B_v, i-1)+1, \mathsf{rank}_1(B_v, j-1)+1]$).

This continues recursively until it finds a leaf. If the interval $[i, j]$ becomes empty returning from a left child, it still has the opportunity to find a symbol $\geq c$ traversing down the right child. Then, the c symbol must be updated to the minimal symbol in the alphabet of the right child. If the interval becomes empty from a right child, this means that no answer is available, because all possible options were tried. The final output is the symbol in the leaf, or empty if no answer is available. To return the position where the rangeNextValue is found, one needs to do an upwards traversal as in select operation.

The Wavelet Tree has also been used to represent multidimensional data as sequences of one dimension; for example, graphs [CN10a], general binary relations [BCN13], two dimensional rectangles [BLNS13], point grids [BLNS09], among others. The objective of representing multidimensional data as sequences is to diminish the space used by data, using a compressed representation of sequences. The rough idea behind these strategies is to group data (for example, by coordinates of the Euclidean plane) and then to represent implicitly the data that share the same value as a subsequence. This strategy follows a similar idea developed by Chazelle [Cha86, Cha88] to map the representation of a two-dimensional space into one-dimensional space.

2.3.5 Wavelet Matrix

A component of the space cost of the Wavelet Tree that can become relevant is the space used to store the topology of the tree. For example, when the sequence has the same length than the number of symbols, i.e., $n = \sigma$, half of the space is used to store the topology of the tree. As an alternative, the Wavelet Matrix [CN12] is a representation of a Wavelet Tree that provides fast navigation and uses a special arrangement of the bit sequences to implicitly represent the topology of the tree.

The basic idea behind the Wavelet Matrix is to store a single bitmap $\tilde{B}_l[1, n]$ per each level of the tree, which is a permutation of the bits in the level. The permutation of bits follows a simple rule: *all* bits with value 0 at one level go to the left, and *all* bits with value 1 go to the right in the next level [CN12]. Each level stores a single integer z_l representing the number of zeros at level l. The topology of the tree is inferred in running time as follows: If $\tilde{B}_l[i] = 0$, then position of the symbol s_i in the level $l + 1$ is $rank_0(\tilde{B}_l, i)$. Analogously,

if $\tilde{B}_l[i] = 1$, the position of s_i in $l+1$ is $z_l + rank_1(\tilde{B}_l, i)$. Note that with this type of permutation, a subset of symbols $[a_v, b_v]$ at level l always appears in a subsequence in $\tilde{B}_l$. The extra space required to store z_l is $\log\sigma \log n$, much less than the space required for the Wavelet Tree. Figure 2.1(b) shows the Wavelet Matrix equivalent to the Wavelet Tree in Figure 2.1(a). See [Nav12] and [Mak12] for further review of compressing techniques and applications of Wavelet Trees.

access/rank/select **on** Wavelet **Matrix** Operations on the Wavelet Matrix simulate the traverse on the Wavelet Tree using the z_l values, mapping the B_v bitmaps of the Wavelet Tree as a subsequence in the $\tilde{B}_l$ bitmaps. The access operation starts at level 0 of the matrix, if $\tilde{B}_0[i]$ is 0, it descends to the next level with the value i updated to $\mathsf{rank}_0(\tilde{B}_0, i)$. Otherwise, it descends with i set to $z_0 + \mathsf{rank}_1(\tilde{B}_0, i)$. This is done recursively until it reaches the last level, where the $S[i]$ symbol is the answer. The $\mathsf{rank}_c(S, i)$ operation needs more than tracking the position i as in the Wavelet Tree, it also needs to carry the subsequence $B_l[p, i]$ in level l related to the range of symbols in the left (right) child of node v in the Wavelet Tree. Initially $p = 0$. Then, at each node v at level l, if c belongs to the first half of the alphabet in node v, it virtually goes to the left child of v setting $i = \mathsf{rank}_0(\tilde{B}_l, i)$ and $p = rank_0(\tilde{B}_l, p)$. Otherwise, it sets $i = z_0 + \mathsf{rank}_1(\tilde{B}_0, i)$ and $p = z_0 + \mathsf{rank}_1(\tilde{B}_l, p)$. At the last level, the answer corresponds to $i - p$. The $\mathsf{select}_c(S, j)$ operation needs to map the leaf node of c to a subsequence at the last level of $\tilde{B}_l$, and track upward its range up to the first level. This is done by saving the p value while it descends towards the last level in the rank operation. Then, it tracks upward the position $i = p + j$ in the last level. If it descended through the left child at level l, it sets the position i to $\tilde{B}_l[\mathsf{select}_0(\tilde{B}_l, i)]$. Otherwise, it sets the position to $\tilde{B}_l[\mathsf{select}_1(\tilde{B}_l, i - z_l)]$. This is done until the first level, where the answer is i.

Claude *et al.* [CNO15] developed the range queries for the Wavelet Matrix, by following the reorder of bitmaps. For consideration of space, we are not including these algorithms, which are based on the strategy to compute the rank operation [CN12].

2.3.6 Compact binary relations

A binary relation links objects from one domain to another domain. Good examples of binary relations are graphs, trees, and even, inverted indexes. Graphs are constituted by edges that denote a relation between vertices, trees relate nodes in a hierarchical way, and inverted indexes relate words with documents. We consider a binary relation $\mathcal{R}$ as a set of t pairs (x, α) over an ordered set of objects T of size n, and an ordered set of labels Σ of size σ; such that x belongs to T and α belongs to Σ. A binary relation can be seen as a binary matrix of size $n \times \sigma$, where columns denote objects, and rows denote labels. Then each pair (x, α) corresponds to an active cell in the binary matrix (Figure 2.2). Operations over binary relations are based on counting and retrieving pairs that match with a label, or the retrieval of the i-th object with a given label, among others [BGIMSR07]. A complete and detailed description of all operations over binary relations is available at [BCN13].

A simple representation reduces a binary relation to a bitmap and a string [BGIMSR07, CN10c, BCN13]. Consider a bitmap B of size $n+t$, and a sequence $S[1, t]$ over an alphabet $\Sigma[1, \sigma]$. The bitmap B encodes the number of pairs found in each column (i.e., the number of labels related with an object) in unary. Let p_i be the number of pairs related with the object x_i, then $B = 1^{p_1}01^{p_2}0...1^{p_\sigma}$. The sequence S is composed by the objects found in each

	1	2	3	4	5	6	7	8	9
A	.	.	1	.	.	.	.	.	.
B	.	.	.	.	.	1	1	.	.
C	.	.	.	1	.	1	.	1	.
D	.	1	.	.	.	.	.	.	.
E	1	.	.	1	1	.	.	.	.
F	.	.	.	.	.	.	.	.	1
G	.	.	.	.	1	.	1	.	.
H	1	1	.	.	.	.	.	.	.

Figure 2.2: A binary relation composed by 15 pairs. The binary relation is represented as a binary matrix where each cell encodes a pair. Columns denote objects and rows denote labels. Figure extracted from [BCN13].

column, ordered by row (i.e., the labels grouped by object in ascending order). Figure 2.3 contains the bitmap and string representation of the binary relation in Figure 2.2. This simple representation allows us to retrieve the labels related to an object x_i by accessing the elements in the subsequence defined by $S[\mathsf{select}(B, i), \mathsf{select}(B, i+1) - 1]$. By representing the sequence S in a Wavelet Tree, Barbay. *et al.* [BCN13] showed that more complicated operations can be reduced to operations in the Wavelet Tree, many of them answered in $O(\log \sigma)$ time. By using RRR bitmaps on the Wavelet Tree, the space of this representation is close to the information-theoretic lower bound for a binary relation of $\log \binom{n \times \sigma}{t}$ bits [BCN13].

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
B	1	10	1	10	10	1	10	1	10	1	10	1	10	10	10

Figure 2.3: The binary relation of Figure 2.2 reduced to a string S and a bitmap B. Figure extracted from [BCN13].

2.4 Compressed data structures

In this section we present some important compressed data structures that will be used along this thesis. The first structure is the k^2-tree, a compact data structure to represent sparse binary matrices. It takes advantages of sparseness and regularities in many real-world matrices to achieve good compression ratios. The k^2-tree has been successfully applied in different contexts such as Web graphs [BLN14], binary relations [BdBN12], geographical raster data [dBÁGB$^+$13], and RDF databases [ÁGBFMP11].

The second structure is the Compressed Suffix Array (CSA) [Sad03], a self-index that provides fast search only requiring a space close to the higher order entropy of the text.

2.4.1 k^2-tree

In 1974, Finkel and Bentley [FB74] introduced the Quadtree, a data structure to maintain pairs of keys. The Quadtree can be seen as a generalization of a binary tree to answer queries regarding binary relations. For instance, suppose a database of employees and their salaries,

a sample query could be "to return all employees with a salary greater than X dollars", or in the case of cities in a map, a query could be "to find the nearest cities in a range of X km". A node in a Quadtree holds a pair (i.e., a 2D key) and four children. Each child node divides the space recursively into four regions. Then the subtree on each child contains the pairs inside a region. A Quadtree with N pairs requires to hold at least $O(N)$ nodes with 4 pointers each, which in practice requires at least $O(N)$ pointers.

In the early 90s, Wise and Franco [WF90] introduced the MatriX (MX) Quadtree to represent sparse binary matrices. The main difference with the original Quadtree is that regions are square and keys are the positions of ones in the binary matrix, and are represented by a path to leaf, as in a trie [Sam06] (instead of an internal node). Like in a sparse matrix, some areas are empty (i.e., they do not contain points), they can be pruned from the tree, reducing the total space. Although the MX is very useful for storing binary matrices, it still requires at least $O(N \log N)$ bits just to manage the pointers of the tree.

Conceptually, the k^2-tree [BLN14, BLN09] corresponds to a MX Quadtree with a succinct representation of the tree shape. The tree is encoded using the generalized Jacobson's level-order [Jac89a] [BDM$^+$05] for cardinal trees of degree k^2. This representation is based on binary sequences, which is capable of representing a tree of N nodes of degree k^2 in $Nk^2 + o(Nk^2)$ bits instead of $O(Nk^2 \log(Nk^2))$ bits of the classical pointer-based representation. The position of active cells is encoded as a root-to-leaf path (like in a trie or prefix tree). The k^2-tree works best when active cells are clustered, because leaves share a large portion of the root-to-leaf paths.

There are methods that compress Quadtrees based on reducing the length of the root-to-leaf paths (i.e., Compressed Quadtree [Cla83, AS99] and Skip Quadtree [EGS05]). However, they do not reduce the space used for encoding the position of the cells on the binary matrix. The Linear Quadtree [Gar82] is another encoding for the Quadtree that compresses the position of cells, which does not exploit the prefixes that are shared by similar root-to-leaf paths. The k^2-tree is a structure where the tree shape itself encodes the data (i.e., the position of active cells).

The k^2-tree stores a $n \times n$ binary matrix through a recursive decomposition of the input into k^2 square submatrices of size $n/k \times n/k$. This recursive decomposition of the input matrix continues until the submatrices are of size $k \times k$, corresponding to a leaf node representing k^2 cells. Each node of the tree has k^2 children, one per each submatrix. They are numbered from 0 to k^2-1, starting from left to right and top to bottom. Each submatrix is represented using a single bit: 1 if the submatrix has at least one cell, or 0 if the submatrix is empty. Thus, if the submatrix is empty, its children are not represented in the next level. The method proceeds recursively for each 1 child until the current submatrix is full of zeros or we reach the basic cells of the original matrix.

In this representation, each active cell of the matrix corresponds to a path in the tree; a mechanism analogously used by binary tries. This means that, for checking the state of a cell, one should check if a path exists until the leaves at level $h = \lceil \log_k n \rceil$. Figure 2.4 shows an example of a k^2-tree with k=2.

Notice that the k^2-tree does not store the boundary of each submatrix. Rather, this is implicitly represented by the path from the root node[5]. As the root node represents the whole matrix, its upper-left corner is located at $(0, 0)$, and the lower-right corner at (n, n).

[5]We assume that the boundary is closed at the upper-left corner and open at the lower-right corner.

The boundaries of internal nodes can be calculated while we traverse down the tree. Let (x, y) be the upper-left corner of the root node with $n \times n$ the size of the matrix. By definition, the size of the submatrix of each child node is $n/k \times n/k$. The upper-left corner of the i-child is defined by (x', y'), with $x' = x + n/k \times (i \mod k)$ and $y' = y + n/k \times (i/k \mod k)$. The lower-right corner is defined by the upper-left corner plus the size of the child submatrix, that is, $(x' + n/k, y' + n/k)$. This continues recursively until we find a leaf node or an empty submatrix. This gives us a time for checking the state of a cell of $O(\log_{k^2} n^2) = O(\log_k n)$, which corresponds to the height of the tree h.

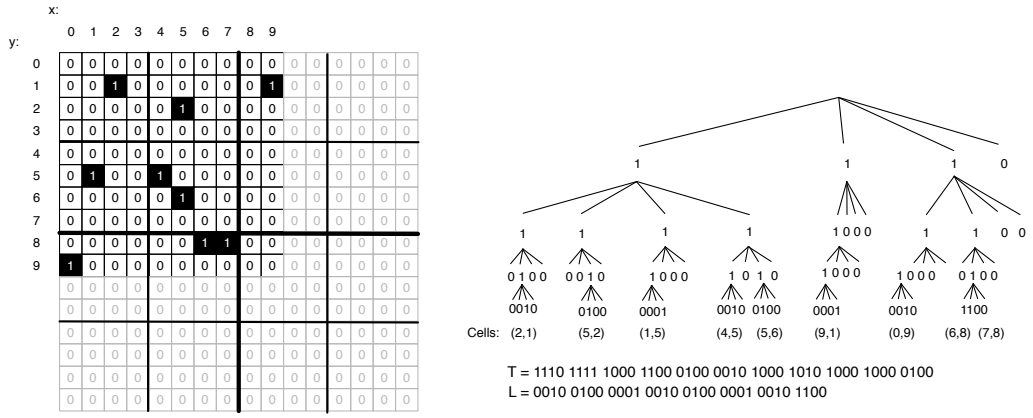

Figure 2.4: A k^2-tree representation for a 10×10 binary matrix.

The k^2-tree is traversed level wise and stored in two bit arrays: T stores the internal levels, and L stores the leaf level, as shown in Figure 2.4. The first of the k^2 children of a 1 bit at position p in T will be at position $p' = rank_1(T, p) \times k^2$. If p' is larger than the size of T, we will access the array L at the position $p'' = p' - |T|$. Notice that this property holds because each bit set to 1 at a level of the k^2-tree adds k^2 bits to the next level. On the other hand, bits set to zero do not have descendants.

The total space used by a k^2-tree depends on the distribution of the input. A complete analysis is shown in [BLN14]. Basically, in the worst case (for a matrix of size $n \times n$ with a uniform distribution of m 1s), the total space is $k^2 m \log_{k^2} \frac{n^2}{m} + O(k^2 m)$ bits. In real collections, such as Web graphs [BLN14] or Spatial data [dBÁGB$^+$13], this upper bound is far away. Compression is achieved because in these domains data is clustered, which generates leaves that share a high portion of the path from the root (i.e., self-similarity). Indeed, different node orderings influence how much the 1s in the matrix are clustered. A simple strategy to improve space is to permute (or rename) the nodes by following a breadth-first search [AD09].

The time cost of the different operations supported by the k^2-tree depends also on the characteristics of the matrix. Ladra *et al.* [BLN14] experiment with different Web Graphs, showing that clustered matrices have a better performance compared with matrices with a uniform distribution of ones.

Some enhancements have been proposed to improve space and time performance of the k^2-tree. These improvements use the following strategies: varying the k value at different levels of the tree, compressing the bitmap L (last level of the tree), and compressing the

submatrices that are full of ones. A brief description of these improvements follows.

Hybrid k^2-tree. A higher value of k produces a shorter tree, improving the query time at the cost of increasing the space. The Hybrid k^2-tree [BLN14] mixes different values of k, with higher values for the first levels and lower values for the last levels. With this strategy the space of the final data structure remains the same, but the time performance improves in matrices with a clustered distribution of 1s (i.e. Web graphs).

Compression of L. The last level of the k^2-tree represents submatrices of size $k \times k$. An alternative to reduce the space of L is to create a vocabulary of the non-empty submatrices at the last level, sorting it by its frequency like in statistical compression. Then, the L bitmap can be replaced by a list of integers pointing to the vocabulary of submatrices. With a skewed distribution of the submatrices, a variable-length encoding representation of the integers will reduce the space, because small numbers should be more frequent than larger ones. Ladra *et al.* [BLN14] compressed these integers using Directly Addressable Codes (DACs) [BLN13], whose main property is the direct access to any position. Moreover, better compression can be achieved by taking larger submatrices $k^l \times k^l$, this is, stopping the recursive decomposition at a higher level $l < h - 1$. Like in the Hybrid k^2-tree, experiments showed that this method only works when the matrix has a clustered distributions of 1s.

Compression of full-of-ones zones. A variant of the k^2-tree has been proposed by de Bernardo *et al.* [dBÁGB$^+$13] to compress larger zones of ones in matrices representing raster data. The basic idea is to stop the recursive decomposition when a zone is full of zeros (like in the original k^2-tree) or when a submatrix is full of ones. They developed two strategies to encode the new kind of nodes in the k^2-tree (i.e., the node encoding a zone that is full of ones). In the 2-bits variant, both empty and full-of-ones submatrices are encoded with a 0 in T. To distinguish if a position p in T is empty or full of ones, they add a second bitmap T'. If the position $T'[rank_0(T, p)]$ is set to 0, it means that the zone is empty; otherwise it is full of ones. This mechanism allows a reduction of the space used by the k^2-tree preserving the same time performance.

In Chapter 6 we will review two extensions of the k^2-tree that are designed to deal with cells in a multidimensional binary matrix: the k^d-tree [dBR14] and the Interleaved k^2-tree [GBBN14, Gar14]. These extensions have been used for representing ternary relations such as RDF triples and temporal graphs.

2.4.2 Compressed Suffix Array

In 1973, Weiner [Wei73] introduces the Suffix Tree, a data structure designed to improve the linear time of traditional pattern matching algorithms. A Suffix Tree is a trie of all the suffixes in a string $T[1, n]$ of size n over an alphabet of $\Sigma[1, \sigma]$. It is capable to locate the occurrences of a pattern $P[1, m]$ in $O(m)$ time. However, it requires to hold at least $O(n)$ nodes [McC76]. In practice, this means that the Suffix Tree requires at least 17 times the size of the text [Kur99].

In early 90s, Manber and Myers [MM93] introduced the Suffix Array (also known as PAT arrays [GBYS92]), a data structure created to overcome the space-consuming of the Suffix Tree. Manber and Myers described the Suffix Array as a sorted list of the suffixes in the

text. Specifically, it is an array of integers $A[1,n]$ of length n, indicating the position of all the suffixes in the text. The suffixes are lexicographic ordered, such that a suffix i is always before the suffix $i+1$, that is, $T[A[i],n] < T[A[i+1],n]$. It requires $n \log n$ bits on top of the text, and it is capable of answering pattern matching operations in $O(m \log n)$ time (which can be improved to $O(m+\log n)$ using extra space). The idea is to simulate the tree traversal of the Suffix Tree through a binary search, by combining the information in the Suffix Array and the text. To locate a pattern $P[1,m]$, we require to search an interval $[s,e]$ in A where the pattern occurs. The interval is obtained through two binary searches. The first binary search obtains the position s where the suffixes are not lexicographically less than P (equal or greater). The second binary search gets the ending position e, which corresponds to the position of the first suffix lexicographically greater than P. The final answer are the suffixes in $A[s,e]$. In Figure 2.5 we show the suffix array for the text `"abracadabra"`. The suffixes starting with "ab" are located in positions $A[7,8]$. In practice, the Suffix Array requires at least four times the size of the text, much less than the space required by the Suffix Tree.

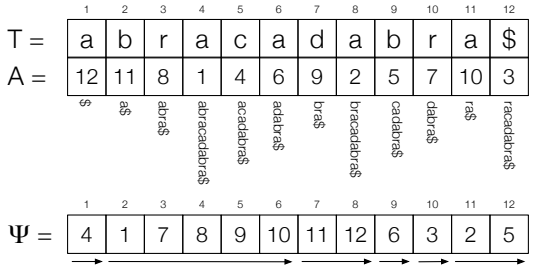

Figure 2.5: Suffix Array and Ψ function for the text `"abracadabra"`.

Although the Suffix Arrays improves the space used by Suffix Trees, they still need access to the text. In the early 2000s, Grossi and Vitter [GV00, GV05] and Sadakane [Sad02, Sad03] introduced the Compressed Suffix Array (CSA), the first attempt to reduce the space of the Suffix Array leaving aside the text. They figured out that suffixes sharing a common prefix (i.e., repeated phrases, words or self repetitions) can be exploited to reduce the size of the Suffix Array [NM07]. The CSA replaces the suffix array $A[1,n]$ by a Ψ function that is defined as $\Psi(i) = i'$ such that $A[i'] = A[i] + 1$ (if $A[i] < n$), and $\Psi(i) = 0$ if $A[i] = n$. In other words, $\Psi(i)$ maps the suffix in $T[A[i],n]$ to the next suffix in the text $T[A[i]+1,n]$. The function Ψ has a powerful property, it can extract the prefix of a suffix $T[A[i],n]$ by several recursive applications. As $A[\Psi(i)] = A[i] + 1$, the first character is at position $A[i]$ in the text; the next symbol is at pos $A[\Psi(i)]$, while the following in $A[\Psi(\Psi(i))]$, and so on.

The CSA can recover the text by a recursive application of Ψ and a simple representation of the symbols in the text. This representation is based on the frequency of symbols Σ in the text. Let $D[1,n]$ be a bitmap of size n with σ ones of the form $D = 10^{c_1-1}10^{c_2-1}...10^{c_\sigma-1}$, where c_i is the frequency of the symbol σ_i in the text. In this way, we can obtain the symbol σ_j in $T[A[i]]$ by doing $j = \mathsf{rank}(D,i)$. As the binary search on the Suffix Array only relies on a comparison of the suffix at pos $A[i]$ against the pattern, it can be simulated through the rank operation in D and a recursive application $\Psi(i)$ to obtain the next symbol in the suffix (i.e., the Suffix Array A is no longer required).

The CSA requires $nH_0(T) + O(n \log \log n)$ bits of space. However, when the text is

highly compressible (i.e., suffixes share prefixes or self repetitions), successive values of Ψ are in increasing order. This produces *runs* of the style $\Psi(i+1) = \Psi(i) + 1$. In Figure 2.5 we denoted the *runs* generated by the text `"abracadabra"`.

By using variable length encoding for representing Ψ, and run length encoding for the *runs*, the space of the CSA is closer to $2nH_k(T) + O(n)$ bits [NM07] (the k-order entropy of the text). In that case, the recovery of $\Psi[i]$ takes $O(\log n)$ (due sampling). Then, the search of a pattern can be answered in $O(m \log^2 n)$ time [NM07, FBN$^+$12b].

Fariña et al. [FBN$^+$12b] extended the CSA to deal with sequences of words instead of characters. In this word-based CSA (WCSA), the text is replaced by a sequence of integers indicating the position of words as they appear in the text. The compression ratio of the WCSA is slightly better than the classic inverted index, but is much faster on the search of phrases.

For a complete review on several compressed self-indexes, see the survey by Navarro and Makïnen [NM07]. Practical implementations of the CSA can be found in the Pizza&Chili Project[6] and in the sdsl-lite library.

[6] `http://pizzachili.dcc.uchile.cl/`.

Chapter 3

Temporal Graphs and previous works

Informally, a temporal graph is a graph where edges can appear or disappear along time [NTM+13]. For example, consider how people are subscribed to social networks and how their friendship relations change along time, how devices connect and disconnect in a communication network, or how humans are in contact along time [TLS+13]. All these previous examples have a set of vertices and the relationships between these vertices (i.e., friendship relations, connections, or physical contacts) that may change along time.

In what follows, we introduce a formal definition of temporal graphs and a brief description of strategies and data structures used to represent temporal graphs.

3.1 Temporal Graphs Concepts

In the literature, temporal graphs are also named *evolving graphs* [Fer02] or *time-varying graphs* [CFQS11, SQF+11, NTM+13]. Among the different definitions of temporal graphs, we use the one introduced by Nicosia *et al.* [NTM+13][1]. In this definition, the time-varying states of edges are captured by the notion of *contacts* between vertices. This notion of contacts embeds the idea that an edge can be active for several time points or time intervals along the lifetime of the temporal graph.

Definition 1 A *temporal graph* $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$ consists of a non-empty set of vertices V, a non-empty set of time instants $\mathcal{T} \subset \mathbb{N}$ with a total order [BWJ98] representing the *lifetime* of the graph, a set of directed edges $E \subseteq V \times V$ between vertices that are active at some $t \in \mathcal{T}$, and a set of contacts $\mathcal{C} \subseteq E \times \mathcal{T} \times \mathcal{T}$, such that for a contact (u, v, t_s, t_e), $[t_s, t_e)$ indicates the time interval when the edge $(u, v) \in E$ is active. We say that an edge $(u, v) \in E$ is *active* (or has an active state) at time instant t if there is a contact $c = (u, v, t_s, t_e)$ such that $t_s \leq t < t_e$; otherwise it is *inactive* (or has an inactive state). We also say that $v \in V$ is a neighbor or neighboring vertex of u at time t if the edge (u, v) is active at time t (See Figure 3.1(a)). As Holme [HS12] claims, we assume that there are no empty or overlapping intervals for a given edge. This is, given two contacts $(u, v, t_i, t'_i), (u, v, t_j, t'_j) \in \mathcal{C}$, then $t_i < t_j$ if and only if $t'_i < t_j$. □

Two additional concepts related to a temporal graph $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$ are the notion of *snapshot* and *aggregated graph* (see examples in Figure 3.1b and 3.1c).

Definition 2 A *snapshot* $G^{(t)}$ of the temporal graph $\mathcal{G}$ is the set of active edges at a given time point t. Formally, $G^{(t)} = \{(u, v) | (u, v, t_s, t_e) \in \mathcal{C}, t \in [t_s, t_e)\}$. □

Definition 3 The *aggregated graph* $G = (V, E)$ of a temporal graph $\mathcal{G}$ is the graph composed by all edges that have been active during the lifetime of $\mathcal{G}$. We define the *aggregated degree* of a vertex u as the degree of u in the aggregated graph. □

[1] Nicosia *et al.* [NTM+13] named the model as time-varying graph. In this work, we use the term temporal graph as a synonym of time-varying graph.

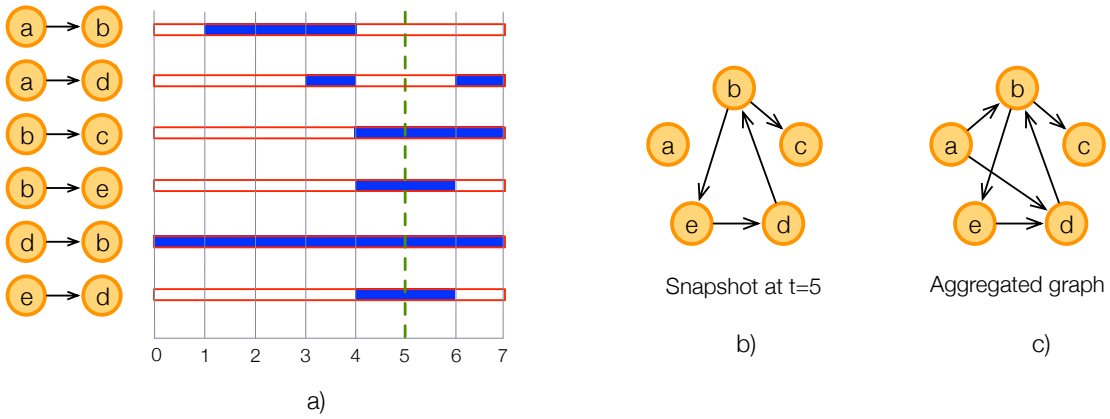

Figure 3.1: Examples of a temporal graph and snapshot: a) An example of a set of contacts among five vertices. b) Snapshot at $t = 5$ (dashed line on the set of contacts). c) The aggregated graph of the temporal graph (Figure adapted from [NTM+13]).

In what follow, we will denote $n = |V|$ the number of vertices, $m = |E|$ the number of edges, $\tau = |\mathcal{T}|$ the number of time points in the lifetime, and $c = |\mathcal{C}|$ the number of contacts.

3.1.1 Operations over temporal graphs

Temporal graphs can be useful in a wide range of domains, such as social graphs, telephone calls, air traffic control, and so on. In each domain, the queries of interest can be different. For example, in a temporal graph representing telephone calls, it may be of interest (for network traffic control) the number of calls (edges) started at a specific time point or during a time interval, while the same query may have no sense for a social graph. In addition, some useful queries can require the simple access to temporal graphs, while other could require more complex algorithms. Thus, we do not intend to define all possible queries that can be of interest for applications that use temporal graphs, we simply present a sensible classification, motivation, and formal definition of a set of basic queries that could be of interest in most application domains.

We start by considering operations (queries) over a temporal graph that extend queries over static graphs with temporal criteria. Then, we introduce queries about changes on edges of a temporal graph. Similar to spatio-temporal queries [PJT00, TP01], queries on temporal graphs can be constrained to a time point or time interval. For a time interval $[t, t')$ and queries about vertices and edges, we consider two possible semantics: (1) A *strong* semantics retrieves data that hold within the interval $[t, t')$. (2) A *weak* semantics retrieves data that occur at any time instant $t_q \in [t, t')$.

In what follows we classify and summarize the basic operations on a temporal graph $G = (V, E, \mathcal{T}, \mathcal{C})$. Operations with time-interval criteria are denoted by a subscript letter: $\mathcal{S}$ for strong semantics, $\mathcal{W}$ for weak semantics, and $\mathcal{I}$ for operations without a specific semantics. When the subscript letter is absent, they refer to a time point constraint. These operations cover the most useful queries in the applications we could think of.

Queries about vertices. These queries retrieve direct and reverse neighbors of a vertex constrained by a time point or time interval. These queries are useful to know who is connected with who at a specific time point or during a time interval, that is, Who were friends of X during the last year? or Who are the telephone numbers called by the number X yesterday?

- DirectNeighbors(G, u, t) lists all vertices v such that there is a contact (u, v, t_s, t_e) with $t_s \leq t < t_e$; that is, it returns $\{v|(u, v, t_s, t_e) \in \mathcal{C}, t \in [t_s, t_e)\}$.
- DirectNeighbors$_{\mathcal{W}}$(G, u, [t, t')) lists all vertices v under the weak semantics such that there is a contact (u, v, t_s, t_e) with $[t, t') \cap [t_s, t_e) \neq \varnothing$; that is, it returns $\{v|(u, v, t_s, t_e) \in \mathcal{C}, [t, t') \cap [t_s, t_e) \neq \varnothing\}$.
- DirectNeighbors$_{\mathcal{S}}$(G, u, [t, t')) lists all vertices v under the strong semantics such that there is a contact (u, v, t_s, t_e) with $[t, t') \subseteq [t_s, t_e)$; that is, it returns $\{v|(u, v, t_s, t_e) \in \mathcal{C}, [t, t') \subseteq [t_s, t_e)\}$.
- ReverseNeighbors(G, v, t) lists all vertices u such that there is a contact (u, v, t_s, t_e) with $t_s \leq t < t_e$; that is, it returns $\{u|(u, v, t_s, t_e) \in \mathcal{C}, t \in [t_s, t_e)\}$.
- ReverseNeighbors$_{\mathcal{W}}$(G, v, [t, t')) lists all vertices u under the weak semantics such that there is a contact (u, v, t_s, t_e) with $[t, t') \cap [t_s, t_e) \neq \varnothing$; that is, it returns $\{u|(u, v, t_s, t_e) \in \mathcal{C}, [t, t') \cap [t_s, t_e) \neq \varnothing\}$.
- ReverseNeighbors$_{\mathcal{S}}$(G, v, [t, t')) lists all vertices u under the strong semantics such that there is a contact (u, v, t_s, t_e) with $[t, t') \subseteq [t_s, t_e)$; that is, it returns $\{u|(u, v, t_s, t_e) \in \mathcal{C}, [t, t') \subseteq [t_s, t_e)\}$.

Queries about edges. These queries retrieve active edges constrained by a time point or time interval or the time when an edge has its next activation. These queries can be seen analogous to the previous ones but with both vertices bounded. Examples of these queries are: Were X and Y friends during the last year? Or does Y call X yesterday?

- Edge(G, u, v, t) is true if there is a contact $(u, v, t_s, t_e) \in \mathcal{C}$ such that $t_s \leq t < t_e$; otherwise, it is false.
- Edge$_{\mathcal{W}}$(G, u, v, [t, t')) is true under the weak semantics if there is a contact $(u, v, t_s, t_e) \in \mathcal{C}$ such that $[t, t') \cap [t_s, t_e) \neq \varnothing$; otherwise, it is false.
- Edge$_{\mathcal{S}}$(G, u, v, [t, t')) is true under the strong semantics if there is a contact $(u, v, t_s, t_e) \in \mathcal{C}$ such that $[t, t') \subseteq [t_s, t_e)$; otherwise, it is false.
- EdgeNext(G, u, v, t) returns t if there is a contact $(u, v, t_s, t_e) \in \mathcal{C}$ such that $t \in [t_s, t_e)$, or returns t'_s if there is a contact $(u, v, t'_s, t'_e) \in \mathcal{C}$ such that $t \leq t'_s$; otherwise, it returns ∞.

Query about the state of the graph. This query retrieves the state of the complete graph at a particular time instant. For example, Which in-air flights (between a pair of cities) were at 4:30 am? or Which pairs of numbers were connected by a call at 9 am?

- Snapshot(t) returns the set of active edges at time instant t.

Queries about changes/events on edges. These queries retrieve the changes that occur at a specific time point or during a time interval. For example, Whom called whom at 10 am or during the last weekend? or Which cities became connected with direct flights during the last summer? or Which cities lost direct flights during the autumn season?

- ActivatedEdges(G, t) lists all edges that have been activated at time t; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_s = t\}$
- DeactivatedEdges(G, t) lists all edges that have been deactivated at time t; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_e = t\}$
- ChangedEdges(G, t) lists all edges which state changed at time t; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_s = t \vee t_e = t\}$
- activatedEdges$_{\mathcal{I}}$(G, [t,t')) lists all edges that have been activated during the time interval $[t,t')$; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_s \in [t,t')\}$
- deactivatedEdges$_{\mathcal{I}}$(G, [t,t')) lists all edges that have been deactivated during the time interval $[t,t')$; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_e \in [t,t')\}$
- changedEdges$_{\mathcal{I}}$(G, [t,t')) lists all edges which state changed during the time interval $[t,t')$; that is, it returns $\{(u,v)|(u,v,t_s,t_e) \in \mathcal{C}, t_s \in [t,t') \vee t_e \in [t,t')\}$

Note that time point queries at time t can be reduced to time interval queries with interval $[t,t+1)$. We choose this set of operations because they provide the primitives to answer queries about local direct connectivity. More complex operations like journeys[2] or temporal graph measures can be computed by using the basic local queries of direct and reverse neighbors, adapting already existing algorithms in the literature [FV02, BXFJ03, SQF+11, HS12, NTM+13, TLS+13]. Table 3.1 presents examples of a subset of queries over the temporal graph in Figure 3.1.

Class	Operation	Point $t_q = 1$	Interval $t_q = [3,5)$	
			Weak Sem.	Strong Sem.
Vertices	DirectNeighbors of a	$\{b\}$	$\{b,d\}$	$\{\}$
	ReverseNeighbors of b	$\{a,d\}$	$\{a,d\}$	$\{d\}$
Edges	Edge (a,b)	True	True	False
	EdgeNext of (a,d)	3	-	-
State	Snapshot	{(a,b), (d,b)}	-	
Events	ActivatedEdges	$\{(a,b)\}$	$\{(a,d),(b,c),(b,e),(e,d)\}$	
	DeactivatedEdges	$\{\}$	$\{(a,b),(a,d)\}$	
	ChangedEdges	$\{(a,b)\}$	$\{(a,b),(a,d),(b,c),(b,e),(e,d)\}$	

Table 3.1: Examples of basic operations over the temporal graph in Figure 3.1.

[2]A journey or trajectory in a temporal graph is a time-ordered sequence of edges that have been active and that connect a starting and an ending vertex.

3.1.2 Types of temporal graphs

Graphs are classified based on the duration of the time intervals of contacts [HS12]. For all edges (u, v) of a temporal graph, if their associated contacts are of the form $(u, v, t, t + 1)$, with t in the lifetime of the graph, we say that the temporal graph is a *point-contact* temporal graph[3]; otherwise, we say that the temporal graph is an *interval-contact* temporal graph. Note that both kinds of temporal graphs fulfill Definition 1.

Using *point-contact* temporal graphs we can model communication systems, where the duration of a contact is a discrete event (e-mails, text messages, and so on) or where the connection itself is more important than the duration of the connection. In *interval-contact* temporal graphs, we can model systems where the interval of a contact is important; for example, proximity graphs (how much time two individuals are close to each other) and infrastructure systems like Internet (see [HS12] for a further review on temporal graphs).

Temporal graphs can be also classified by the dynamism of their contacts [DEGI10]. We say that a temporal graph is *fully-dynamic* if the contacts of any of its edges occur at any time point. If there exists only one contact per edge, we say that the graph is *partially-dynamic*. If all contacts start at the beginning of the lifetime of the graph, we say that the graph is *decremental* because, as time goes, there are fewer active edges. In contrast, a temporal graph is *incremental* if all contacts end at the end of the lifetime.

Depending on the type of the graph, some operations are always empty or equivalent to other operations. For example, a time interval query over a *point-contact* graph using a *strong* semantics always returns empty, because the duration of all contacts is a time point. The same happens with DeactivatedEdges operations on *incremental* graphs, which also return empty because all edges remain active until the end of the lifetime. This also implies that if we are asking for the DeactivatedEdges at the last time point of the graph, the operation is equivalent to a Snapshot, since all edges are deactivated at the end of the lifetime.

In an *incremental* graph, operations with a *weak* semantics over an interval $[t, t')$ can be computed by doing a time point query over t'. This is because as contacts always end at the end of the lifetime, the upper bound of the interval gets all the active contacts. Similarly, operations with a *strong* semantics can be answered by a point query over t because contacts remain active until the end of the lifetime. On *point-contact* graphs, the operation ActivatedEdges for a time instant t can be computed as a Snapshot(t), because all edges are active for only a time instant, the activated edges at t are also the edges that are active at t. Likewise, DeactivatedEdges for a time instant t can be computed as Snapshot$(t - 1)$ because the edges deactivated at t are those that were activated at $t - 1$.

3.2 Representations for temporal graphs

A temporal graph could be represented as several static graphs (or snapshots), one per each time point in the lifetime of the graph. A representation based on this idea is the *presence matrix* [FV02, p. 3], a binary matrix of size $m \times \tau$, where each cell (i, j) indicates the state of edge i at time point j. The main problem with this snapshot-based representation is that the space used is several times larger than the space used to store active edges in the

[3]Holme and Saramäki defined this as a contact sequence, but we renamed the concept to point-contact temporal contact.

temporal graph. Indeed, when edges remain unchanged a long time interval, a snapshot at time instant t is a small variation of the previous snapshot at time $t-1$. This observation was also pointed out by Salzberg and Tsotras [ST99] in what they call the ***copy*** strategy, which creates a specific entry for each time point in a data structure.

In the area of Semantic Web, some proposals for indexing temporal RDFs [GHV07, GHV05] fall in the ***copy*** category, pursuing time efficiency by using locality to minimize the number of read operations from secondary memory. A first proposal that was especially designed to efficiently answer graph-pattern queries is tGRIN [PUS08]. The basic idea of tGRIN is that vertices that are closer (in terms of time and path distance) to each other should be stored in the same page, since they will appear together in a query answer. Later, the work in [TB09] proposed a meta-index over intervals during which named graphs [CBHS05] are valid, and where each named graph has its own index.

A more compact representation of temporal graphs can be classified into two approaches: (i) ***log of events*** (or log of changes) is an approach that stores the events of activation/deactivation of edges or time intervals when edges are active and (ii) ***changes between snapshots*** is an approach that divides the log of events in small pieces and stores a snapshot between pieces. In terms of the classification proposed by Salzberg and Tsotras [ST99], these approaches correspond to the ***log*** and ***copy+log*** strategies, respectively.

In the ***log*** strategy the idea is to traverse the log of events on edges, adding or removing edges to the answer until the time of the query is reached. The main issue with this strategy is that the traverse is usually done in linear time with respect to the number of events, which can be slow for large temporal graphs. A lightweight version of the *presence matrix* [FV02] follows this strategy, which only stores the time points at which edges are activated/deactivated. A second work by the same authors [BXFJ03] presents a log of edges, a data structure based on linked adjacency lists that store a sublist per each neighbor sorted by time, indicating the time interval when an edge is active.

Distributed indexes also use log of changes. The DeltaGraph [KD13] is a distributed index that groups graphs in a hierarchical structure based on common edges. The G-star index [LOH13] stores versions of vertices for each time point when an edge is added or removed. With a control version system, it stores what changes occur with respect to an earlier version of the same vertex. This resembles the notion of overlapping B-trees [BHK85, BKMK90], also developed for multidimensional data structures HR-Tree [NST99] [NST98] and RT-Tree [XHL90], where the basic idea is that, given two trees, the most recent tree corresponds to an evolution of the older one, and subtrees can be shared between both trees.

The main goal of the snapshot-change or ***copy+log*** based representation is to find the closest snapshot to a query time t, and then to compute the changes from there. In [RLK+11], authors developed the *FVF-framework*, a compression strategy based on only storing what changes with respect to a representative graph, but focusing on solving other than adjacency queries over time. A preliminary work in [dBBCR13, GBBN14] describes three different methods to index temporal graphs, mainly based on representing data as a snapshot and a log of changes of vertices or edges. Similarly, the work in [RBR12] proposes a compact moving-object indexing based on the idea of storing snapshots of objects locations at some time points and a log of translations between consecutive snapshots.

Discussion In the literature we found several strategies for representing temporal data. They can be classified into ***copy*** and ***log*** strategies, which create an entry for each time

instant, or those storing logs of changes. The main issue with the ***copy*** strategy is its space cost for contacts that hold active for long periods of time. Conversely, with the ***log*** strategy we can reduce the space, but at the expense of time cost to retrieve the state of edges at certain time instant. A combination of both strategies, the ***copy+log*** strategy, acts like a sampling on the log. With this idea, we could balance the space and time used to retrieve temporal information. These different strategies can be seen as meta representations because they allow the use of different data structures to store the ***copy*** component (snapshot) and the ***logs***, without making big changes in the algorithms to obtain the operations for temporal graphs. In the next section we will review how to use compact data structures and compression techniques to implement these strategies.

3.3 Representing temporal graphs with compact data structures

In this section we will review the first attempts to reduce the space of data structures for temporal graphs. We first introduce some preliminary proposals that directly implement the ***copy*** and ***copy+log*** strategies using compact data structures for statics graphs and compression techniques designed for inverted indexes.

3.3.1 Snapshot k^2-tree

The Snapshot k^2-tree is the simplest method for reducing the space of the *presence matrix* (***copy*** strategy). The idea is to replace the snapshot representation for a k^2-tree. The main advantage of this method is that time point queries can be answered very fast, as the state of the graph is stored in only one data structure. Although this mechanism is easy to implement, it does not provide primitives for recovering the state of edges on time intervals. Thus, time interval operations can only be answered by successive access on each instant of the time interval query. As we pointed in the beginning of the section, the Snapshot k^2-tree does not improve the space when edges remain active for several periods of time. Thus, it is not suitable for temporal graphs whose contacts have long time intervals.

3.3.2 Differential k^2-tree

The differential k^2-tree [dBBCR13] is based on sampling the state of the temporal graph for some time instants, and storing the differences between the current snapshot (i.e., the set of active edges) of the graph with respect to the last sampling. The differences are composed by the edges whose states (active or inactive) in the current time instant changed with respect to the last snapshot. Both sampling and differences are represented using k^2-trees. An edge is active at time t_k if it appears exclusively in the sampling or only in the differential k^2-tree stored at t_k; otherwise it is inactive. The main problem with this representation is that the differences tend to be similar in consecutive time instants, thus, they store the state change of an edge many times. This strategy can be classified into the ***copy+log*** method, although the log is the accumulation of events since the first time instant until the current time point.

3.3.3 Log-based temporal graph index (ltg-index)

The ltg-index is a set of time-ordered *snapshots* and *logs of changes* (or simply *logs*) between consecutive snapshots (see Figure 3.2). The structure keeps a log for each vertex, storing

the edge and the time instant where a change has been produced. Thus, it can be seen as an implementation of the ***copy+log*** strategy.

To retrieve direct neighbors of a vertex, the previous snapshot is queried, and then the *log* is traversed adding or removing edges to the result. For reverse queries, we add a special graph called *delgraph*, which is composed of the removed edges within the time interval related to the *log*. This graph is used as an index to avoid check all the log entries in reverse queries. Thus, to recover the reverse neighbors we check the previous snapshot and the corresponding *delgraph* to decide which logs must be traversed. To compute time interval queries we need to check the logs whose changes fall into the query.

The *snapshots* and *delgraphs* are represented by k^2-trees. The changes in *logs* are stored using statistical compression on edges, and *d*-gaps on time instants.

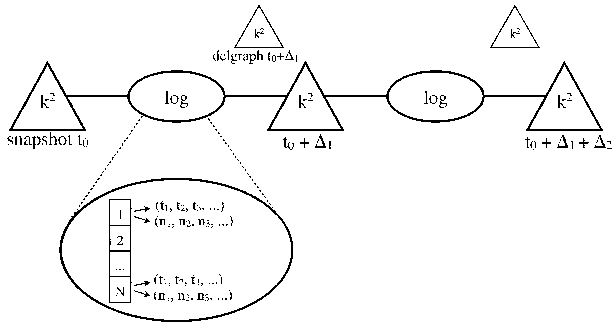

Figure 3.2: Components of the ltg-index

Discussion A preliminary version describing the data structures for temporal graphs presented in this section was published in [dBBCR13]. Empirical results show that the ltg-index obtains the smallest space in general, and achieves a similar time for the Snapshot k^2-tree. This suggests that with a careful design, the ***log*** could achieve less space keeping a reasonable time. The differential k^2-tree is a simple approach that is very fast when only time-instant queries are needed, but it requires more space than the ltg-index.

In the following section we will review how to represent temporal graphs under the ***log*** strategy using inverted index technology. We will also present a novel strategy for representing temporal graphs as a sequence of contacts.

Chapter 4

Preliminary proposals for compressing temporal graphs

This chapter first presents our two preliminary structures that reduce the necessary space to represent temporal graphs based on the ***log*** strategy. We use inverted indexes technology to compress the logs as d-gaps encoded using PForDelta [ZLS08, ZHNB06] and ETDC [BFNP07]. Although the basic ideas of the first structure (EdgeLog) were previously introduced in [BXFJ03], and the second one (EveLog) is a simple log of events, we present here their novel implementations using compressed inverted-indexes techniques to achieve space efficiency.

Later we introduce a novel approach of temporal graphs compression, based on representing the temporal graph as a sequence of contacts. The sequence is compressed using a slightly tuned version of the CSA self-index [Sad02, Sad03], which enables to perform operations without recovering all the sequence.

4.1 EdgeLog: The time-interval log per edge

The time-interval log per edge [BXFJ03] (denoted as EdgeLog) divides the representation of the temporal graph $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$ into its aggregated graph and the temporal information of its active edges. The aggregated graph is represented by $n = |V|$ adjacency lists, while an ordered list of time intervals attached to each neighbor of a node encodes temporal information. We denote by L_u the list of neighboring vertices of u found in the aggregated graph of $\mathcal{G}$. For each neighbor v in the adjacent list L_u, a list $L_{u,v}$ contains the ordered time intervals indicating the time when the edge (u, v) is active. We refer to the list of time intervals $L_{u,v}$ as the temporal log of (u, v). Figure 4.1 illustrates the example of the time-interval log per edge of the temporal graph in Figure 3.1.

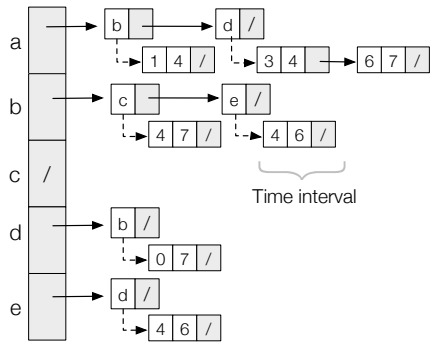

Figure 4.1: Time-interval log per edge of the temporal graph in Figure 3.1.

Notice that there are other possible implementations for the inverted lists[1] with different

[1]Possible examples are interval trees and segment trees.

features that will provide different space/time trade offs. However, as we will see in the experimental evaluation, this simple structure of inverted lists is easy to compress, achieving not only good compression ratios but also good time performance.

Compressing the data structure Both adjacency lists and temporal logs can be organized in monotonically increasing sequences of positive integers and be represented as d-gaps.

We now analyze the space requirements considering that the d-gaps are represented with an encoder using space close to the information-theoretic lower bound. Let $n_u = |L_u|$ the length of the adjacency list of vertex u. As the maximum number of neighbors is the set of vertices, we got that $n_u \leq n$, where $n = |V|$ is the number of vertices in the graph. Therefore, the compressed space of the adjacency lists is:

$$\sum_{u \in V} \log \binom{n}{n_u} \leq \sum_{u \in V} n_u \log \frac{n}{n_u} + \sum_{u \in V} O(n_u)$$

Assuming a uniform degree distribution in the aggregated graph, this is $n_u = m/n$, we obtain:

$$\begin{aligned} \sum_{u \in V} \log \binom{n}{n_u} &\leq \sum_{u \in V} \frac{m}{n} \log \frac{n^2}{m} + \sum_{u \in V} O(m/n) \\ &= m \log \frac{n^2}{m} + O(m) \end{aligned}$$

The procedure is similar for the temporal information. Let $n_{u,v} = |L_{u.v}|$ be the length of the list of the time intervals related to the edge (u, v). As the maximum length of a temporal log is the size of the lifetime $\tau = |\mathcal{T}|$, then $n_{u,v} \leq \tau$, the compressed space of the temporal logs is:

$$\sum_{u,v \in E} \log \binom{\tau}{n_{u,v}} \leq \sum_{u,v \in E} n_{u,v} \log \frac{\tau}{n_{u,v}} + \sum_{u,v \in E} O(n_{u,v})$$

Assuming a uniform distribution of contacts per edge, the temporal log of an edge is in average $n_{u,v} = \frac{2c}{m}$ $(\leq \tau)$. We use $2c/m$ since the length of all temporal logs is $\sum_{u,v \in E} n_{u,v} = 2c$, as a contact is represented using two time instants (the one that starts the contact, and another one that ends the contact). Therefore, the space is:

$$\begin{aligned} \sum_{u,v \in E} \log \binom{\tau}{n_{u,v}} &\leq \sum_{u,v \in E} \frac{2c}{m} \log \frac{m\tau}{2c} + \sum_{u,v \in E} O(2c/m) \\ &= 2c \log \frac{m\tau}{2c} + O(c) \end{aligned}$$

The final structure is composed by the n compressed adjacency lists, plus the m compressed temporal logs of time intervals. This gives a total space of $m \log \frac{n^2}{m} + 2c \log \frac{m\tau}{2c} + O(m + c)$ bits. The structure manages n pointers to the adjacency lists, and m pointers to the temporal logs. This require an extra space of $O(n \log m + m \log \frac{c}{m})$ bits.

Query processing The algorithm to obtain $\mathsf{Edge}(u, v, t)$ is straightforward: first we need to decompress L_u, then to check if neighbor v exists in L_u, then to decompress the temporal log $L_{u,v}$, and finally to check if there exists an interval $[t_s, t_e)$ in $L_{u,v}$ such that $t_s \leq t < t_e$. Assuming that the out-degree of u in the aggregated graph is m/n^2, $\mathsf{Edge}(u, v, t)$ can be done in $O(m/n+c/m+\log(m/n)+\log(c/m))$ through binary search over L_u and the temporal log $L_{u,v}$. This method is also valid to compute the $\mathsf{EdgeNext}$ query. Following the same strategy, an algorithm to answer $\mathsf{DirectNeighbors}(u, t)$ queries will take $O(m/n(1+c/m+\log(c/m)))$, because for each element v in L_u, it needs to check through a binary search, which one of the time-intervals in the temporal log $L_{u,v}$ containts t. Time-interval queries for active edges and direct neighbors have the same worst-case time cost, because they only require a second binary search in the list of time intervals. The $\mathsf{Snapshot}(t)$ query is computed by obtaining all the neighboring vertices active at time t through $\mathsf{DirectNeighbors}(u, t), \forall u \in V$.

Operations to recover the edges that have been activated/deactivated or changes at a time point t cannot be efficiently implemented on this structure, as they need to review the changes related to all edges. The idea is to obtain which of the $L_{u,v}$ time-interval lists contains t as an endpoint. If t is found at an even position, the edge has been activated at time t; otherwise, it has been deactivated at time t. Therefore, operations $\mathsf{ActivatedEdges}(t)$, $\mathsf{DeactivatedEdges}(t)$, and $\mathsf{ChangedEdges}(t)$ are computed in $O(m(1+c/m+\log(c/m)))$ time. The time-interval versions of these operations have the same running time, since they only require a second binary search.

4.2 EveLog: The adjacency log of events

The adjacency log (denoted by EveLog) is an event-based abstraction of a temporal graph represented by an array of n lists, one per each vertex in the graph. Each list L_u describes a time-ordered list of *neighboring changes* of a vertex u, from the first time point of the temporal graph. A *neighboring change* of a vertex u is a tuple (v, t) indicating that an event of activation/deactivation of the edge (u, v) occurs at time t. A neighboring change occurs at the endpoint of the time interval of each contact $c = (u, v, t_s, t_e) \in \mathcal{C}$, such that the edge (u, v) becomes active at time $t = t_s$ and inactive at time $t = t_e$. For this representation, all edges are initially inactive.

The EveLog resembles the basic idea introduced by Ferreira and Viennot [FV02] to reduce the space of the presence matrix, which stores the time points when the state of an edge changes to become active or inactive. Then, the time intervals when an edge is active can be reconstructed by traversing the adjacency log and updating the state of the edge in terms of the type of event in the log. The adjacency log is analog to the adjacency list for static graphs, but with neighboring vertices ordered by time. Figure 4.2 illustrates the example of the adjacency log for the temporal graph in Figure 3.1.

[2] Analytical time costs assume that the average number of active neighbors at time t is also m/n.

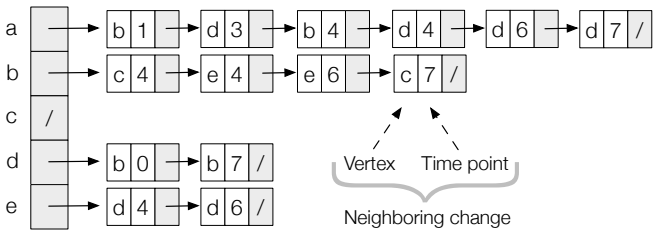

Figure 4.2: Adjacency log of the temporal graph in Figure 3.1.

Computing neighboring changes The state (active or inactive) of an edge is indirectly represented, because this can be derived by traversing the adjacency log of a vertex until finding the time of the query. An algorithm to obtain DirectNeighbors(u, t) updates the activation state of each edge, adding or removing a neighboring vertex of u for each event in L_u until finding the first t' such that $t' \geq t$. The algorithm maintains many entries as the number of neighbors of the vertex u in the aggregated graph.

Another way to answer DirectNeighbors queries is based on how many times a neighboring vertex appears in the time-ordered list of neighboring changes. This is possible because we know that all edges are initially inactive and, in consequence, the first occurrence of an edge in a list means that the edge becomes active, the second occurrence means that the edge becomes inactive, and so on. More formally, let L be the adjacency log of a temporal graph $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$ and L_u be the list of neighboring changes of vertex $u \in V$, then an edge $(u, v) \in E$ at time t is active if there exists an even number of pairs $(v, t') \in L_u$ for the largest t' such that $t' \leq t$. Inversely, the edge (u, v) is inactive if there exists an odd number of tuples $(v, t') \in L_u$ for the largest t' such that $t' \leq t$. We call this property of the log of events the *parity property*.

Compressing the data structure The idea to compress the data structure is to separate each adjacency log L_u of vertex u into two lists: (i) the list of time points T_u and (ii) the list of neighboring changes N_u, both in the same order as events occur. Since the list of time points T_u corresponds to a monotonically increasing sequence of positive integers, they can be represented as d-gaps. With respect to the list of neighboring changes N_u, we just need to store the neighboring vertex whose state changes, but not the type of change. In these lists, some values are more frequent than others, depending on the distribution of neighboring changes of the temporal graph. Then, statistical compression encoding is useful, taking into account the vocabulary frequency over all vertices in the lists of neighboring changes.

Now we analyze the space of the time instants of the neighboring changes. Let $n_u = |T_u|$ the length of the adjacency log of the vertex u. The maximum number of neighboring changes related to a vertex is twice the number of contacts in the graph, as one neighboring change is created when an edge becomes active, an another when it becomes inactive. As the time instants in T_u are less than the lifetime of the graph τ, the compressed space of

the list of time instants is:

$$\sum_{u \in V} \log \binom{\tau}{n_u} \leq \sum_{u \in V} n_u \log \frac{\tau}{n_u} + \sum_{u \in V} O(n_u)$$

Assuming a uniform number of contacts per vertex, the expected length of the list of time points is $n_u = 2c/n$ ($\leq \tau$). Then the compressed space can be expressed as:

$$\begin{aligned} \sum_{u \in V} \log \binom{\tau}{n_u} &\leq \sum_{u \in V} \frac{2c}{n} \log \frac{n\tau}{2c} + \sum_{u \in V} O(c/n) \\ &= 2c \log \frac{n\tau}{2c} + O(c) \end{aligned}$$

With respect to the list of neighboring changes N_u, we just need to store the neighboring vertex whose state changes, but not the type of change. In these lists, some values are more frequent than others, depending on the distribution of neighboring changes of the temporal graph. Then, statistical compression encoding is useful, taking into account the vocabulary frequency over all vertices in the lists of neighboring changes. Taking C_n as the concatenation of all N_u for all u in V, with $|C_n| = 2c$, then the compressed space used by the lists of neighboring changes is $2cH_0(C_n)$ bits. For our implementation, we choose the End-Tagged Dense Code [BFNP07] (ETDC) compression technique, because it has a faster decoding speed than Huffman encoding, with a 2.5% overhead over final space. Thus, we need to add the vocabulary table of ETDC, $n \log n$ bits.

The final space of EveLog is $2c \log \frac{n\tau}{2c} + 2cH_0(C_n) + O(c)$ bits, plus the space used by the pointers to access both lists T_u and C_n, $2n \log(2c)$ bits. Note that the final space of the data structure depends on the distribution of neighboring changes along time. Given that $H_0(C_n) \leq \log n$, the upper bound for this space is $2c \log \frac{n^2\tau}{2c} + O(c)$ bits, plus $n \log n$ bits for the space of the vocabulary table.

Query processing The algorithms to compute queries about the active edges and direct neighbors of a vertex u at time t are based on three main steps: (i) to get the list of time points T_u, (ii) to find t' in T_u such that t' is the lowest $t' > t$ and, finally, (iii) to traverse N_u until t' applying the parity property to obtain the state of an edge. The sequential counting of the parity property is upper bounded by the number of contacts in the temporal graph (i.e., $O(c/n)$)[3]. Similar to the case of EdgeLog, ReverseNeighbors queries are unfeasible in the EveLog, because all the adjacency lists should be revised.

The time cost to answer Edge(u, v, t) queries is $O(c/n + 1)$. This because we need to first decompress the list of neighboring changes in N_u, and then to check the number of occurrences of the vertex v. To compute the DirectNeighbors(u, t) query, we use a key-value data structure keeping an average of m/n elements (the average of neighboring vertices appearing in N_u). This gives us $O(c/n + m/n)$ of time cost, where the term m/n is the cost to check the number of occurrences of edges (i.e., vertices) in the list of neighboring changes. Time-interval versions of Edge and DirectNeighbors queries have the same time costs because there is only one extra check of the changes that occurred during the time interval of the

[3]Assuming a uniform number of contacts per vertex.

query. The Snapshot(t) query is answered by applying the DirectNeighbors(u, t) query for all u in V.

Like EdgeLog, the computation of the operation ChangedEdges(t) is not efficient. In this case, it needs to recover the neighboring changes occurred at time t for all vertices. The idea is to search the range $[p, p']$ where t occurs in T_u. Then, the answer is the set of neighboring changes in $N_u[p, p'], \forall u$. This operation has a time cost of $O(n \times (c/n + m/n))$, where the term m/n is the maximum size of the key-value data structure used to recover the set of neighbors that change their state. The time-interval version of this operation works similarly, but looking for the neighboring changes occurred at $[t, t')$.

The operation ActivatedEdges(t) can be computed as a difference between ChangedEdges(t) and Snapshot$(t-1)$. As the changes at time t can be an activation or a deactivation, by doing the difference we are obtaining the edges that were activated at time t. Similarly, the operation DeactivatedEdges(t) is computed as the intersection between ChangedEdges(t) and Snapshot$(t-1)$. The time costs of these operations are also $O(c+m)$. In the implementation, we compute both Snapshot and ChangedEdges one vertex at a time.

4.2.1 Reverse neighbors in EdgeLog and EveLog

Since EdgeLog and EveLog only support direct neighbors queries, we are including the inverted aggregated graph as a set of adjacency lists, each of them stored as d-gaps. This allows us to answer reverse-neighbor queries by just checking the state of the incoming neighbors, using $m \log \frac{n^2}{m} + O(m + n \log m)$ bits of extra space.

4.3 CSA for Temporal graphs (TG-CSA)

The *Temporal Graph CSA*[4] (TG-CSA) is an adaptation of Sadakane's Compressed Suffix Array (CSA)[Sad03] for compressing temporal graphs. It is based on the *word-based CSA* (WCSA) that allows CSA to deal with large (integer-based) alphabets (see [FBN$^+$12a] for details).

The main idea behind the TG-CSA is simple. First, we create a sequence S by concatenating all the contacts in a temporal graph. Then the sequence is represented in a CSA so that we obtain a searchable and compressed representation of a temporal graph. Therefore, neighboring operations can be performed by binary searches as in the pattern matching algorithm of the CSA.

Before we show how to create the sequence S, we need to map each contact of the temporal graph into a 4-tuple of integers. Given a temporal graph $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$, let $(u, v, t_s, t_e) \in \mathcal{C}$ be a contact in $\mathcal{G}$, and $M(u, v, t_s, t_e) = (m_1(u), m_2(v), m_3(t_s), m_4(t_e))$ be a mapping that transforms the contact (u, v, t_s, t_e) into a 4-tuple of integers. The mapping $M(\cdots)$ is defined as $m_1(u) = u$, $m_2(v) = v + n$, $m_3(t_s) = t_s + 2n$ and $m_4(t_e) = t_e + 2n + \tau$, where $n = |V|$ is the number of vertices in the graph, and $\tau = |\mathcal{T}|$ is the size of the lifetime.

The mapping $M(\cdots)$ defines an interval of integers for each component in the 4-tuple, such that the integers representing source vertices are smaller than those representing target vertices, and also smaller than those representing time instants. For instance, the interval of integers used in the first component m_1 is $\Sigma_1 = [1, n]$, in the second m_2 is $\Sigma_2 = [n + 1, 2n]$,

[4]The TG-CSA was a joint work with N. R. Brisaboa, A. Fariña and M.A. Rodriguez, presented in [BCFR14].

in the third m_3 is $\Sigma_3 = [2n+1, 2n+\tau]$, and in the fourth m_4 is $\Sigma_4 = [2n+\tau+1, 2n+2\tau]$. This allow us to distinguish if an integer is representing a source or target vertex, as well as the endpoints of the time interval of each contact.

Let $g = (u, v, t_s, t_e) \in \mathcal{C}$ be a contact, and $S(g) = m_1 m_2 m_3 m_4$ be the sequence formed by the components of the mapping $M(g)$. Let $g_1, g_2, \ldots, g_c$ be the set of contacts $\mathcal{C}$ in an increasing order by the first term, then by the second one, and so on. Then the sequence of the graph $\mathcal{G}$ is defined as $S = S(g_1)S(g_2)\ldots S(g_c)$ over an alphabet $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4$ (i.e., the union of the integers representing the source and target vertices, and the time interval). The size of the alphabet is $|\Sigma| = 2n + 2\tau$, and the length of the sequence is $|S| = 4c$ (as it is the concatenation of the four components of c contacts in the graph). Finally, a WCSA is built over S.

Compression in the sequence Regarding the nature of the sequence S, there is an important property that we conceptually describe here. Since the alphabets in the 4-tuples are disjoint, and they are ordered, such that $\Sigma_1 < \Sigma_2 < \Sigma_3 < \Sigma_4$, the first 25% entries in the suffix array A (i.e., $A[1, c]$) will point to the first terms of all the contacts, the next c entries (i.e., $A[c+1, 2c]$) to the second terms, and so on. Consequently, the first 25% entries of Ψ ($\Psi[1, c]$) will point to a position in the range $[c+1, 2c]$, because in the indexed sequence each symbol $u \in \Sigma_1$ is followed by a symbol $v \in \Sigma_2$, and so on. This property allows compression, since contiguous regions of Ψ are in increasing order. As in text compression, this produce *runs* of the form $\Psi(i+1) = \Psi(i)+1$ that are compressible in a space close to the high-order entropy of the sequence.

Recovering contacts Note that, in the standard CSA, if $A[i], (i \in [3c+1, 4c])$ points to the last term of the j-th contact, then $\Psi[i]$ would store the position in A pointing to the first term of the following $(j+1)$-th contact in the ordered list ($A[i] + 1 = A[\Psi[i]]$) that would be in the range $[1, c]$. However, we modify these pointers in the last 25% of Ψ, because instead of pointing to the position $x = A[\Psi[i]]$ corresponding to the first term of the following contact, we want them to point to the first term of the same contact. That is, $A[\Psi'[i]] = x - 1$, or $A[\Psi'[i]] = c$ if $x = 1$.

This small modification in Ψ allows us to retrieve the four components of each contact. If the first component is at pos i, the second is found at $\Psi[i]$, the third at $\Psi[\Psi[i]]$, and so on. With the modification, the first component is also found at $\Psi[\Psi[\Psi[\Psi[i]]]]$. With this modification, we loose the possibility of the recovering of the whole sequence, because consecutive applications of Ψ will cyclically obtain the four elements of the same contact, but this will not be required to compute the operations.

Dealing with a noncontinuous alphabet Note that even though vertex i always exists in the temporal graph, either source vertex $u' = m_1(i) = i$ or target vertex $v' = m_2(i) = i+n$ could actually not be used. Similarly a time t' could not occur as an initial or as an ending time of a contact, yet we could be interested in retrieving all the edges that are active at that time t'.

To overcome the existence of holes in the alphabet Σ, a bitmap $B[1, 2n+2\tau]$ is used. We set $B[i] \leftarrow 1$ if the symbol σ_i from Σ occurs in a mapped contact, and $B[i] \leftarrow 0$ otherwise. Therefore, each of the four terms within a mapped contact $M(u, v, t_s, t_e)$ will correspond to a

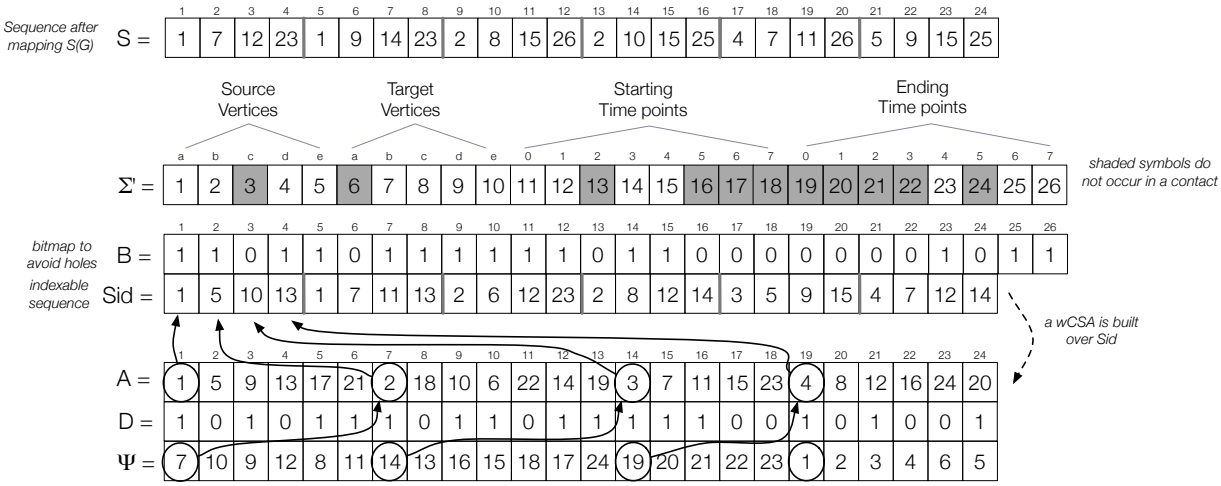

Figure 4.3: Structures involved in the creation of a TG-CSA for the temporal graph in Figure 3.1. Sequence S is constructed by applying the mapping $M(\cdots)$ to each contact.

1 in B. Now, an alphabet Σ' of size $\sigma' = rank_1(B, 2n+2\tau)$ is created containing the positions in B where a 1 occurs. For each symbol $\sigma_i \in \Sigma$ a *mapID(i)* function is defined that assigns an integer $i' \in \Sigma'$, so that $i' \leftarrow$ *getmap(i)* $= rank_1(B, i)$ if $B[i] = 1$, and $0 \leftarrow getmap(i)$ if $B[i] = 0$. The reverse mapping is provided via an $getunmap(i') = select_1(B, i')$ function. At this point, a sequence $Sid[1, 4c]$ over the new alphabet Σ' can be created by setting $Sid[i] \leftarrow getmap(S[i])$. This will be the final sequence represented in the CSA.

To sum up, the TG-CSA representation consists of a bitmap B, and the structures D and Ψ from the WCSA. By using run-length encoding for the runs in Ψ, the sequence representation of the graph requires $4cH_k(S)+O(c)$ bits, because the length of the sequence is $4c$. The upper bound of this space is $4c(1+\log(n+\tau))+O(c)$ bits, because the sequence belongs to an alphabet of size $2(n+\tau)$. The bitmap B is compressed with the Raman *et al.* strategy [RRR02], and D uses a faster bitmap representation from [FBN$^+$12a] with size of $1.375|D|$ bits. Figure 4.3 depicts all the structures involved in the creation of a TG-CSA that represents the temporal graph in Figure 3.1.

Query processing We can take advantage of the CSA capabilities at search time to solve all the typical operations of a temporal graph. Regarding DirectNeighbors and ReverseNeighbors operations, we binary search the range in $A[l, r]$ for the given source or target vertex, and for each position $i \in [l, r]$, we apply Ψ circularly up to the third or fourth component. Then, we check if either the starting-time and ending-time constrains hold or not. In Figure 4.4 we show the pseudocode of the algorithm to obtain DirectNeighbors. With the binary search and the sequential check of the temporal constraint, the expected time for answering DirectNeighbors is $O(\log c + c/n \log c) = O((1 + c/n) \log c)$ as each access to $\Psi[i]$ takes $O(\log c)$ time.

The Edge operation is expected to be faster than DirectNeighbors as we can do a pattern matching search with the phrase $u \cdot v$, rather than search by a unique vertex u. This phrase search will return a much shorter initial range than the search for the vertex u. However, it

Algorithm: DirectNeighbors(u, t) using the TG-CSA.

```
u ← getmap(m1(u));                          /* map into final alphabet without holes */
if u = 0 then return ∅;                     /* vertex does not appear as source vertex */
ts ← getmap(m3(t));
te ← getmap(m4(t));
N ← ∅;
[lu, ru] ← CSA_binSearch(u);                /* range A[lu, ru] for vertex u */
[lts, rts] ← CSA_binSearch(ts);             /* range A[lts, rts] for starting time ts */
[lte, rte] ← CSA_binSearch(te);             /* range A[lte, rte] for ending time te */
for i = lu to ru do
    x ← Ψ[i];
    y ← Ψ[x];
    if y ≤ rts then
        z ← Ψ[y];
        if z > rtf then
            N ← N ∪ {getunmap(m2^-1(x))};
return N;
```

Figure 4.4: Obtaining the direct neighbors of a vertex in a contact that is active at time t.

still requires a sequential check for the temporal constraint in $O((1 + c/m) \log c)$ time.

To solve Snapshot queries returning the set of active contacts (u, v, t_1, t_2) such that $t_1 \leq t < t_2$, we can binary search $[lt_s, rt_s] \leftarrow CSA\_binSearch(getmap(m_3(t))$ and $[lt_f, rt_f] \leftarrow CSA\_binSearch\ (getmap(m_4(t))$. All the contacts pointed by $A[2c + 1, rt_i]$ hold $t_s \leq t$, and those in $A[rt_f + 1, 4c]$ hold $t_2 > t$. Therefore, $\forall i \in [2c + 1, rt_s]$, if $\Psi[i] > rt_f$ we recover the source and target vertexes as $\Psi[\Psi[i]]$ and $\Psi[\Psi[\Psi[i]]]$. The original values are obtained via *getunmap()*. As before, this mechanism checks the temporal constraints sequentially. Thus, it takes at most $O((1 + c) \log c)$ time. Note that this bound is very pessimistic, since the number of contacts to check depends on the distribution of contacts through time.

In ChangedEdges, ActivatedEdges and DeactivatedEdges operations, the temporal constraint is computed straightforward by recovering the range related to the endpoints of the time interval related to each contact. For ActivatedEdges(t) the range is obtained by a binary search over the third component, $[lt_s, rt_s] \leftarrow CSA\_binSearch(getmap(m_3(t))$. Similarly, in DeactivatedEdges(t) we search for a range in the fourth component. The ChangedEdges operation is computed as the combination of both operations. Assuming that c/τ is the average number of neighboring changes in the temporal graph, the three operations can be answered in $O((1 + c/\tau) \log c)$ time.

4.4 Discussion

In this chapter we presented three different data structures based on text indexing technology. The main aim with these compressed structures is to provide a fair baseline to compare our more sophisticated proposed structures. The three structures are based on a ***log*** representation The main differences among them are how the log is created and which portion of the log is traversed to check the temporal constraint of temporal graph operations.

Both EdgeLog and EveLog are simple structures using well-known technology. They are expected to be very space efficient on temporal graphs with a large number of contacts per

edge, because d-gaps will tend to be very small. The algorithms to compute the operations using EdgeLog require to traverse the list of neighbors, and check the temporal constraint over the list of time intervals of each neighbor. In the EveLog, the algorithms traverse the list of neighboring changes related to the queried vertex, checking the parity property on each neighbor. Although the time to process DirectNeighbors is asymptotically the same for both structures (as the length of the union of the time intervals lists is exactly the number of neighboring changes of a vertex), we expect a better running time in EdgeLog, because the EveLog has to maintain the parity property with a data structure. Both structures require the transposed aggregated graph to answer ReverseNeighbors. The performance expected on vertices with a high in-degree is poor, because many inverted lists will have to be checked.

In the TG-CSA the temporal graph is represented as a sequence of 4-tuples. Each 4-tuple is an integer mapping of each contact. This mapping divides the suffix array into four sections, each of them containing a component of contacts. This produces a good compression property, because pointers Ψ of the structure tend to be growing in each section. Unfortunately, this property is also the main drawback of the representation. When there are few occurrences of the symbols in the vocabulary, Ψ will not be highly compressible, because few zones of Ψ are in increasing order (i.e., few *runs* will be generated). This can occur when there are few contacts related to each edge, or when the distribution of contacts is uniform over time.

The main advantage of TG-CSA is that it can answer queries over any component of a contact in the same way. So, it does not require the transposed aggregated graph for ReverseNeighbors. Searching for all the contacts of a source vertex u is performed exactly with the same mechanism as searching for all the contacts starting at a specific time t. Answering operations use a binary search over one of the four sectors of A, depending on the term of the contact that is searched for. For each of the entries in the area of A, the Ψ function is applied three times to recover the remaining terms of each contact (i.e., target vertex and time interval). Then, the temporal constraint is checked over each recovered contact.

In the following chapter, we improve the time efficiency of the ***log*** strategies of the EdgeLog, EveLog and the TG-CSA, by representing the temporal graph as a sequence of changes over a Wavelet Tree. We will also improve the time of ReverseNeighbors by extending the Wavelet Tree to represent multidimensional symbols.

Chapter 5

Temporal graphs based on compact sequence representations

This chapter presents a new ***log*** strategy based on compact data structures that avoids redundancy while making it possible to answer neighboring queries. We disregard here ***copy*** and ***copy+log*** strategies because they need to store the complete graph (snapshot) at several time points of the graph lifetime, even if the graph has few changes between two consecutive snapshots. Our goal is to avoid the sequential computation of the temporal constraint on contacts, and to be efficient in space.

Based on this strategy, we design two compact data structures, CAS and CET, which represent the temporal graph as a sequence of events. This sequence will be represented with a Wavelet Tree plus a bitmap, as in the compact representation of binary relations [BGIMSR07, BCN13]. The Wavelet Tree enables to check the parity property in logarithmic time, instead of traversing the log.

5.1 Compact Adjacency Sequence (CAS)

The Compact Adjacency Sequence (CAS) represents temporal graphs as a temporal adjacency log by using a similar strategy to that proposed by Navarro and Claude [Nav07, CN10b, CN10a] for compressing static graphs using adjacency lists, and to the compact representation of binary relations developed by Barbay et al. [BGIMSR07, BCN13]. This strategy concatenates all the adjacency lists into a long sequence. Then, the structure, although based on the adjacency log, is capable of computing the number of occurrences of neighboring changes in logarithmic time.

5.1.1 Sequence transformation of the adjacency log

The transformation of the adjacency log into a sequence groups the neighboring changes of each vertex by time. Each group is defined by the set of neighboring vertices $L_u^{(t)} = \{v|(v,t) \in L_u\}$, where L_u is the list of neighboring changes of the vertex u defined for EveLog (i.e. an event of activation/deactivation of the edge (u,v) at time t). This grouping reduces the size of each L_u if there exist more than one contact per time point. Using these groups, the sequence transformation of a temporal graph is the concatenation $T(\mathcal{G}) = T(u_1)T(u_2)...T(u_n)$, where $T(u) = \bar{t}_{k_1} L_u^{(t_{k_1})} \bar{t}_{k_2} L_u^{(t_{k_2})} ... \bar{t}_{k_r} L_u^{(t_{k_r})}$, where $\bar{t}_{k_i}$ is the codification of time point t_{k_i} and $L_u^{(t_{k_i})}$ are the neighboring changes of the vertex u occurred at time t_{k_i}, with $1 \leq i \leq r$. Changes are stored in ascending time order, with $t_{k_i} < t_{k_{i+1}}$.

Note that the alphabet of $T(\mathcal{G})$ is composed by $n + \tau$ symbols, with n the number of vertices and τ the number of time points in the lifetime of the graph. As an example, the CAS structure for the temporal graph in Figure 3.1 is shown in Figure 5.1.

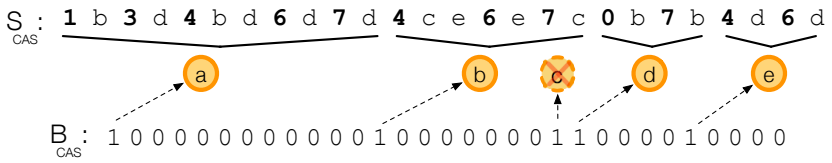

Figure 5.1: The CAS structure for the temporal graph in Figure 3.1. (The elements in dash lines are not explicitly stored in the data structure).

5.1.2 The data structure

CAS has two components: (1) a Wavelet Tree W that represents the sequence $S_{\mathsf{CAS}} = T(\mathcal{G})$ and (2) a bitmap B_{CAS} of length $(n + |S_{\mathsf{CAS}}|)$ that indicates the positions in $T(\mathcal{G})$ where the neighboring changes of different vertices occur (see Figure 5.1). The length of the neighboring changes $T(u)$ of each vertex is codified in unary coding, with as many zeros as the length of each $T(u)$. We chose a Wavelet Tree to represent the sequence S_{CAS} because it allows counting and searching over the sequence, providing primitives to obtain the symbol frequency in a range and the position of a successor, all in logarithmic time.

In this structure, the operation $start(u) = \mathsf{select}_1(B_{\mathsf{CAS}}, u) - u$ returns the position in S_{CAS} where the neighboring changes of a vertex u start. Note that the subsequence defined by $S_{\mathsf{CAS}}[start(u), start(u+1)]$ represents the neighboring changes of vertex u during the whole lifetime of the temporal graph. The operation $end(u, t) = \mathsf{rangeNextValuePos}(S_{\mathsf{CAS}}, start(u), start(u + 1), t + 1)$ returns the position in S_{CAS} of the first $t' \geqslant t + 1$ in the subsequence representing the neighboring changes of u. Here we are looking for the position of $t+1$ due to the ordering of time-point symbols in S_{CAS}.

The size of the data structure depends on the distribution of the neighboring changes through time, due to the clustering of neighboring changes by time to transform the adjacency log into a sequence. The sequence $S_{\mathsf{CAS}} = T(\mathcal{G})$ has the alphabet $\Sigma = V \cup \mathcal{T}$ $(|\Sigma| = n + \tau)$.

The maximum lengths of the sequences S_{CAS} and B_{CAS} are $4c$ and $4c + n$, respectively, which occur when there is only one contact (active edge) per time point and each subsequent contact is related to a different edge. If this is the case, each contact is represented using two symbols for neighboring vertices and two symbols for the time points that represent the time interval of the contact and, therefore, there are no contacts occurred at the same time that can be grouped in $L_u^{(t)}$.

We chose to implement the Wavelet Tree as a Wavelet Matrix [CN12, CN08], because it has a small overhead to represent the topology of the Wavelet Tree. Using a compressed bitmap representation [RRS07], the space of S_{CAS} in the Wavelet Matrix is $4cH_0(S_{\mathsf{CAS}}) + o(c)\log(n + \tau) + \log(4c)\log(n + \tau)$ bits, plus the compressed bitmap B_{CAS} using $(4c + n)H_0(B_{\mathsf{CAS}}) + o(c + n)$ bits. Best compression in CAS is achieved when the distribution of contacts per edge is non-uniform, because the zero order entropy H_0 of S_{CAS} and B_{CAS} guarantees the minimal space by assigning short codes to more frequent symbols. The worst case occurs with a uniform number of changes per edge, when the upper bound of the space becomes $4c\log(n + \tau) + O(n + c)$ bits, since $H_0(S_{\mathsf{CAS}}) \approx \log(n + \tau)$ and $H_0(B_{\mathsf{CAS}}) \approx 1$.

5.1.3 Query processing

Queries about vertices The DirectNeighbors(u,t) query is solved by getting the frequency of changes (i.e., parity property) of neighboring vertices of u. The frequency of neighboring vertices is obtained using the rangeReport operation over the Wavelet Tree representing S_{CAS}, this is, rangeReport$(S_{\mathsf{CAS}}, [start(u), end(u,t)], [1,n])$. The rangeReport operation is solved in $O((1+k)\log\sigma)$ time, where k is the number of symbols reported. Assuming a uniform degree distribution in the aggregated graph, the output size $k = m/n$, and as the alphabet in S_{CAS} is $\sigma = n+\tau$, the DirectNeighbors has a time cost of $O((1+m/n)\log(n+\tau)+m/n)$, where the last term m/n is the time used to check the parity property of each neighboring vertex. This indicates that for each possible neighbor is checked in $O(\log(n+\tau))$ time, which can be less than a sequential search in a *log* based representation. The algorithm for direct neighbors is shown in Figure 5.2.

Algorithm: DirectNeighbors(u,t) returns the set of active neighboring vertices of u at time point t using CAS.

$C =$ rangeReport$(S_{\mathsf{CAS}}, [start(u), end(u,t)], [1,n])$
$ans \leftarrow$ empty set;
foreach $(v,f) \in C$ **do**
  if f *is odd* **then** ans.add(v);
return ans;

Figure 5.2: Algorithm to answer DirectNeighbors(u,t) by using the CAS data structure.

For the DirectNeighbors$_{\mathcal{I}}(u,t,t')$ query, the algorithm obtains the active neighbors at time t and then adds or removes the edges according to the neighboring changes in the time interval $[t,t')$ and the desired semantics (i.e., strong or weak semantics), which also takes $O((1+m/n)\log(n+\tau)+m/n)$. See the algorithm in Figure 5.3 for more details.

Algorithm: DirectNeighbors$_{\mathcal{I}}(u,t,t',Q)$ returns the set of direct neighbors of u in the time interval $[t,t')$ under semantics Q using CAS.

N = DirectNeighbors(u,t) ;
$R =$ rangeReport$(S_{\mathsf{CAS}}, [end(u,t), end(u,t')], [1,n])$;
$C \leftarrow$ empty set;
foreach $(v,f) \in R$ **do**
  C.add(v) ;
$ans \leftarrow$ empty set;
if Q *is strong* **then**
  ans $= N \setminus C$;
if Q *is weak* **then**
  ans $=$ N $\cup$ C ;
return ans;

Figure 5.3: Algorithm to answer DirectNeighbors$_{\mathcal{I}}(u,t,t',Q)$ by using the CAS data structure.

For the reverse neighbors, the ReverseNeighbors(v,t) query requires to find, for any vertex u, which lists L_u contains the vertex v, and then to check if v appears an odd or even number of times until it finds the first $t' > t$. This means to get the positions of all the occurrences of v in S_{CAS}, and check what subsequence of neighboring changes $T(u)$ is related to each occurrence.

We know that the total number of occurrences of v is $\mathsf{rank}_v(S_{\mathsf{CAS}}, |S_{\mathsf{CAS}}|)$, and the position of the i-th occurrence of v in S_{CAS} can be obtained with $\mathsf{select}_v(S_{\mathsf{CAS}}, i)$. With the operation $belong(i) = \mathsf{rank}_1(B_{\mathsf{CAS}}, select_0(B_{\mathsf{CAS}}, i))$ we recover the vertex u associated with the i-th occurrence of v by using the bitmap B_{CAS} that points to the starting position of each list of neighboring changes $T(u)$. The operation $belong(i)$ has a time cost of $O(1)$, as only requires a rank/select operation in the bitmap B_{CAS}. Assuming a uniform distribution of contacts per vertex, we can obtain the lists where v appears in $O(\log(n+\tau)+c/n(\log(n+\tau)+1))$ time, where c/n is the average number of contacts per vertex, and the first term is the time required for the rank to recover the total number of occurrences of v, and the latter term is the recovering the positions of all occurrences of v using select over the Wavelet Tree and the $belong(i)$ operation. Assuming $\log(n+\tau) > 1$ and $c/n > 1$, this time can be rewrote as $O(c/n \log(n+\tau))$.

Finally, for each list where v appears, we count how many times the edge (u, v) appears until t doing $\mathsf{rangeCount}(S_{\mathsf{CAS}}, [start(u), end(u, t)], [v, v])$ in $\log(n+\tau)$ time. This schema for reverse neighbors depends on the frequency of symbol v in S_{CAS}. Assuming a uniform number of contacts per edge and a uniform degree distribution in the aggregated graph, the total time cost of $\mathsf{ReverseNeighbors}(v, t)$ sums up $O(c/n \log(n+\tau) + m/n \log(n+\tau))$.

The algorithm to compute $\mathsf{ReverseNeighbors}_{\mathcal{I}}(v, t, t')$ queries uses the same strategy than the algorithm to compute $\mathsf{ReverseNeighbors}(v, t)$ queries, but adding or removing the vertices that have changed in the time interval, according to the desired semantics. This is also computed in $O(c/n \log(n+\tau) + m/n \log(n+\tau))$. The algorithm in Figure 5.4 obtains the reverse neighbors in a time interval. Note that the $\mathsf{ReverseNeighbors}(v, t)$ query can be solved with algorithm in Figure 5.4 by using a weak semantics over the time interval $[t, t)$.

Algorithm: $\mathsf{ReverseNeighbors}_{\mathcal{I}}(v, t, t', Q)$ returns the set of reverse neighboring vertices of v in the time interval $[t, t')$ under semantics Q using CAS.

```
R ← empty set;
for i ← 1 to rank_v(S_CAS, |S|)] do
    R.add( belong(select_v(S_CAS, i)));
ans = empty set;
foreach u ∈ R do
    c = rangeCount(S_CAS, [start(u), end(u, t)], [v, v]);
    if Q is weak then
        if c is odd OR rangeCount(S_CAS, [end(u, t), end(u, t')], [v, v]) > 1 then
            ans.add(u);
    if Q is strong then
        if c is odd AND rangeCount(S_CAS, [end(u, t), end(u, t')], [v, v]) = 0 then
            ans.add(u);
return ans;
```

Figure 5.4: Algorithm to answer $\mathsf{ReverseNeighbors}_{\mathcal{I}}(v, t, t')$ by using the CAS data structure.

Queries about edges To answer the state of an edge (u, v) at a time point t (i.e., $\mathsf{Edge}(u, v, t)$), we just need to know if v appears an odd or even number of times in the interval $S_{\mathsf{CAS}}[start(u), end(u, t)]$. This can be done in $O(\log(n+\tau))$ by a rangeCount operation at the endpoints of the interval.

The $\mathsf{Edge}_{\mathcal{I}}(u, v, t, t')$ query is also obtained in $O(\log(n+\tau))$, adding a second rangeCount

between t and t' and following the appropriate semantics (i.e., weak or strong semantics). The $\mathsf{EdgeNext}(u, v, t)$ query checks if the edge is active in the time point t; if the edge is not active, the query returns the next time point when the edge will be active. As for the Edge query, the $\mathsf{EdgeNext}(u, v, t)$ query is answered in $O(\log(n+\tau))$ (see algorithm in Figure 5.5).

Algorithm: $\mathsf{EdgeNext}(u, v, t)$ returns the next activation time of an edge (u, v) after time point t using CAS.

```
if rangeCount(S_CAS, [start(u), end(u,t)], [v, v]) is odd then
    return t
else
    c = select_v(S_CAS, 1 + rank_v(S_CAS, end(u,t)))
    if c < end(u,t') then
        return rangePrevValue(S_CAS, start(u), c, t)
    else
        return ∞;
```

Figure 5.5: Algorithm to answer $\mathsf{EdgeNext}(u, v, t)$ by using the CAS data structure.

Query about the state of the graph As it occurs for the EveLog and $\mathsf{EdgeLog}$ strategies, the $\mathsf{Snapshot}(t)$ query in CAS is answered with the $\mathsf{DirectNeighbors}(u, t)$ query for all u in V.

Queries about changes/events on edges The operation $\mathsf{ChangedEdges}(t)$ is computed by retrieving the neighboring changes occurring in the subsequences $S_{\mathsf{CAS}}[end(u, t-1), end(u, t)], \forall u$. This is computed over all vertices doing a $\mathsf{rangeReport}$ operation in $O((c/\tau + n)\log(n+\tau))$ time, where c/τ is the average number of neighboring changes per time point.

The $\mathsf{ActivatedEdges}(t)$ operation is computed by using the same strategy than EveLog, obtaining the difference between $\mathsf{ChangedEdges}(t)$ and $\mathsf{Snapshot}(t-1)$. This is computed one vertex at a time, by retrieving the neighboring changes occurring in $S_{\mathsf{CAS}}[end(u, t-1), end(u, t)]$ minus the active neighbors $\mathsf{DirectNeighbors}(u, t-1)$. The same strategy is used by $\mathsf{DeactivatedEdges}(t)$, but returning the intersection between the neighboring changes in t and the direct neighbors at $t-1$. The time cost of these operations is the time to compute $\mathsf{Snapshot}$ and $\mathsf{ChangedEdges}$, that is, $O((m+n)\log(n+\tau) + (c/\tau + n)\log(n+\tau))$.

The time-interval versions of these operations are computed similarly. Algorithm in Figure 5.6 shows the $\mathsf{activatedEdges}_{\mathcal{I}}$ operation.

Algorithm: $\mathsf{activatedEdges}_{\mathcal{I}}([t, t'))$ returns the set of edges that were activated in the time interval $[t, t')$ using CAS.

```
ans = empty set;
foreach u ∈ V do
    N = DirectNeighbors(u, t − 1);
    C = rangeReport(S_CAS, [end(u,t), end(u,t')], [1, n]);
    ans.insert(C \ N) ;
return ans;
```

Figure 5.6: Algorithm to answer $\mathsf{activatedEdges}_{\mathcal{I}}(t, t')$ by using the CAS data structure.

5.2 Compact Events ordered by Time (CET)

The basic idea of CET is to represent the changes on a temporal graph as 2-tuples in a multi dimensional sequence, representing source and destination vertices of a neighboring change. The sequence used to represent the temporal graph is a time-ordered arrangement of events, that is, edges that change at the same time are stored together. Then, DirectNeighbors and ReverseNeighbors queries can be computed using range operations over the subsequences related to the time-point or time-interval of the query. For instance, the recovering of DirectNeighbors(u, t) needs to obtain the frequency of the symbols of the form $(u, *)$, where $*$ denotes any vertex, in the subsequence related with t since the first time instant of the graph.

We could use traditional multidimensional indexes to represent temporal graphs and to obtain the state of an edge (u, v) at time t by counting how many times the tuple (u, v, t') occurs with $t' \in [0, t]$. But, if we choose, for example a k-d tree[Ben75], it will require at least $O(\sqrt{c})$ comparisons to get the state of an edge. Range trees [Lue78, BCKO08] will improve the query time to $O(\log^3 c)$, but it will require a heavy penalty on space. Secondary memory indexes based on the B-tree and R-tree [Gut84] could be used, but as we have pointed before, none of them consider data compression, and indexes based on secondary memory will perform several times slower than a main-memory based representation. An alternative solution using a compact data structure is to represent each dimension of the sequence using a Wavelet Tree. This alternative solution, however, will require more operations to answer range queries, because there is no direct mapping between the elements of the different dimensions and the levels of the Wavelet Trees.

The data structure designed specifically here for temporal graphs is based on a novel variant of the Wavelet Tree, that we called Interleaved Wavelet Tree, capable of operating with sequences of d-dimensional symbols while using asymptotically the same space required by d sequences of Wavelet Trees.

5.2.1 The Interleaved Wavelet Tree

The Interleaved Wavelet Tree is a variant of the Wavelet Tree that represents a sequence $S[1, n] = s_1 s_2 \dots s_n$ of multidimensional symbols of the form $s_i = (s_{i,1}, s_{i,2}, ..., s_{i,d}) \in \Sigma_1 \times \Sigma_2 \times ... \times \Sigma_d$, where Σ_j is an alphabet $[1, \sigma_j]$[1]. With some abuse of notation we will say that $\Sigma^d = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_d$ is the multidimensional alphabet of S of size $\sigma' = \prod_{j=1}^{d} \sigma_j$. Without loss of generality, we assume $\sigma = \max_{1 \le j \le d} \sigma_j$, thus $\sigma' \le \sigma^d$. Then, S can be represented in a plain form using $nd \log \sigma + o(nd \log \sigma)$ bits.

In the Interleaved Wavelet Tree the multidimensional alphabet Σ^d is transformed into a plain alphabet $\Sigma' = [1, \sigma']$ over a Wavelet Tree, where each symbol in Σ' is the result of a transformation $\Sigma^d \to \Sigma'$ defined by interleaved bits (Morton Code [Sam06]). The interleaved codification $\bar{\sigma}'$ of a multidimensional symbol $\bar{\sigma} \in \Sigma^d$ is defined by $\bar{\sigma}' = \bar{\sigma}_{1,w}, \dots, \bar{\sigma}_{d,w}, \bar{\sigma}_{1,w-1}, \dots, \bar{\sigma}_{d,w-1}, \dots, \bar{\sigma}_{1,1}, \dots, \bar{\sigma}_{d,1}$, where $w = \log \sigma$ is the number of bits needed to codify a value in one dimension and $\bar{\sigma}_{i,j}$ is the j-th bit of the symbol in the dimension i.

The Interleaved Wavelet Tree is a balanced binary tree whose leaves are labeled with multidimensional symbols in Σ^d and whose internal nodes handle a d-range of symbols. The root node covers the d-range $[1, \sigma] \times [1, \sigma] \times \cdots \times [1, \sigma] = \Sigma^d$ and the internal nodes

[1]The alphabet is an ordered sequence of symbols that can be mapped to $\mathbb{N}$.

handle the d-range $[\mathbf{a}_v, \mathbf{b}_v] = [[a_{v1}, b_{v1}], [a_{v2}, b_{v2}], \ldots, [a_{vd}, b_{vd}]] \subseteq [1, \sigma] \times [1, \sigma] \times \cdots \times [1, \sigma]$. Each node v is a bitmap $B_v[1, n_v]$ representing the sequence $S_v[1, n_v]$ of S composed by the multidimensional symbols in $[\mathbf{a}_v, \mathbf{b}_v]$. Let $r = h \mod d$ be the modulo between the current depth h and the number of dimensions of the alphabet d. The i-th value of $B_v = [1, n_v]$ is defined as follows: if the r-component of the corresponding element s_i is less than or equal than $\frac{b_{vr}-a_{vr}}{2}$, i.e., $s_{i,r} \lfloor \frac{b_{vr}-a_{vr}}{2} \rfloor$, then $B_v[i] = 0$; otherwise, $B_v[i] = 1$. Then, a symbol $S_v[i] \in [\mathbf{a}_v, \mathbf{b}_v]$ with $B_v[i] = 0$ ($B_v[i] = 1$) will be represented in the left child (right child) of v in the next level of the tree.

Figure 5.7 shows the Interleaved Wavelet Tree for the alphabet with 2 dimensions, where $\Sigma_1 = \{0, 1, 2, 3\}$ and $\Sigma_2 = \{a, b, c, d\}$. Alphabet Σ^2 is composed by all combinations of symbols in Σ_1 and Σ_2. Each element in Σ^2 is codified into a Σ' following the interleaved bit codification; for example, symbol $1c$ is coded as $\mathbf{01}\mathbf{1}0 \in \Sigma'$, because symbol 1 in Σ_1 is coded as 01 and symbol c in Σ_2 is coded as 10. Notice that when representing a temporal graph with a set V of vertices, if source and target vertices are the same, then only a vocabulary in $\Sigma_1 = V$ will be necessary and the 2-dimensional vocabulary in Σ^2 will be composed by all possible pairs of symbols in $\Sigma^2 = V \times V$.

The height of the Interleaved Wavelet Tree is $d \log \sigma$, with $d\sigma$ leaves and $d\sigma - 1$ internal nodes. Thus, as in the Wavelet Tree, each level of the tree stores n bits, which gives an upper bound of $O(nd \log \sigma)$ bits to represent the multidimensional sequence S. The space used by the Interleaved Wavelet Tree can be reduced by using compressed representations of bit sequences [SG06, RRS07] to $O(nH_0(S))$, following the same proof given by Grossi *et al.* [GGV03, Nav12] when all symbols in Σ^d appear at least once in the sequence. We will sketch the proof considering a sequence of four symbols in a 2D alphabet. Consider that in the bitmap at root node $B_{v_{root}}$ contains n_0 zeros and n_1 ones. At this point, the zero-order entropy of $B_{v_{root}}$ is $n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1}$, with $n = n_0 + n_1$. Consider v_l and v_r be the left and right children of v_{root}. Let n_{00} and n_{01} be the number of zeros and ones in the left child bitmap B_{v_l}, and n_{10} and n_{11} the number of zeros and ones in the in the right child bitmap B_{v_r}. At this second level, the zero-order entropy of the bitmap B_{v_l} is $n_{00} \log \frac{n_0}{n_{00}} + n_{01} \log \frac{n_0}{n_{01}}$ bits. Analogously, the zero-order entropy of B_{v_r} is $n_{10} \log \frac{n_1}{n_{10}} + n_{11} \log \frac{n_1}{n_{11}}$ bits. Now, adding the space of the root node the total space is $n_{00} \log \frac{n}{n_{00}} + n_{01} \log \frac{n}{n_{01}} + n_{10} \log \frac{n}{n_{10}} + n_{11} \log \frac{n}{n_{11}}$. Indeed, this is the total zero-order entropy as the codes 00, 01, 10 and 11 could effectively represent four symbols in a two dimensional binary alphabet. By adding the space of the leaves up to the root, we get that $\sum_{c \in \Sigma'} n_c \log(n/n_c) = H_0(S)$, where n_c is the frequency of the symbol $c \in \Sigma'$.

The Interleaved Wavelet Tree representation of the tree requires $O(d\sigma)$ pointers, using $O(d\sigma \log n)$ bits, which can be wasteful for large multidimensional alphabets (the most common case for scenarios of large temporal graphs), but as in the Wavelet Tree, it can be improved by using an implicit representation such as the Wavelet Matrix [CNO15, CN12].

The definition of access, rank, and select operations of the Wavelet Tree are adapted to operate with multidimensional symbols, keeping the same semantics as in unidimensional sequences (or text). The algorithms of these operations are the same than those of the Wavelet Tree and can be answered in $O(d \log \sigma)$. Range operations over unidimensional sequences defined by [MN06] and [GNP12] have been redesigned to work with multidimensional sequences as in the original definition in [Cha88]. The range operations defined in

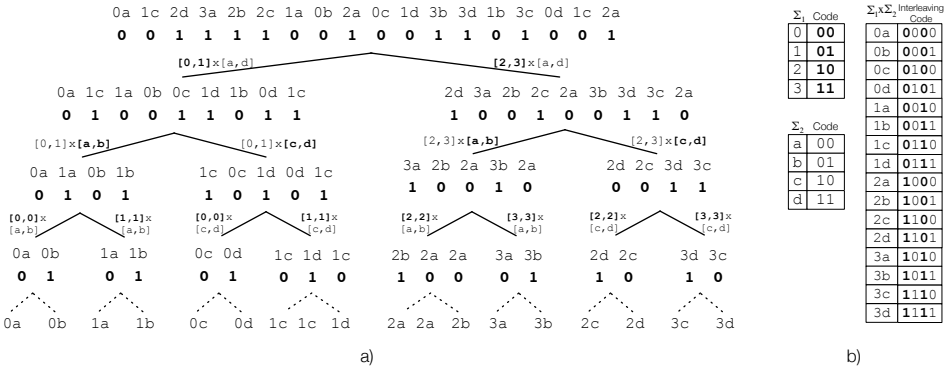

Figure 5.7: An example of an Interleaved Wavelet Tree: a) An Interleaved Wavelet Tree of the multidimensional sequence S = $\langle$0a, 1c, 2d, 3a, 2b, 2c, 1a, 0b, 2a, 0c, 1d, 3b, 3d, 1b, 3c, 0d, 1c, 2a$\rangle$. b) Alphabet of elements in S, $S[i] \in \Sigma^2 = \Sigma_1 \times \Sigma_2$. (The rightmost table contains the interleaving bits of Σ^2).

[Cha88] answer orthogonal range queries[2] in a d-range $[\mathbf{a}, \mathbf{b}]$, which is defined as the d closed intervals of each dimension $([a_1, b_1], [a_2, b_2], \ldots, [a_d, b_d]) \subseteq \Sigma^d$. In particular, let $q = [\mathbf{a}, \mathbf{b}]$ be a d-range, then

- rangeCount$(S, [i, j], q)$ returns the number of tuples in $S[i, j]$ that fall in q.
- rangeReport$(S, [i, j], q)$ returns the number of occurrences of the tuples in $S[i, j]$ that fall in q.

The algorithms to answer these operations are slightly adaptations of those proposed in [MN06, GNP12], but following the interleaved codification of each symbol. The algorithms for rangeCount and rangeReport operations take $O(d \log \sigma)$ and $O((1 + occ)d \log \sigma)$, respectively, where occ is the number of multidimensional symbols that fall in the d-range.

5.2.2 The multidimensional sequence representation

CET represents for each time instant t a sequence of symbols of an alphabet $\Sigma = V \times V$, where each 2-dimensional symbol represents an edge that changes at time t.

The set of edges that change at time t is defined by $C^{(t)} = \{(u, v) | (u, v, t_s, t_e) \in \mathcal{C}, t = t_s \vee t = t_e\}$ (i.e., additions or deletions). Therefore, CET represents temporal graphs with an Interleaved Wavelet Tree as a sequence of the form $T'(\mathcal{G}) = C^{(t_1)}C^{(t_2)}C^{(t_3)}...C^{(t_\tau)}$; that is, the concatenation of the edges that change for each time point of the temporal graph in ascending time order until the end of the lifetime.

5.2.3 The data structure

CET has two components: (1) an Interleaved Wavelet Tree storing the sequence $S_{\mathsf{CET}} = T'(\mathcal{G})$ and (2) a bitmap B_{CET} of length $(\tau + |S_{\mathsf{CET}}|)$, composed by $\tau = |\mathcal{T}|$ bits with value

[2]Orthogonal range queries are range queries whose region are axis-aligned rectangles.

1 indicating the starting position of neighboring changes in S_{CET} occurring at different time points (see Figure 5.8). The query processing on CET is based on counting over the subsequence of S_{CET} storing the neighboring changes until the requested time point. Bitmap B_{CET} is used to obtain the position where a time point t starts in S_{CET}. Note that symbols codifying neighboring changes at the same time point are coded as a 0 value in B_{CET}, and preceded by a 1. We define de function $start(t) = \mathsf{select}_1(B_{\mathsf{CET}}, t) - t + 1$. Similarly, to obtain the position where t ends, we define $end(t) = \mathsf{select}_1(B_{\mathsf{CET}}, t+1) - t$. Note that the subsequence $S_{\mathsf{CET}}[start(t), end(t)]$ contains the edges that have become active/inactive at time t.

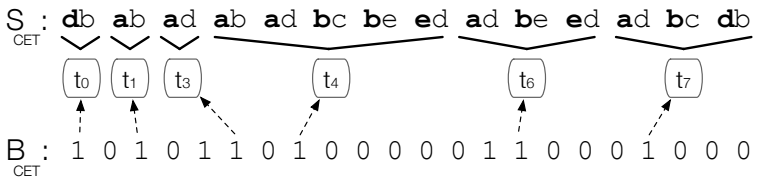

Figure 5.8: The CET for the temporal graph in Figure 3.1. The elements in dash lines are not explicitly stored in the data structure.

The sequence of symbols in the S_{CET} belongs to the alphabet $\Sigma = V \times V$, with $|\Sigma| = n^2$. The length of S_{CET} is $2c$ (i.e., twice the number of contacts) because, for each contact, there are two events: one to activate and one to deactivate an edge. As in CAS, we choose the Wavelet Matrix to implement the Interleaved Wavelet Tree[3]. Using RRR compressed bitmaps the space of S_{CET} in the Interleaved Wavelet Tree is $2cH_0(S_{\mathsf{CET}}) + o(c)\log n^2 + \log(2c)\log n^2$ bits, while the size of B_{CET} is $(2c+\tau)H_0(B_{\mathsf{CET}}) + o(c+\tau)$ bits. In the worst case, assuming a uniform number of changes per edge, the upper bound of this space is $2c\log m + O(c+\tau)$ bits, since $H_0(S_{\mathsf{CET}}) \approx \log m$ and $H_0(B_{\mathsf{CET}}) \approx 1$.

5.2.4 Query processing

Queries about vertices To obtain the direct neighbors of a vertex u at time t, we just need to define the d-range $[(u,u),(1,n)]$, where n is the number of vertices in the graph. Then, it needs to check if u appears an odd or even number of times in the subsequence $S[1, end(t)]$ and filters out those edges that appear an even number of times. Analogously, the d-range to get the reverse neighbors of a vertex v is $[(1,n),(v,v)]$. As in CET, the frequency of the vertices is obtained using the rangeReport operation. Therefore, assuming a uniform degree distribution over the aggregated graph, direct and reverse neighbors are obtained in $O((m/n+1)\log n^2 + m/n)$, where the last term is for checking the parity property. Algorithm in Figure 5.9 answers the DirectNeighbors(v,t) query, which is the same used by the ReverseNeighbors(v,t) query but replacing the d-range.

To obtain the active neighbors of a vertex in a time interval $[t,t')$, we need to get the active neighbors at time t and then, to apply the desired semantics of the pairs found in the range (t,t'). The active neighbors at time t are obtained following the Algorithm 5.9,

[3]Note here that the Wavelet Matrix become more relevant because some symbols on Σ may not appear in S_{CET}.

Algorithm: DirectNeighbors(u, t) returns the set of active neighbors of vertex u at time point t using CET.

```
d = [(u, u), (1, n)];
C = rangeReport(S_CET, [1, end(t)], d)
ans ← empty set;
foreach (u, v) ∈ C do
    if f is odd then
        ans.add(v);
return ans;
```

Figure 5.9: Algorithm to answer DirectNeighbors(v, t) by using the CET.

while the changes in the interval (t, t') are obtained through the rangeReport operation. As for time-point queries about neighbors, the d-range should be carefully chosen if we want direct or reverse neighbors. We have omitted algorithms for time-interval queries since they are very similar to the DirectNeighbors query, just adding a d-range operation to the subsequence defined by $S_{\mathsf{CET}}[start(t+1), end(t')]$ to obtain the edges that changed during the time interval of the query.

Queries about edges The edge (u, v) is active at time t if the pair (u, v) appears an odd number of times in the subsequence $S_{\mathsf{CET}}[1, end(t)]$. Then, the Edge query can be done by a rank operation over the Interleaved Wavelet Tree in $O(\log n^2 + 1)$. To obtain the edge in a time interval $[t, t')$, it is necessary to obtain the state of (u, v) at time t using the Edge operation, and then to check if (u, v) appears in the time interval (t, t'), verifying if the rank operation at $end(t+1)$ is less than the rank at $end(t')$ position. This is also obtained in $O(\log n^2 + 1)$.

To answer the EdgeNext(u, v, t) query, we first check whether or not the edge (u, v) is active at time t. If so, the algorithm returns t; otherwise, it searches the position of the next occurrence of (u, v) in the sequence S_{CET} and returns the time point associated with the neighboring change. See algorithm in Figure 5.10 for further details.

Algorithm: EdgeNext(u, v, t) returns the next activation time of edge (u, v) at time point t using CET.

```
r = rank_{u,v}(S_CET, end(t));
if r is odd then
    return t
else
    c = select_{u,v}(S_CET, 1 + r)
    if c < end(t') then
        return rank_1(B_CET, select_0(B_CET, c));
    else
        return ∞;
```

Figure 5.10: Algorithm to answer Edge(u, v, t) by using the CET.

Query about the state of the graph To answer the Snapshot(t) query, we need to modify the interval used to retrieve neighbors, setting it up to the range $[(1, n), (1, n)]$, which will recover the state of all the edges in the temporal graph.

Queries about changes/events on edges Since neighboring changes are arranged by time, operations to compute activation/deactivations of edges are efficiently computed. They just need to retrieve the symbols that appear in a subsequence. This can be done using range operations over the Interleaved Wavelet Tree. In particular, operation ChangedEdges(t) is computed by doing a rangeReport over the subsequence $S_{\mathsf{CET}}[start(t), end(t)]$.

The ActivatedEdges(t) operation is computed similarly, but with a slightly modification of the rangeReport algorithm over $S_{\mathsf{CET}}[start(t), end(t)]$. Remember that the basic idea of the rangeReport algorithm is to traverse the tree down, carrying the frequency of each symbol appearing in a subsequence $S[a, b]$. If there are symbols that belong to the left (right) child, we continue traversing down the tree, carrying the frequency of each symbol using rank operations over the endpoints of the subsequence. Once a leaf is found, we know the frequency of the symbol in the leaf at position $S[a]$ (the beginning of the subsequence), and the frequency of the symbol at position $S[b]$, which is the end of the subsequence. In the original rangeReport operation, we report the difference between those values, for all leaves that match the subsequence. In our modification of the rangeReport algorithm for the ActivatedEdges(t) operation, we return the symbols only if they occur an odd number of times before the position $start(t)$. Analogously, for the DeactivatedEdges(t) operation, we return the symbols that occur an even number of times at position $start(t)$. These operations can be computed in $O((c/\tau+1)\log n^2)$ using the Interleaved Wavelet Tree, where c/τ is the average number of contacts per time-point. Note that time-interval versions are computed using the same strategy.

5.3 Improving the representations

So far, all the proposed sequence representations consider the most general type of temporal graphs: the *interval-contact*. In this section we show how to reduce the space of the structures regarding the temporal properties found in *point-contact* and *incremental* graphs. The key idea is to reduce the length of the sequence representation by removing redundant temporal information.

We first review how to improve the space in *point-contact* temporal graphs. Recall that in *point-contact* temporal graphs all contacts are active for only one time instant, this is, contacts are of the form $(u, v, t, t+1)$. In both CAS and CET the size of the sequence can be reduced by only considering the first neighboring changes (the one that activates the edge at t), as we know that the deactivation of the edge will occur at the next time instant. With this change, the parity property is no longer required to obtain the state of an edge. Instead, we just need to retrieve the subsequence regarding the time instant that we are searching for.

The retrieval of DirectNeighbors(u, t) in CAS requires to extract the subsequence related to the vertex u and time t, which is located in $S_{\mathsf{CAS}}[end(u, t-1), end(u, t)]$. Then, by just doing rangeReport($S_{\mathsf{CAS}}, end(u, t-1), end(t), [1, n]$), and without checking the frequency of the symbols in the subsequence, we obtain the active neighbors of u at time t. Regarding time interval operations in $[t, t')$, the *strong* semantics requires to remove the symbols that appear more than once in $S_{\mathsf{CAS}}[end(u, t-1), end(u, t'-1)]$. The *weak* semantics is obtained by returning all the symbols in the subsequence, exactly as in *interval-contact* graphs. The Snapshot operation is computed by doing DirectNeighbors over all vertices, same as before. To recover the events on edges, we take into account that in *point-contact* graphs the edges

activated at time t are the same that are active at time t (see Section 3.1.2). So, they can be obtained by doing the Snapshot operation. In CET, all operations are computed similarly. For queries about vertices and edges we need to adjust the range of symbols, and recover the subsequence in $S_{\mathsf{CET}}[start(t), end(t)]$. For time interval queries over $[t, t')$, the subsequence is $S_{\mathsf{CET}}[start(t), end(t')]$, and the semantics is computed as we explained before.

Although this new representation for *point-contact* reduces the size of the sequence, it does not improve the asymtotic time cost of the operations. However, if the graph has a low number of active edges per time instant (i.e., few neighboring changes per time instant), we expect a time improvement with respect to the *interval-contact* representation, because the rangeReport operation will be performed over a small subsequence (the one regarding the time instant of the query).

Regarding the *incremental* graphs, its main characteristic is that once an edge becomes active, it remains active until the end of the lifetime. Thus, all contacts are of the form (u, v, t, τ), where $\tau = |\mathcal{T}|$ is the lifetime of the graph. Consequently, the structures does not need to store the neighboring changes occurring at τ. Thus, and as in *point-contact*, the sequences S_{CAS} and S_{CET} are reduced to the half. The algorithms to compute operations do not require any modification since, by definition, no edge is active at time τ and, therefore, the time cost is the same of the *interval-contact* representation.

The strategy for improving *incremental* graphs can be also used to improve the general representation of *interval-contact* graphs. As before, if a contact in the graph ends at the last time instant, we do not store that neighboring change. This strategy works on any *interval-contact* graph. However, its usefulness depend on how many contacts effectively end at the last time instant of the graph.

5.4 Discussion

The goal of CAS and CET structures is to compute the parity property in logarithmic time, instead of a sequential search as in the structures in Chapter 4. Both CAS and CET encode the temporal graph as a sequence of events or neighboring changes. A neighboring change is a triple (u, v, t) indicating that the edge (u, v) was activated or deactivated at time t.

The events in CAS are ordered primarily by source vertex, and secondary, by the time the events occur. This produces a sequence S_{CAS} that has n parts, each of them related to the events of a vertex. The sequence S_{CAS} is represented in a Wavelet Tree. The objectives are twofold: (1) to check the parity property in logarithmic time, and (2) compress the sequence to the first order entropy space. The DirectNeighbors operation is performed by looking the portion related to the queried vertex, and checking how many times the neighboring vertices appear through a rangeReport operation in the Wavelet Tree. The Edge operation is performed similarly, but doing a rank over the sequence.

The main issue in CAS is that the ReverseNeighbors(v, t) operation requires to check each apparition of the vertex v in the sequence. This is not efficient, because it essentially implies a sequential search, although a rank operation is used. Similarly, operations regarding events on edges (i.e., ChangedEdges) and the recovery of the state of the graph (i.e., the Snapshot operation) require to count neighboring changes over all sections of the sequence. Therefore, the time cost of these operations depends primarily on the number of vertices in the temporal graph.

In CET, the events are ordered by the time they occur. This produces a sequence

composed by τ parts, one per each time instant in the lifetime. The sequence is built over an alphabet of 2-dimensions, encoding neighboring changes as edges. We extend the Wavelet Tree to be able to handle this 2-dimensional sequence in what we call the Interleaved Wavelet Tree. As the Interleaved Wavelet Tree is based on the Wavelet Tree, the parity property is also checked in logarithmic time. Also, the sequence can be compressed by using RRR bitmaps to the zero order entropy space. The DirectNeighbors(u, t) operation is performed by retrieving the frequency of symbols of the form $(u, *)$ until the time instant t is encoded in the sequence. Similarly, the Edge operation just needs to do a rank over the sequence.

The main advantage of CET over CAS is its ability to retrieve ReverseNeighbors in the same time than DirectNeighbors. Due to the 2-dimensional representation of the Interleaved Wavelet Tree, we just need to update the range to $(*, v)$ to obtain the frequency of the edges whose target vertex is v. Another advantage is that operations of events on edges are easily obtained, just extracting the subsequence related to the time instant of the query. Regarding the Snapshot operation, a rangeReport over all the symbols is sufficient to check the parity property on all edges.

Although both methods improve the sequential time to check the temporal constraint by performing operations in logarithmic time, there is no guarantee that the compressed space of the data structures is close to the information-theoretic lower bound of representing temporal graphs. In the next chapter, we present a compact data structure based on a multidimensional k^2-tree, which is capable to achieve this lower bound.

Chapter 6

Temporal graphs based on multidimensional points

This chapter introduces a novel compact data structure that represents temporal graphs using a space close to the information-theoretic lower bound. The representation uses a 4D interpretation of contacts, similar to the representation of contacts in TG-CSA. The new structure is based on the k^d-tree [dBR14], the multidimensional version of the k^2-tree [BLN14]. As such, the k^d-tree only achieves good space when the input data is clustered, a property that does not necessarily apply to temporal graphs.

We will propose some modifications to the k^d-tree with the aim to reduce the use of space for unclustered data, while keeping good time performance. In particular, we propose two compressed data structures, ck^d-tree and bck^d-tree, based on representing temporal graphs as a whole space-time data structure (mixing nodes and time in the same representation), which is capable of recovering the state of an edge at any time without storing snapshots and without counting the number of changes.

6.1 Multidimensional compact data structures

In this section we review two already existing multidimensional compact data structures: the k^d-tree [dBÁGB$^+$13, dBR14] and the Interleaved k^2-tree [GBBN14, Gar14]. The former is d-dimensional extension of the k^2-tree to deal with multidimensional matrices, and the latter is a specialization of the k^2-tree for storing 3D binary matrices as several 2D matrices.

6.1.1 The k^d-tree

The k^d-tree [dBÁGB$^+$13, dBR14] is a generalization of the k^2-tree for representing d-dimensional binary matrices. The d-dimensional matrix of size $n_1 \times n_2 \times \ldots \times n_d$ is recursively divided into k^d submatrices. To simplify the analysis, we will assume that $n_i = n$ for all i, where each node of the tree has k^d children that represent each of the submatrices. The submatrices are numbered from 0 to $k^d - 1$, following a row-major order[1]. This means that the first dimension is contiguous for the first k^{d-1} submatrices (i.e., submatrices are ordered by the highest dimension, then the second, and so on). The codification of the tree into the bitmaps T and L follows the same strategy: a 1 bit if the submatrix is non-empty, and a 0 otherwise. Figure 6.1 shows an example of k^d-trees with $k = 2$.

The navigation of the tree is analogous to the one in the k^2-tree, the first children of a 1 bit at the position p in T is found at the position $p' = rank_1(T, p) \times k^d$ in $T|L$ (the concatenation of bitmaps T and L). To check if a cell $c = (c_1, c_2, c_3, ..., c_d)$ exists in the input matrix, we need to check which child node represents the submatrix where c falls. Starting from the root node, this requires to compute for each dimension the value $v_i = \frac{c_i}{n/k}$, where n is the size of the current submatrix. Then, the j-th submatrix, with $j = \sum_{i=1}^{d} v_i \times k^{(i-1)}$, contains

[1] Other orders have been proposed, but they do not make any improvement on the space or the navigation time [dBR14].

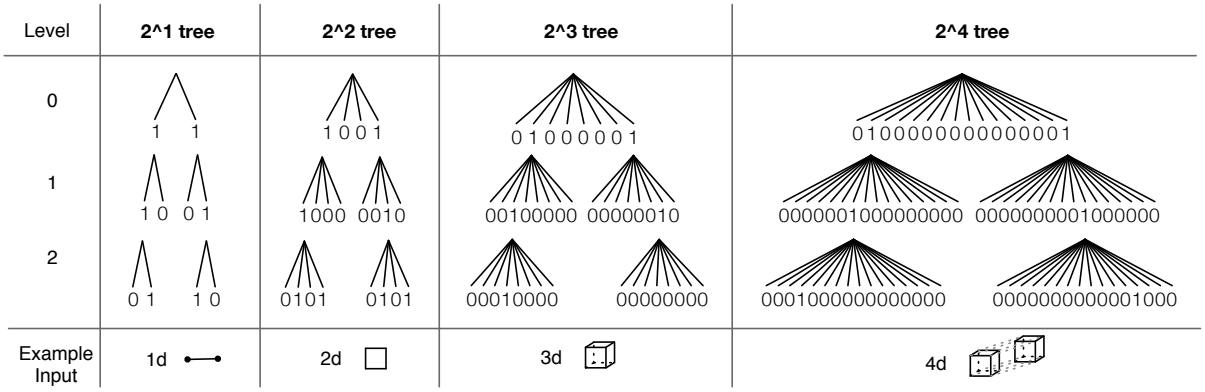

Figure 6.1: Example of k$^{\mathsf{d}}$-trees for 1, 2, 3 and 4 dimensions with $k = 2$. We include an example of the input: points in a line for 1D, cells in a square matrix for 2D, cells in a cube matrix 3D, and cells in a tesseract for 4D.

the cell. If the j-th bit is set to 1, it traverses down the k$^{\mathsf{d}}$-tree using the rank operation, but updating the c_i by $c'_i = c_i \mod n/k$ to reflect the new position with respect to the smaller submatrix in the j-child. This procedure continues until we find a leaf node or an empty submatrix. The navigation time to check the existence of a cell is $O(\log_{k^d} n^d) = O(\log_k n)$, which corresponds to the height of the tree $h = \lceil \log_k n \rceil$. Other range operations depend on how many components are fixed with a value, and how many submatrices should be retrieved at each level.

Following the same space analysis of the k^2-tree in [BLN14], in the worst-case for a d-dimensional binary matrix of size n^d with m 1s uniformly distributed, the space used by the k$^{\mathsf{d}}$-tree is $k^d m \log_{k^d} \frac{n^d}{m} + O(k^d m)$ bits. For $k = 2$, the space achieves its minimum, with $2^d m \log_{2^d} \frac{n^d}{m} + O(2^d m)$ bits. This equation reveals one of the issues with k$^{\mathsf{d}}$-tree, its space increases exponentially with the number of dimensions.

As the k$^{\mathsf{d}}$-tree is based on the k^2-tree, the same optimization techniques of the k^2-tree can be applied to reduce its space and time performance, just adjusting the navigation pattern.

6.1.2 The Interleaved k^2-tree

The Interleaved k^2-tree [GBBN14, Gar14] is a specialization of the k^2-tree specially designed to deal with 3D data. It is useful to represent ternary relations with a skewed distribution in one of its dimensions. The main idea is to partition the ternary relations into m sets of binary relations, where m is the number of different values in the skewed dimension. Then, the ternary relations can be represented by m different binary relations, each of them represented with a different k^2-tree.

The partition works as follows: Let W be a set containing triples of the form $(x, y, z) \in X \times Y \times Z$. Assume that the set Z has a lower cardinality than X and Y, and that $|Z|$ is the number of different items in Z. Then, the partition is defined as a set of $|Z|$ different binary relations of the form $Z_i = \{(x, y) | (x, y, z_i) \in W\}$. Finally, each of the Z_i relations is represented by a binary matrix in a k^2-tree.

The Interleaved k^2-tree (ik^2-tree) corresponds to the merge of the $|Z|$ different k^2-trees

in one structure. The idea is to *interleave* the bits representing the same branches of the $|Z|$ k^2-trees. As in the k^2-tree, each node has k^2 children, each of them representing a submatrix with a variable number of bits. Each bit represents one item z_i of the skewed dimension. At the first level, each of the k^2 nodes contains $|Z|$ bits, one per item in Z. The i-th bit of a node is set to 1 if the Z_i matrix contains at least one cell in the corresponding submatrix of the node; otherwise, the bit is set to 0.

In the following levels, the number of bits of each internal node corresponds to the number of ones in its parent node. For example, if a parent node has m ones, each of its k^2 children will contain m bits. The number of ones in the parent node indicates the number of items in Z that have a cell in its corresponding submatrix. Following the example, the number of ones in the parent node indicates that there are m different R_i matrices with a cell in the submatrix related to the parent node. Figure 6.2 shows an example of a ik^2-tree with $k = 2$ and three z_i values.

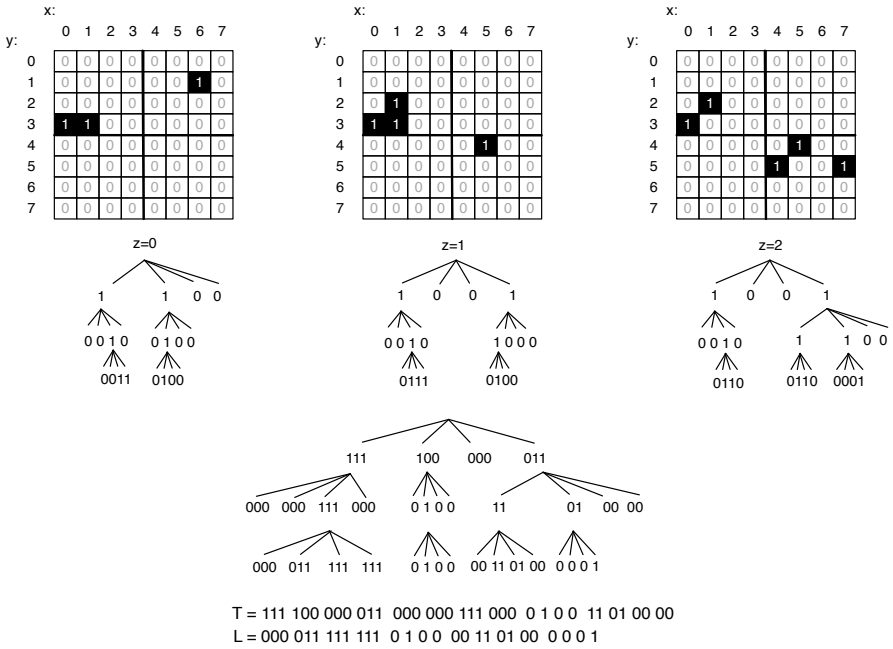

Figure 6.2: An ik^2-tree representation of three binary matrices. Each binary matrix is encoded as a k^2-tree, which represents triples with a fixed value for the z component. The ik^2-tree is composed by the interleaved bits of the same branches of the three k^2-tree.

The tree is also stored in two bit arrays: T stores all levels except the last one, which is stored in the bitmap L. The structural properties of the original k^2-tree hold for the ik^2-tree, a 1 bit in the level l will generate k^2 bits at level $l + 1$. However, as each node has a variable number of bits, and the number of bits of each node depends on the active bits of the parent, the navigation strategy must be updated accordingly. Because we know that each of the k^2 nodes in the first level contains $|Z|$ bits, we must adjust the navigation by skipping the first $|Z| \times k^2$ bits.

Assume a node starting at position p in T, with $m > 0$ active bits. Then, the first child

begins at position $p' = (\mathsf{rank}_1(T, p) + |Z|) \times k^2$ and has a size of m bits in $T|L$. The number of active bits of the first child is $m' = \mathsf{rank}_1(T, p' + m) - \mathsf{rank}_1(T, p')$. Notice that the factor $|Z| \times k^2$ skips the nodes in the first level.

Queries in the ik^2-tree can be divided into queries retrieving tuples with a fixed value in the Z component, and queries retrieving tuples within a range value in the Z component. The formers only require to traverse down the tree by checking a specific bit of each node. For example, to retrieve all cells with the value z_i, we start from the k^2 children of the root. In each child we verify if the z_i bit is active. If it is active, we traverse down the tree using the rank operation until the leaves. If the bit is inactive, it means that the current submatrix is empty for z_i. Queries retrieving a range value in the Z component require more work, because we need to check all bits of the nodes involved in the query. The idea is to perform many queries by fixing the value of the third component. For example, if we want to recover all tuples with the Z component between the values z_i and z_{i+w}, we need to perform a fixed query for the values z_i, z_{i+1}, ..., z_{i+w}.

The space used by the ik^2-tree corresponds to the combined space of the $|Z|$ different k^2-trees. To achieve less space and navigation time, we can directly apply all the enhancements designed for the k^2-tree. The ik^2-tree has been applied to *RDF* triples and temporal graphs [GBBN14, Gar14]. In some cases, like evolving raster data, the k^d-tree outperforms in both space and time the ik^2-tree [dBR14].

ik^2-trees for temporal graphs As temporal graphs are binary relations evolving over time, they can be represented as triples of the form (u, v, t_k), where t_k indicates the time instant when the edge (u, v) has been activated or deactivated. Indeed, each contact of the graph (u, v, t_i, t_j) generates two triples (u, v, t_i) and (u, v, t_j), corresponding to the time points when the edge (u, v) is activated and deactivated, respectively. The Interleaved k^2-tree [GBBN14, Gar14, dBR14] uses the idea of this 3D encoding to store temporal graphs. The triples are indexed by the temporal component.

By using the ik^2-tree, the state of an edge (u, v) at time t_k is active if there is an odd number of triples (u, v, t_m), where $t_m \in [0, t_k]$. The retrieval of direct and reverse neighbors and snapshot queries work in the same way, by counting how many triples are related to each neighbor. This representation is very compact because the state change is only stored once. However, it requires to count each neighboring change to recover the state of an edge, which can be expensive on edges with many contacts.

In the following sections we will reveal how to deal with the exponential increasing of the space when using the k^d-tree and how to deal with the sparseness due the d-dimensional space.

6.2 The Compressed k^d-tree (ck^d-tree)

A trivial approach to represent a temporal graph is to use a k^4-tree where contacts are cells in a 4-dimensional binary matrix, with two dimensions encoding edges and the others two encoding time intervals. This simple representation has the problem referred as *curse of dimensionality* [Sam06], which indicates that when the number of dimensions increases, the available data becomes sparse.

A k^4-tree has nodes with 16 bits (for $k = 2$), because there are 2^4 submatrices where 4D points can fall in each recursive space partition (i.e., space increases *exponentially* in the

number of dimensions). When a k^4-tree represents a temporal graph, the time constraint imposes that cells encoding values $t_s \geqslant t_e$ will never be used and, in consequence, the maximum number of contacts stored in the same leaf is equal to 4. Therefore, the 4D points to represent contacts become very sparse, producing many unary paths where internal nodes tend to have leaves storing a single cell. Therefore, the direct use of k^4-trees for representing temporal graphs does not compress as well as the k^2-tree does for static graphs because the self-similarity mechanism (on the paths) used by the k^2-tree cannot be replicated for temporal graphs.

Without considering any kind of regularity (i.e., self-similarity), the information-theoretic lower bound on the number of bits needed to represent a 4D matrix is the logarithm of the number of possible matrices of size n^4, with m active cells. We call entropy to this value, and it is expressed by $\mathcal{H} = \log \binom{n^4}{m}$. Using the Stirling's approximation, one can get (see Lemma 8 in [Pag99] and Section 2.1 in [BM99]):

$$\begin{aligned}\mathcal{H} = \log \binom{n^4}{m} &= \log n^4! - \log m! - \log(n^4 - m)! \\ &\approx n^4 \log n^4 - m \log m - (n^4 - m) \log(n^4 - m)\end{aligned}$$

Rewriting $\log(n^4 - m)$ as $\log n^4 - \log(1 - m/n^4)$, we get:

$$\mathcal{H} \approx n^4 \log \frac{n^4}{m} + (m - n) \log(1 - m/n^4) \tag{6.1}$$

Considering that we are dealing with a sparse matrix, we can assume that $m = o(n^4)$. Then, the second term in Eq. 6.1 is bounded by $O(m)$, thus:

$$\mathcal{H} = \log \binom{n^4}{m} \leq m \log \frac{n^4}{m} + O(m) \tag{6.2}$$

The k^4-tree representing the binary matrix (for $k = 2$) requires $16m \log_{16} \frac{n^4}{m} + O(16m) = 4m \log \frac{n^4}{m} + O(m)$ bits, which is, asymptotically, four times the entropy $\mathcal{H}$.

How can one achieve more compression taking into account the sparseness of the 4-dimensional space? We propose in this thesis a variant of the k^d-tree that is able to compress unary paths on leaves representing one contact (a 1 cell) using the "entropy" $\mathcal{H}$ space. The idea behind our proposal is to stop the decomposition of the d-dimensional binary matrix when a submatrix with only one cell (contact) is found. This produces three kinds of nodes in the k^d-tree: *white* leaf nodes representing an empty submatrix, *black* leaf nodes representing a submatrix with only one cell (representing only one contact), and *gray* internal nodes representing a submatrix with many 1 cells (many contacts). The isolated cell in a black leaf is stored as a relative position with respect to its submatrix in a separated array.

Unlike the k^2-tree (and the k^d-tree), where the whole tree itself encodes active cells, we conveniently choose which portion of the root-to-leaf paths are encoded in the tree (as a *gray* node), and which part is encoded outside the tree (an isolated cell in a *black* leaf).

Conceptually, the proposed strategy resembles the idea of the Point Region (PR) Quadtree [Sam06, pp. 42-46]. The main difference with the PR Quadtree is that we are reducing the total space used by the whole data structure, by encoding the active cells as root-to-leaf

paths over a succinct representation of the tree, were pointers are replaced by bitmaps. The succinct representation of the tree follows the generalized Jacobson's level-order [Jac89a] [BDM$^+$05] encoding for cardinal trees. We extend this representation to encode each kind of node (white, black and grey) using only one extra bit per node. We also take advantage of the level-order encoding to retrieve the path of a black leaf in constant time.

As the ck$^\mathsf{d}$-tree relies on the k$^\mathsf{d}$-tree, the tree can be traversed using minor modifications of the original operations until a black node is found or the fixed depth is reached.

6.2.1 Encoding the tree

The first step to compress the k$^\mathsf{d}$-tree is to develop a strategy for encoding the three kinds of nodes in the tree: white, black and gray nodes. Bits in T are 1 when the corresponding submatrices have one or more cells, corresponding to black leaves or gray nodes, respectively, and 0 for empty submatrices (white leaves) as in the original k$^\mathsf{d}$-tree. To differentiate if the 1 bit belongs to a black leaf or gray node, we create a second bitmap B that stores one bit for each 1 in T. Black leaves are marked with 1, while gray nodes are marked with 0. Note that the size of B is the number of 1s in T. This encoding schema is based on the work developed by de Bernardo *et al.* [dBÁGB$^+$13] to compress submatrices full of ones in a k^2-tree.

The navigational mechanism on the new k$^\mathsf{d}$-tree must be updated to take into account the new codification in B of black and gray nodes. Now, the first children of an internal node at position p in T will start at position $p' = (\mathsf{rank}_1(T,p) - \mathsf{rank}_1(B, \mathsf{rank}_1(T,p)) \times k^d$, because we need to subtract the number of 1 bits in T that encode black leaves until the current position p, as they do not generate new children. If a position p in T is set to 1, $B[\mathsf{rank}_1(T,p)]$ can take two values: 1 if p corresponds to a black leaf, or 0 if p corresponds to a gray internal node.

6.2.2 Encoding isolated cells in black leaves

The unary path of an isolated cell (i.e., a black leaf) is represented as a relative position of the cell with respect to the upper-left corner of its corresponding submatrix. If the upper-left corner is located at position $(u_1, u_2, ..., u_d)$ of the matrix, the isolated cell $(p_1, p_2, ..., p_d)$ is stored as $(u_1 - p_1, u_2 - p_2, ..., u_d - p_d)$. For example, the cell $(9, 1)$ in Figure 6.3 is stored as $(1, 1)$, because the upper-left corner of its submatrix starts at position $(8, 0)$. The relative positions of the cells are stored in level order, as black leaves appear in the tree, into a d-dimensional array A. Each entry corresponds to a black leaf in the ck$^\mathsf{d}$-tree. The unary path of a black node at position p is stored at the position $A[\mathsf{rank}_1(B, \mathsf{rank}_1(T,p))]$. Figure 6.3 shows the compressed version of the k^2-tree in Figure 2.4.

Note here that black leaves at the last level of the tree do not require to store the relative position with respect to its submatrix. This because, at the last level, the submatrices have a size of k^d, and the relative positions are already encoded by the position of the 1 bits in T. Therefore, the bitmap B and the array A have no entries for black leaves occurring at the last level of the ck$^\mathsf{d}$-tree.

The unary paths can be stored using $d \times \log(n/k)$ bits per black leaf, where n/k corresponds to the size of the largest submatrix that can be stored in the tree. But this option is naive, because the size of the submatrices depends on the level where the black leaves occur.

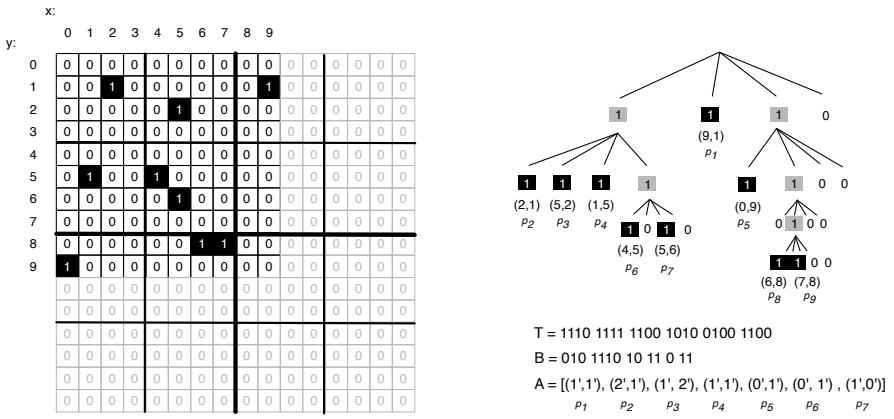

Figure 6.3: A compressed version of the k^2-tree in Figure 2.4 with a binary matrix of size $n = 10$ and $k = 2$.

Another alternative is to use a variable-length encoding such as DACs [BLN13], but there is no guarantee that space will be the minimum required by a uniform distribution of isolated cells. Because the depth of the black leaf indicates the size of the submatrix and, therefore, the maximum relative positions of isolated cells, we could reduce the space by partitioning A into A_l different arrays, one per level of the k$^{\mathsf{d}}$-tree of height $h = \lceil \log_{k^d} n^d \rceil$. Then, a black leaf at level l uses $d \times \log(n/k^{l+1})$ bits. This guarantees an improvement of the space with respect to the k$^{\mathsf{d}}$-tree. In the k$^{\mathsf{d}}$-tree the isolated cell is a unary path stored as a leaf in the last level, using $k^d \times \log(n/k^{l+1})$ bits in total.

The partition of A requires to be aware of the level of the node that we are visiting. In order to simplify this, we also split the bitmaps T and B by level and the navigation of the tree is updated accordingly. The first children of an internal node at position p at level l is found at position $p' = (\mathsf{rank}_1(T_l, p-1) - \mathsf{rank}_1(B_l, \mathsf{rank}_1(T_l, p-1))) \times k^d$ at level $l+1$. On the other hand, to check if p at level l is a black leaf or a gray internal node, we verify if $B_l[\mathsf{rank}_1(T_l, p)]$ is 1 (or 0, respectively). The unary path of a black node at position p in T_l is found at position $A_l[\mathsf{rank}_1(B_l, \mathsf{rank}_1(T_l, p))]$.

6.2.3 Space analysis

Assume a d-dimensional binary matrix of size n^d, storing m cells uniformly distributed. Each cell is a 1 in the binary matrix and, in the worst case, this requires to store a node for each level of the tree, requiring a total of $h = \lceil \log_{k^d} n^d \rceil$ nodes. As each internal gray node requires to store k^d bits in the bitmap T, and another k^d bits in B, this induces a total space of $2k^d m \lceil \log_{k^d} n^d \rceil$ bits. However, not all nodes can be different in the upper levels of the tree. In the worst case, all nodes exist up to level $h' = \lfloor \log_{k^d} m \rfloor$ (that level contains m different nodes). From that level, the worst case is that each of the m paths to leaves is unique and, consequently, they are stored as a black leaf. Each black leaf at this level is storing a cell as an offset with respect to a submatrix of size $n^d/(k^d)^{h'}$. This offset is, indeed, the path to the leaf of the traditional k$^{\mathsf{d}}$-tree. The total space to store all offsets of

cells in black leaves is $m \log(n^d/m)$ bits. Thus, in the worst case, the total space in bits is:

$$2 \sum_{l=1}^{\lfloor \log_{k^d} m \rfloor} (k^d)^l + m \log \frac{n^d}{m} = m \log \frac{n^d}{m} + O(k^d m).$$

As in the k$^{\mathsf{d}}$-tree, the formula suggests that a smaller k will achieve less space. It also depends exponentially on the number of dimensions, but only in one of their terms. For $k = 2$ and $d = 4$, the space is $m \log \frac{n^2}{m} + O(16m) = m \log \frac{n^4}{m} + O(m)$ bits, which is asymptotically the information-theoretic minimum space (Eq 6.2) necessary to represent all binary matrices of size n^4 with m 1s (i.e., m contacts).

6.2.4 Construction

The construction algorithm for the ck$^{\mathsf{d}}$-tree is based on the *inplace-construction* algorithm of the k$^{\mathsf{2}}$-tree and k$^{\mathsf{d}}$-tree [BLN14, p. 158] [dBR14]. It is based on an iterative selection of the cells that are active for each submatrix in the tree. With this strategy, we can build the bitmaps T and B, and the arrays A by level from left to right. We assume that the input binary matrix is represented as an array $P[1, m]$ of m points of the form $(p_1, p_2, ..., p_d)$, with p_i indicating the position of the cell in each dimension. Notice that the maximum depth of the tree is $h = \log_{k^d} n^d = \log_k n$. Therefore, at the beginning of the construction algorithm, there are h available empty bitmaps T_l and B_l, as well as, h arrays A_l. Recall that A_l is a d-dimensional array, holding $d \times \log n/k^{l+1}$ bits per entry.

The algorithm works by maintaining a queue of subproblems to be resolved, each of them corresponding to a submatrix and a node in the tree. A subproblem is composed by the size of the submatrix, the level l in the tree of the current node, and the interval $[a, b]$ of the array P, which contains the points that fall into the submatrix. The initial step is to enqueue the subproblem representing the root node at level $l = 0$ and the interval $[1, m]$ representing the whole matrix of size n. The following steps are based on setting the bits in T and B, and enqueue new subproblems, one for each child of the node. When the interval $[a, b]$ contains only one cell, we mark the bitmap B and set the array A encoding the corresponding unary path.

For each subproblem, we generate at most k^d new subproblems, each of them corresponds to the submatrix of a child node at level $l + 1$. A subproblem at level l with an interval $[a, b]$ for a submatrix of size n is processed as follows. We subdivide the interval $[a, b]$ into k^d subintervals and assign to each point in $[a, b]$ a key i, with $0 \leq i < k^d$. The key is defined by

$$i = \sum_{j=1}^{d} \left(\frac{p_j}{n/k} \mod k \right) \times k^{j-1}, \tag{6.3}$$

where the denominator n/k is the size of the submatrices of the child nodes.

As in the k$^{\mathsf{2}}$-tree, we sort the interval by the key using counting sort, generating k^d subintervals $[a_i, b_i]$ of $[a, b]$. Then, we append to the bitmap T_l a 1 bit if $a_i < b_i$, and a 0 otherwise. For each 1 in T_l, we append a 1 in bitmap B_l if the size of the interval is $b_i - a_i = 1$ (i.e., it is a black leaf), and a 0 otherwise. For each 1 in B_l, we append the codification of the unary path of the point in $P[a_i]$. The unary path of $p = P[a_i]$ is calculated as a relative position of the cell with respect to the current submatrix as (p_0

mod $n/k, p_1 \mod n/k, ..., p_d \mod n/k$). For each 0 in B_l (a non-black leaf), we enqueue a new subproblem with the interval $[a_i, b_i]$ with a submatrix size of n/k at level $l+1$. See Figure 6.4 for the complete algorithm.

In the worst case, when the matrix is full of ones, the construction size requires to keep at most m items in the queue. In the average case, assuming a uniform distribution of the m cells, the construction time is $O(m \log_{k^d} m)$, because for each level of the tree we need the counting sort for each interval.

Algorithm: construct($P[1,m], n, k, d$) builds the ck$^{\mathsf{d}}$-tree with the set of d-dimensional points in P.

Output: The bitmaps T_l and B_l and the array A_l by level.

```
Q.enqueue(⟨1, m, 0⟩);
while Q is not empty do
    ⟨a, b, l⟩ = Q.dequeue();
    K[a,b] = ComputeKeys(P[a,b], l,n,k);                    /* Array of keys in P[a,b] */
    I[0,k^d-1] = CountingSort(P[a,b],K[a,b]);   /* The interval holding each key in P[a,b] */
    for i = 0 to k^d − 1 do
        a_i, b_i = I[i];
        if a_i = b_i then                                          /* case 1: white node */
            T_l.append(0);
        else if b_i − a_i = 1 then                                 /* case 2: black leaf */
            T_l.append(1); B_l.append(1);
            A_l.append(Path(P[a_i])) ;
        else if b_i − a_i > 1 then                                  /* case 3: gray node */
            T_l.append(1); B_l.append(0);
            for j = 0 to k^d − 1 do
                Q.enqueue(⟨a_i, b_i, l + 1⟩)
```

Figure 6.4: Algorithm for constructing the ck$^{\mathsf{d}}$-tree. The output is the bitmap T_l, B_l, and the arrays A_l encoding the unary paths. Function ComputeKeys returns the key for each cell $(p_1, p_2, ..., p_d)$ in $P[a,b]$ as in equation 6.3.

6.2.5 Orthogonal range search

The algorithm for range search in the ck$^{\mathsf{d}}$-tree is similar to the one in PR quadtrees, traversing down all the child nodes whose submatrices intersect with the region to retrieve. The region is also defined by two extreme cells, the upper-left and lower-right cells. As we are not storing explicitly the boundaries (upper-left and lower-right cells) of the submatrices, we calculate them as we traverse down the tree.

The search works as follows. We start from the root node representing the whole matrix, this is the region between the upper-left cell at $(0, 0, ..., 0)$ and the lower-right cell at $(n, n, ..., n)$. Then, we recursively traverse down all the k^d children following the rank operations defined in the last section. Let $(u_1, u_2, ..., u_d)$ be the upper-left cell of a submatrix of size n. The j-th component of the lower-left cell of the i-th child submatrix is defined by $u'_j = u_j + (n/k) \times ((i/k^j) \mod k)$, and the lower-right cell by $(u'_0 + n/k, u'_1 + n/k, ..., u'_d + n/k)$. We stop the recursion if the boundary of the submatrix does not intersect with the region, or if the current node is a black leaf. In the case that we reach the last level of the tree, we return $(u_0, u_1, ..., u_d)$ because the upper-left cell corresponds to a 1^d submatrix, i.e., a cell.

Figure 6.5 presents the algorithm for the orthogonal range search, which retrieves the

active cells in a region R. The algorithm is invoked with parameters $\mathsf{range}(l, n, u, z = 0, R)$, where l is the level of the tree to traverse, n is the size of the current submatrix, u is is the d-dimensional position of the upper-left cell of the submatrix, z is the position of the current node in T_l, and R is the query region. As the root node is virtual in the bitmap codification of the tree, we set $l = -1$ to represent the level of the root node, u is $(0, 0, ..., 0) \in n^d$, and $z = 0$.

Algorithm: $\mathsf{range}(l, n, u, R, z)$ returns the set of active cells in region R.

Output: Cells inside the region defined by R.

```
if region(u, u + n) ∩ region(B) is empty then return;
if T_l[z] = 1 then                                          /* Black leaf or Gray node */
    if l = depth-1 then                                     /* Black leaf at last level */
        output u;
    else if B_l[rank(T_l, z)] = 1 then                      /* Black leaf */
        p = rank(B_l, rank(T_l, z));
        output A_l[p];
    if l = -1 then                                          /* Root node */
        z' = 0;
    else
        z' = (rank(T_l, z − 1) − rank(B_l, rank(T_l, z − 1))) × k^d;
                                                /* Searching in all children submatrices */
    for i = 0 to k^d − 1 do
        for j = 0 to d − 1 do
            u'_j = u_j + n/k × (i/k^j  mod k)
        range(l + 1, n/k, u', R, z' + i)
```

Figure 6.5: Algorithm for orthogonal range search in the ck$^\mathsf{d}$-tree.

Note that by using range search, one can compute operations over temporal graphs in similar way than the k^2-tree does for static graphs. The next section shows how to obtain these ranges and gives an upper bound of the time cost required to obtain some operations.

6.3 Compressed k$^\mathsf{d}$-tree for temporal graphs

As we said at the beginning of last section, contacts of a temporal graph can be represented by cells in a 4D binary matrix, two dimensions for representing the edges and two dimensions for representing the time interval when the edge is active. Therefore, the entropy of a temporal graph $\mathcal{G} = (V, E, \mathcal{T}, \mathcal{C})$ represented in a 4D binary matrix can be expressed as:

$$\mathcal{H} = \log \binom{n^2 \times \frac{\tau(\tau-1)}{2}}{c} \leq c \log \frac{n^2 \times t^2}{c} + O(c), \tag{6.4}$$

where $c = |\mathcal{C}|$ is the number of contacts (4D cells), $n = |V|$ is the number of vertices, and $\frac{\tau(\tau-1)}{2}$ is the maximum number of different time intervals in lifetime of size $\tau = |\mathcal{T}|$.

Refinements of this representation depend on the type of the temporal graph. For example, we know that in a *point-contact* all contacts last for only one instant, thus, they can be represented as 3D cells, where the time interval is replaced by the time point when the edge is active. In the same way, *incremental* (*decremental*) temporal graphs can be represented

by 3D cells, because we know that the time intervals end at the end of the lifetime (or start at the beginning of the lifetime). In this case, the third dimension is storing the time point when the contacts start (or end).

In this section we show how to obtain the active neighbors by performing a range search over the matrix. This is the same mechanism used by the k^2-tree to retrieve direct and reverse neighbors.

6.3.1 Operations as range search

Because we are storing contacts in 4D binary matrices, we can compute the adjacency operations of temporal graphs by doing orthogonal range searches. Indeed, this is the rationale behind the successor (predecessor) algorithms of the k^2-tree used to compute direct (reverse) neighbors in static graphs. In that case, they recover the active cells in a row (column).

For *interval-contact* temporal graphs, the idea is to recover the active cells inside a 4D region. For example, to obtain the active direct neighbors of vertex u at the time point t, i.e., DirectNeighbors(u, t), we need to recover the contacts $(u, \cdot, t_s, t_e)$, with the second component unbounded and with the temporal constraint such that $t_s \leq t < t_s$. This temporal constraint can be translated into a range over the third and fourth component, such that $t_s \in [0, t]$ and $t_e \in (t, \tau)$. This range defines the region between the cells $(u, 0, 0, t{+}1)$ and $(u{+}1, n, t{+}1, \tau)$. Observe that we are fixing the first component, and the second component is represented as a whole range $[0, n)$ in the second dimension (representing target vertices).

The same idea can be extended to recover direct neighbors for *point-contact* temporal graphs. In this case, we are converting a contact of the form $(u, v, t, t + 1)$ into a 3D cell of the form (u, v, t). Then, DirectNeighbors(u, t) just requires to recover the cells in the range $(u, 0, t)$ and $(u + 1, n, t + 1)$, because we know that contacts only last one time-point. For *incremental* temporal graphs with contacts (u, v, t, τ), where τ is the graph's lifetime, contacts can be also stored as 3D cells. In this case, the cell is of the form (u, v, t). As we know that all contacts end at the last time-point, direct neighbors can be recovered by the range $(u, 0, 0)$ and $(u + 1, n, t + 1)$. Operations for *decremental* temporal graphs can be derived in the same way.

Interval queries over vertices and edges require to manage the *weak* and *strong* semantics. The *weak* semantics retrieves all the contacts overlapping the interval query over $[t, t')$, this is, all contacts such that $[t_s, t_e) \cap [t, t') \neq \varnothing$. The constraint is equivalent to recovering contacts such that $t \leq t_e$ and $t_s \leq t'$. These inequalities define the corresponding range over the third and fourth components as $t_s \in [0, t')$ and $t_e \in (t, \tau)$, respectively. Then, weak DirectNeighbors operation over an interval can be computed by retrieving the cells inside the region $(u, 0, 0, t + 1)$ and $(u + 1, n, t', \tau)$.

As the *weak* semantics retrieves the overlapping contacts with respect to the query interval, there can be duplicated edges in the output. We removed these duplicated items by adding an extra step, sorting the target vertices of each edge[2]. This extra step is not required in CAS and CET [CARB15], because they already return non-duplicated edges.

The *strong* semantics retrieves all the active contacts during the $[t, t')$, this is, it retrieves

[2]When the input is small, the sorting method is faster than creating a hash table to remove duplicated items.

Operation	Interval-contact (4D)	Point-contact (3D)	Incremental (3D)
Edge$((u,v),t)$	$(u,v,0,t+1)$ $(u+1,v+1,t+1,\tau)$	(u,v,t) $(u+1,v+1,t+1)$	$(u,v,0)$ $(u+1,v+1,t+1)$
DirectNeighbors(u,t)	$(u,0,0,t+1)$ $(u+1,n,t+1,\tau)$	$(u,0,t)$ $(u+1,n,t+1)$	$(u,0,0)$ $(u+1,n,t+1)$
Weak DirectNeighbors$(u,[t,t'))$	$(u,0,0,t+1)$ $(u+1,n,t',\tau)$	$(u,0,t)$ $(u+1,n,t')$	DirectNeighbors(u,t')
Strong DirectNeighbors$(u,[t,t'))$	$(u,0,0,t')$ $(u+1,n,t+1,\tau)$	-	DirectNeighbors(u,t)
Snapshot(t)	$(0,0,0,t+1)$ $(n,n,t+1,\tau)$	$(0,0,t)$ $(n,n,t+1)$	$(0,0,0)$ $(n,n,t+1)$
ActivatedEdges(t)	$(0,0,t,0)$ $(n,n,t+1,\tau)$	Snapshot(t)	$(0,0,t)$ $(n,n,t+1)$
ActivatedEdges$([t,t'))$	$(0,0,t,0)$ (n,n,t',τ)	$(0,0,t)$ (n,n,t')	$(0,0,t)$ (n,n,t')
DeactivatedEdges(t)	$(0,0,0,t)$ $(n,n,\tau,t+1)$	Snapshot$(t-1)$	-
DeactivatedEdges$([t,t'))$	$(0,0,0,t)$ (n,n,τ,t')	$(0,0,t-1)$ $(n,n,t'-1)$	-

Table 6.1: Application of orthogonal range search to compute temporal graphs operations. The search range is defined by the region between the upper-left and the lower-right cells in the first and the second row of each operation. Ranges are provided for Interval-contact, Point-contact and Incremental temporal graphs. We show operations that are equivalent to others.

all contacts such that $[t,t') \subseteq [t_s,t_e)$. Therefore, the range for the third and fourth components are $t_s \in [0,t)$ and $t_e \in [t',\tau)$, respectively. The strong DirectNeighbors operation is computed by retrieving the cells inside the region $(u,0,0,t')$ and $(u+1,n,t+1,\tau)$.

The EdgeNext operation is computed by retrieving the first contact found in the output of the Edge operation over the interval $[t,\tau)$, considering the weak semantics. As the interval in a *weak* semantics contains the contacts of the edges that are active at time t (until the end of the lifetime), the first contact in the output is the next activation of the edge.

Table 6.1 shows the upper-left and lower-right cells defining the boundaries to compute operations for Interval-Contact, Point-Contact and Incremental temporal graphs. The ReverseNeighbors operation can be computed as the DirectNeighbors operation by updating the unbounded range to the first component. The DeactivatedEdges operation can be computed by updating the time constraint to the fourth component.

6.3.2 Time analysis

The time to compute the operations over temporal graphs depends on how many components are fixed in the search range. The search range used to recover direct neighbors (or reverse neighbors) fixes one component. Thus, the worst-case scenario is to traverse down k^3 submatrices per node until the leaves, where cells are of the form $(u-1,\cdot,\cdot,\cdot)$. As we reported in the space analysis in Section 6.2.3, the depth of the tree for m 1s uniformly distributed in a 4D matrix is $h = \log_{k^4} c$. Thus, in the worst case, this gives us an upper bound of $(k^3)^h \in O(c^{3/4})$. This, indeed, is not the ideal $O(m/n)$ (average active neighbors for any time point, with m the number of edges), but it is better than checking the state of

all active cells in $O(c)$.

Because the Edge operation fixes two components in the range search (the source and target vertex), its upper bound is $(k^2)^h \in O(\sqrt{c})$. As a Snapshot query does not fix any component, its upper bound is $(k^4)^h \in O(c)$. Note that, because the contacts satisfy the *temporal constraint* of the time interval (third and fourth components), the average performance is, indeed, better than the time under a uniform distribution of ones. Operations retrieving events on a time instant fix the third or the fourth component of each contact. Thus, its upper bound time is $O(m^{3/4})$.

The same analysis can be followed for 3D representations. As the *incremental* temporal graphs fix one component for DirectNeighbors (ReverseNeighbors) operations, the worst case upper bound is $(k^2)^h \in O(c^{2/3})$. The Edge operation fixes two components, which gives $k^h \in O(c^{1/3})$. The Snapshot operation is still $O(c)$. Operations for *point-contact* temporal graphs traverse down less matrices per node. The DirectNeighbors (ReverseNeighbors) operation fixes two components and runs in $(k)^h \in O(c^{1/3})$. The Edge operation only traverses down through one submatrix, and this is done in $O(\log_{k^3} c)$. The Snapshot operation is computed in $(k^2)^h \in O(c^{2/3})$ because it fixes the time component. The operations regarding events are also computed in $O(c^{2/3})$, as they only fixes the time component of each triple.

6.3.3 Hybrid representation of *interval-contact* graphs

In real temporal graphs, such as the Web, a great percent of links between pages remain active for long periods of time, while others do not. Consider, for example, a newspaper website that shows in its homepage a menu linking to different sections (e.g., sports and politics), and a body that points to the articles of the day. As we can see, the homepage shares properties of an *incremental* graph for the menu, because links remain active for long periods of time; and shares properties of an *interval-contact* graph for the body, which is constantly updated. We propose here to take into account this type of cases by partitioning contacts of a temporal graph into two sets of contacts represented by 4D and 3D tuples. In this way, we take advantage of the space reduction of the 3D representation of contacts following an *incremental* or *point-contact* graph.

The partition works by splitting contacts into three groups that satisfy properties of *incremental*, *point-contact* or *interval-contact* graphs. If the contact ends at the end of the lifetime graph, it belongs to the *incremental* class, and if the duration of the contact is a time point, it belongs to the *point-contact* class, both contacts stored as 3D tuples. Otherwise, the contact belongs to the *interval-contact* class and it is stored as a 4D tuple. To answer temporal graph operations, it is necessary to perform the range search over both, the 4D and 3D representations, and combine the answers. The combination step is straightforward, except for interval operations. Using a *weak* semantics, we need to remove duplicated edges, while using a *strong* semantics, we need to delete the edges that appear twice or more times. This step is necessary to ensure the *strong* semantics constraint. Although this may suggest that operations require twice the time of the original structure in the worst case (as we perform the range search over two data structures), it works very well in practice, as we will show in the experimental section. Notice that this improvement only works when a great percentage of contacts belongs to the *incremental* or *point-contact* class.

6.4 Improving the Compressed k^d-tree

In this section we provide two techniques to enhance the performance of the ck^d-tree. The first one, referred as node compression, compresses nodes that have more than one leaf but with few direct children. This is done by encoding half of the dimensions as a new parent node, and the other half of dimensions as children of the parent node. The second strategy, referred as black-leaves buckets, improves time by grouping black leaves into buckets of a fixed size. Although the improvement techniques of the k^2-tree seen in Section 2.4.1 are sensible, we must recall that they only work due to specific characteristics found in Web (static) graphs and raster data.

6.4.1 Node compression / Dimensional partition

With a non-uniform distribution of data in d dimensions, most nodes of the k^d-tree tend to have few children. For example, in a k^4-tree representing a 4 dimensional matrix (with $k = 2$), nodes are represented by $2^4 = 16$ bits, even if they have only one child. Thus, most of the nodes in the bitmap T store many 0s. Our proposal is to diminish even more the space of the k^d-tree by reducing the arity of internal nodes (representing the sparse ones using less bits). This can be done by breaking down the assumption that children in a k^d-tree must represent exactly one of the k^d multidimensional submatrices and by allowing that internal (grey) nodes represent submatrices in fewer dimensions. In this way, a submatrix represented by an internal node using k^d bits can be redefined by a new node with at most $k^{\lceil d/2 \rceil}$ children, using $k^{\lfloor d/2 \rfloor}$ bits each. This parent node represents $k^{\lceil d/2 \rceil}$ submatrices in $\lceil d/2 \rceil$ dimensions. The child nodes represent $k^{\lfloor d/2 \rfloor}$ submatrices in the remaining $\lfloor d/2 \rfloor$ dimensions. This allows a space reduction in nodes with few children because less bits are necessary to represent empty submatrices.

To compress an internal node v, we require to divide its binary representation in $k^{\lceil d/2 \rceil}$ blocks. We create a new node v_p with $k^{\lceil d/2 \rceil}$ children. If the j-th block of v is non-emtpy (i.e., it has at least one child), we set the j-th child of v_p pointing to the j-th block. Therefore, the space can be reduced if many blocks are empty. For example, a node in a k^4-tree with $k = 2$ and only one child can be transformed into a new parent node with a child, both using k^2 bits each. This allows us to reduce the space up to 50% of the original node. In a 3D matrix, the space can be reduced up to 75%, because nodes using $2^3 = 8$ bits can be converted into a parent node of $2^2 = 4$ bits (to store the information of the first two dimensions) and a child node of 2^1 bits to store the last dimension. Due to the level order used to represent the tree in the T_l bitmaps, this node compression strategy works only in nodes of an entire level of the k^d-tree. Figure 6.6 shows the compressed version of a sparse internal node with one child.

The tree navigation algorithms must be updated accordingly to be aware of the split used to partition the dimensions of each submatrix.

When the node is sparse, there is also an improvement in time because less bits are traversed to check which one is pointing to a child. For instance, consider a node in a 4D space, with only one child (Figure 6.6). In the traditional representation (Figure 6.6a), a node uses 16 bits, thus, we need to check each of the 16 bits to see which one of its children exists. In contrast, when using node compression (Figure 6.6b), we only need to check 8 bits to find which one is the active child. Thus, while node compression reduces the space,

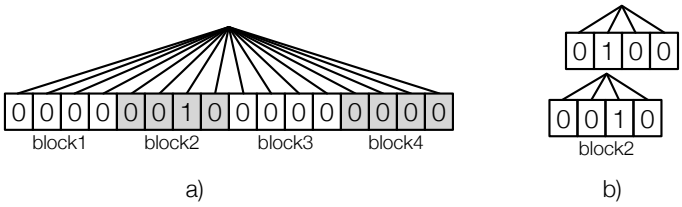

Figure 6.6: Compression for a sparse node with few children: a) A sparse node in a k^4-tree with $k = 2$ and one child. b) The compressed version of the node in a).

it also reduces the traverse time because the tree has less nodes.

Non-uniform matrices with more than two dimensions get best results. This is because in two dimensions, with $k = 2$, a node will use 2^2 bits, which is exactly the same space of a parent with a child using 2^1 bits each. Hence, neither space or time gain is expected for $d = 2$.

When data is uniformly distributed (i.e., the worst-case scenario), the node compression technique increases the space of the data structure. This is because each internal node has all its k^d children. Thus, internal nodes will require $k^d + k^{d/2}$ bits, as all blocks in node compression will be used. This also increases the expected height of the tree to two times the height h' of the ck^d-tree without node compression, because $\log_{k^{d/2}} m = 2h'$. Therefore, the space required for bitmaps T and B and the time to traverse the tree will increase in this case. In practice, however, temporal graphs are non-uniform. In the experimental section we will show that both space and time are improved in 4D representations using this technique.

6.4.2 Bucket black-leaves

As Samet claims in [Sam06, p. 45], when data is clustered (i.e., not uniform) the PR Quadtree may contain many empty nodes. Thus, the tree becomes unbalanced, which also happens in the ck^d-tree. To overcome this unbalance, Samet aggregated leaves into buckets, proposing the Bucket PR Quadtree. Each bucket can store b cells, where b is the bucket capacity [MHN84]. We follow the same strategy proposed by Samet and defined the Bucket ck^d-tree (bck^d-tree), whose black leaves encode at most b cells. The main goal of this method is to speed up the time to retrieve data in sparse submatrices with a non-uniform distribution of cells. This structure requires to modify the construction of bitmaps T_l and B_l by stopping the recursive decomposition until we find b or less cells in the current submatrix. The cells are also stored in the array A_l in level order (from left to right) using also an offset with respect to the submatrix.

Because buckets can store b or less cells, we need to indicate how many cells are stored in each bucket. We create a new bitmap C_l, one per each level, storing the size of the bucket in each black leaf in unary coding.If the current submatrix has $k \leq b$ cells, we append $k - 1$ zeros followed by a 1 bit in C_l. This is done from left to right, for all black leaves found at level l, and for all levels. The first cell of the black leaf at position p in T_l is found at position $p' = select_1(C_l, rank_1(B_l, rank_1(T_l, p)))$ in A_l, and the last one at position $p'' = select_1(C_l, 1 + rank_1(B_l, rank_1(T_l, p)))$ in A_l. The total length of the bitmap C_l is m, because each cell is encoded with a 1 (if the bucket only holds one cell), or with a

0 otherwise. Figure 6.7 shows the bck^d-tree of the ck^d-tree in Figure 6.3.

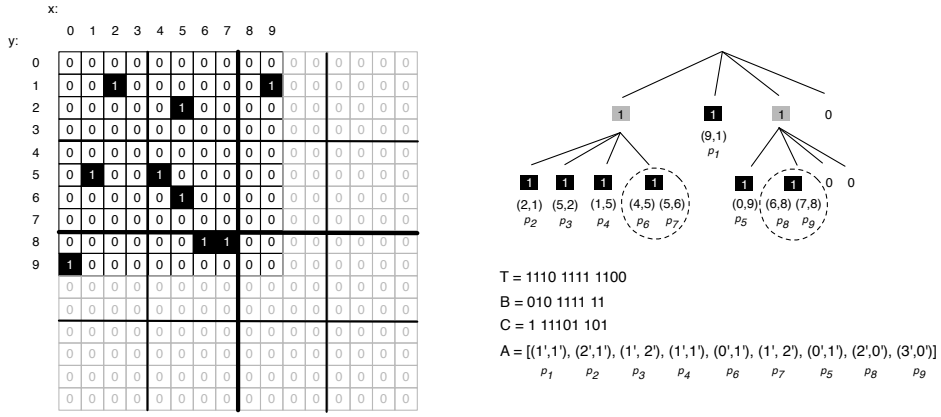

Figure 6.7: The bucket version of the ck^d-tree in Figure 6.3. Black leaves encode at most 2 cells (bucket size of $b = 2$). Two circles mark black leaves that were merged into a bucket.

By grouping nodes into buckets we expect to reduce the retrieval times, because the average depth of the tree decreases with the size of the bucket. In a uniform matrix, the expected depth of the tree is $h'' = \lfloor \log_{k^d}(m/b) \rfloor$, as each black leaf will hold at most b cells. This induces a space-time tradeoff because, with a shorter tree, less space is used in bitmaps T_l and B_l, but more space is required in the arrays A_l and extra m bits are used in the C_l bitmap. The space in A_l increases because it depends on the level l where black leaves are. As more entries will be stored in the first levels of the tree, more space will be used (i.e., the space increases with b). Regarding the time performance, once we find a black leaf, we need to check each of the b entries in the bucket to find a single cell. In the experimental section we will evaluate the trade-off using three different bucket sizes.

6.5 Discussion

In this chapter we proposed the ck^d-tree, a novel compact data structure for storing multi-dimensional binary matrices. The main aim of the ck^d-tree is to improve the space used by the k^d-tree when the cells in the matrix do not show clustering properties. In the worst-case scenario (i.e., uniform distribution of 1s in the matrix), the ck^d-tree uses a space close to the information-theoretic lower bound for representing all the matrices of size n^2t^2 with c 1s.

We use the ck^d-tree for representing the contacts of temporal graphs as a cell in a 4D binary matrix. The main advantage of this strategy is that temporal operations are run as a orthogonal range search on the ck^d-tree, by fixing some components of the range. This gave us, in most operations, a time in $O(c^\epsilon)$, with $\epsilon \leq 1$, where c is the number of contacts in the graph. For instance, for operations over vertices, we fix one component (the source or target vertex), which gives us a time cost of $O(c^{3/4})$. Although this is not the ideal, we are able to avoid the sequential search over a temporal log for checking the state of an edge.

We also propose two improvements to ck^d-tree that reduce space and time costs by considering sparse nodes (i.e., nodes with few children) and grouping nearby leaves into buckets.

Chapter 7

Evaluation

This chapter presents the experimental evaluation of the compressed data structures presented in the thesis. Three sections compose this chapter. The first section presents an analytical comparison of the structures presented in Chapters 4, 5, and 6. Then, in the following two sections, experiments compare the space and time performance of all the structures by using several real and synthetic temporal graphs.

We consider the ik^2-tree [GBBN14, Gar14] and the Snapshot k^2-tree as the baselines under the *log* and *copy* strategies, respectively. For the ik^2-tree, we generate the corresponding ternary relations representing the neighboring changes of each contact. For the Snapshot k^2-tree (denoted as Snap.k^2-tree), we generate the snapshot of active edges in temporal graphs with a short lifetime. In addition, we consider the k$^\mathsf{d}$-tree as the baseline for the ck$^\mathsf{d}$-tree. The k$^\mathsf{d}$-tree represents contacts of graphs with 4D and 3D tuples as described in Section 6.3.

7.1 Analytical comparation

Table 7.1 summarizes the theoretical upper bounds of space and time for adjacency operations, assuming a uniform degree distribution in the aggregated graph, and a uniform distribution of contacts per vertex and per edge. We omitted the compressed reverse aggregated graph for EdgeLog and EveLog, and some terms in $O(n+m+\tau+c)$, as they only represent a small fraction of the final space. Recall that $n=|V|$ is the number of vertices, $m=|E|$ is the number of edges, $c=|\mathcal{C}|$ is the number of contacts, and $\tau=|\mathcal{T}|$ is the lifetime of the temporal graph. Then, the average in/out-degree of a vertex is m/n and, in consequence, m/n is the expected size of the output for the direct- and reverse-neighbor queries.

	ck$^\mathsf{d}$-tree	CAS	CET	EveLog	EdgeLog	TG-CSA
Space (bits)	$O(c\log\frac{n^2\tau^2}{c})$	$O(c\log(n+\tau))$	$O(c\log m)$	$O(c\log\frac{n^2\tau}{2c})$	$O(m\log\frac{n^2}{m}+c\log\frac{m\tau}{2c})$	$O(c\log(n+\tau))$
DirectNeighbors	$O(c^{3/4})$	$O(m/n\log(n+\tau)+m/n)$	$O(m/n\log n^2+m/n)$	$O(c/n+m/n)$	$O(c/n+m/n)$	$O(c/n\log c)$
ReverseNeighbors		$O((c/n+m/n)\log(n+\tau))$		$O(m/n+mc/n^2)$	$O(m/n+c/n+m^2/n^2)$	
Edge	$O(\sqrt{c})$	$O(\log(n+\tau))$	$O(\log n^2)$	$O(c/n)$	$O(c/m+m/n)$	$O(c/m\log c)$
Snapshot	$O(c)$	$O((m+n)\log(n+\tau)+m)$	$O(m\log n^2+m)$	$O(c+m)$	$O(c+m)$	$O(c\log c)$
ChangedEdges	$O(c^{3/4})$	$O((c/\tau+n)\log(n+\tau))$	$O(c/\tau\log n^2)$	$O(c+m)$	$O(c+m)$	$O(c/\tau\log c)$

Table 7.1: Space (in bits, omitting $O(n+\tau+m+c)$ in general) and time costs for adjacency operations. Space is given in terms of a temporal graph with a uniform degree distribution of the aggregated graph, and a uniform number of contacts per vertex and per edge. We omitted the space for storing the inverted aggregated graph of EveLog and EdgeLog. Operations ActivatedEdges and DeactivatedEdges have the same performance as ChangedEdges. Value $n=|V|$ is the number of vertices, $m=|E|$ is the number of edges, $\tau=|\mathcal{T}|$ is the lifetime of the temporal graph, and $c=|\mathcal{C}|$ is the number of contacts. We omitted binary search time cost in EveLog and EdgeLog, because temporal logs decompression is linear.

In CAS and CET, most of the space depends on the size of the alphabet used in the sequence that codifies the neighboring changes. In CAS, the alphabet depends on the number n of vertices plus the number τ of time points in the lifetime of the graph, because it combines symbols to represent neighboring vertices and time of changes. In CET, the size of the alphabet is $m \leq n^2$ because the sequence stores the edges associated with neighboring changes. Note that when the lifetime is large enough (i.e., $\tau \gg c$), CAS uses less space than CET because τ increases the size of B_{CET} faster than the $\log(n+\tau)$ factor of S_{CAS}. The space of the TG-CSA also depends on the alphabet of the sequence. However, it is twice the alphabet of CAS. Thus, it uses more space than CAS and CET.

For the basic log-based representations, EdgeLog achieves less space than EveLog because while EdgeLog adds only a new time point in the temporal log per each new contact of an edge, EveLog may need to add not only a time point but also a vertex in the adjacency log per each new contact. Notice that in both structures the reverse aggregated graph would be necessary in order to improve retrieval times for ReverseNeighbors.

The space of the ck$^{\mathsf{d}}$-tree is the information-theoretic lower bound for storing c contacts in a 4D matrix. The size of the matrix is n^2t^2, because it needs to cover the maximum number of edges and the maximum number of time intervals. Thus, the final space depends on the number of vertices and the length of the lifetime.

Assuming a length of the lifetime equal or less than the number of vertices (i.e., $\tau \approx n$) and a small number of contacts per edge (i.e., $c \approx m$), the ck$^{\mathsf{d}}$-tree is the best structure. Regarding the sequence-based structures, the space of CAS and CET is similar to the one of TG-CSA, but less than the space of EveLog and EdgeLog. This is because the space for representing pointers to the adjacency log and pointers to the temporal log in EveLog and EdgeLog, respectively, is larger than the space for the implicit representation of the topology of the tree in the Wavelet Matrix of CAS and CET.

Operations on all data structures require to check the state of edges, either retrieving neighboring nodes or getting a snapshot, or doing a range operation. EveLog, EdgeLog, and TG-CSA also require to check the historical activation of each edge, which can be expensive for a long history. Regarding the ck$^{\mathsf{d}}$-tree, operations take $O(c^{\epsilon})$, with $\epsilon \leq 1$. This is due to the number of components fixed in the orthogonal range search used to recover active neighbors.

The CAS and CET structures show a logarithmic behavior in terms of the number of vertices and the number of time points of the temporal graph for Edge and DirectNeighbors queries. The EveLog and EdgeLog structures show the worst results computing DirectNeighbors queries in $O(c/m+m/n)$ (i.e., number of contacts per edge plus the out-degree in the aggregated graph) due to the linear traversing of the adjacency log of EveLog and the decompression of the temporal log of each edge in the EdgeLog. Notice that for graphs with a small number of contacts per vertex, the linear traversing of EveLog and EdgeLog can be as fast as the logarithmic time of CAS and CET to compute DirectNeighbors queries. This is because the compression techniques used by EveLog to process Edge queries are cache friendly (PForDelta [ZHNB06]), which outperforms the logarithmic time of the Wavelet Tree in short lists. The Edge queries on the EveLog structure is not very fast because it needs to check all the changes per vertex. The EdgeLog structure, in contrast, needs to check all the time points when the edge changes its state. Note, however, that this can be fast for short lists due to the cache-friendly design of PForDelta.

The structures CET, TG-CSA and ck$^{\mathsf{d}}$-tree have the same time cost for both DirectNeighbors

and ReverseNeighbors queries. Both CAS and EveLog need to check all contacts related to the vertex for the ReverseNeighbors query. The EdgeLog achieves better behavior than EveLog because it only checks the possible set of active edges, but again, it needs to recover and check all the time points when the edges change their activation state (as in the Edge query).

Overall, the semantics of time-interval queries does not affect the time to compute a query, because all edges or neighboring vertices must be checked.

The Snapshot operation in CAS, EveLog, and EdgeLog is computed with several applications of DirectNeighbors over all vertices. Thus, as in DirectNeighbors, the best performance is obtained in CAS, as it does not require to traverse the temporal log. In CET, the operation is computed through a rangeReport operation over the Wavelet Matrix. Its time performance is logarithmic with respect to the total number of edges in the graph. The range used to obtain the Snapshot operation in ck$^{\mathsf{d}}$-tree does not fix any component, thus, in the worst case it requires to check all contacts. Similarly, in the worst case, the TG-CSA needs to check the temporal constraint of the query on the time intervals of all contacts.

Regarding the operations that retrieve activations, deactivations, or state changes of edges at a time-point or a time-interval, CET is the fastest structure. This is because the sequence S_{CET} is organized by grouping the neighboring changes by time. Therefore, a simple rangeReport operation can be used to retrieve what happens at a time point or during a time interval. The approach followed by TG-CSA is similar, as contacts starting or ending at the same time instant are grouped in the same section of the sequence. As organization of the log of EveLog and CAS is by vertex, computing these operations requires to check each vertex, even if there is no neighboring changes at the desired query time. Similarly, the EdgeLog needs to check the temporal log for each edge. In the ck$^{\mathsf{d}}$-tree, the retrieval of events on edges fixes the third or fourth component, thus its time performance is in $O(c^{3/4})$.

In summary, there is a trade-off between space and time when choosing a data structure for temporal graphs. Indeed, while some structures win in space, others win in time (and vice versa), depending on the characteristics of the temporal graph and the type of query.

7.2 Experimental setting

We ran several experiments with real and synthetic temporal graphs. Table 7.2 gives the main characteristics of these graphs: name, type, number of vertices, edges and contacts, length of the lifetime, number of contacts per edge, space of a plain EdgeLog representation (4-byte and 8-byte integers for pointers), and the space of the entropy $\mathcal{H}$ defined in Section 6.2.

The Comm.Net is a synthetic dataset simulating short communications between random vertices. The Powerlaw dataset is also synthetic; it simulates a power-law degree graph, where few vertices have many more connections than the other vertices, but with a short lifetime. The Flickr-Day and Flickr-Sec datasets are incremental temporal graphs, both encoding the time when persons became friends in the Flickr social network. The Flickr-Day [CMG09] dataset uses a granularity by *day* with a lifetime of 134 days, from 2006-11-02 to 2007-05-18 [1]. The Flickr-Second dataset captures the creation of a friendship with granularity by *second*, since the creation of the social network. The Wiki-Links dataset is composed of the history of links between articles of the English version of Wikipedia. It has

[1] Available at `http://socialnetworks.mpi-sws.org/data-www2009.html`.

a time granularity by *second*. It corresponds to the history dump of Wikipedia[2] of 2014-03-04. Wiki-Edit [Kun13] is a point-contact temporal graph, indicating the time when a user edits a Wikipedia article[3]. Time is stored in *seconds* since the creation of Wikipedia. The Yahoo-Netflow dataset contains communication records between end users in the Internet and Yahoo! servers [Lab14]. Finally, the Yahoo-Session dataset is a point-contact temporal graph. It contains the time when a user searched a set of query terms in the Yahoo! search engine.

Dataset	Type	Vertices (n)	Edges (m)	Lifetime (τ)	Contacts (c)	c/m	Size	$\mathcal{H}$
I-Comm.Net	Interval	10,000	15,940,743	10,001	19,061,571	1.20	389	64
I-Powerlaw	Interval	1,000,000	31,979,927	1,001	32,280,816	1.01	750	136
I-Wiki-Links	Interval	22,608,064	564,224,135	414,347,809	731,468,598	1.30	14,535	6,724
I-Yahoo-Netflow	Interval	103,661,224	321,011,861	114,193	955,033,901	3.21	14,339	6,543
G-Flickr-Days	Increm.	2,585,570	33,140,018	135	33,140,018	1.00	798	127
G-Flickr-Secs	Increm.	6,204,134	71,345,977	167,943,898	71,345,977	1.00	1,728	630
P-Wiki-Edit	Point	21,504,192	122,075,170	304,002,801	266,720,840	2.18	4,226	2,465
P-Yahoo-Session	Point	171,340,122	311,277,761	1,209,601	907,128,116	2.91	14,285	7,116

Table 7.2: Description of temporal graphs used in the experimental evaluation. Column Size denotes the space of the structure using a plain representation of the EdgeLog, while column $\mathcal{H}$ denotes the information-theoretic lower bound of the space to store a temporal graph. Both Size and $\mathcal{H}$ are given in MB.

The number of dimensions of the binary matrix encoding the temporal graphs depends on the dynamic properties of the dataset. We used 4D matrices for interval (I) temporal graphs, and 3D matrices for incremental (G) and point-contact (P) graphs.

The time performance was measured at a query level (i.e., measuring the time that took to run a neighboring operation). We divided the evaluation by class of operation suggested in Section 3.1.1 (i.e., by operations about edges, vertices, state of the graph and events along time). For each operation class, we run its time instant and time interval, considering the *weak* and the *strong* semantics. Depending on the type of operation we evaluate the average time per query or the average time per output contact (i.e., a direct/reverse neighbor, or an active edge), details are explained in the section of each operation class.

The experiments ran on a machine with two quad-core processors Intel Xeon CPU E5620@2.4 Ghz, and 64GB DDR3 RAM at 1067MHz. The operating system was Ubuntu GNU/Linux 12.04 and the compiler was the GCC 4.8.3 with -O3 compile optimization.

Implementation details The key-value data structure used to check the parity property in EveLog is the hash table (unsorted_map of C++11) whose insertion takes $O(1)$ on average. To compress the sequence of neighboring changes we used a C implementation of ETDC[4]. Both, EveLog and EdgeLog use a C++ implementation of PForDelta from the PolyIRTK project[5]. The smallest average space was obtained with PForDelta block size of 128 for EveLog and 32 for EdgeLog (experiments were conducted on all the datasets). When the number of elements to compress is less than the block size, we replaced PForDelta by the Golomb-Rice code [Gol66].

[2]Downloaded from `http://dumps.wikimedia.org/enwiki/`.

[3]Downloaded from `http://konect.uni-koblenz.de/`.

[4]Available at `https://github.com/diegocaro/etdc/`.

[5]Available at `http://code.google.com/p/poly-ir-toolkit/`.

We implemented the Wavelet Tree and the Interleaved Wavelet Tree as a Wavelet Matrix [CN12]. We used the RRR compressed bitmaps [RRS07, CN08] in CAS and CET structures from the Compact Data Structure Library (libcds) [6]. We used the Succinct Data Structure Library (sdsl-lite) [GBMP14b] to create the bitmaps in the k^d-tree, ck^d-tree and bck^d-tree, and libcds for managing the arbitrary width arrays in ck^d-tree and bck^d-tree.

For the k^2-tree, ik^2-tree, and TG-CSA we used the implementation gently provided by their authors[7].

7.3 Space evaluation

This section shows the space evaluation for all the structures. It first includes a subsection to evaluate the enhancements proposed for the ck^d-tree and the bck^d-tree.

7.3.1 Sensitivity to node compression and bucket size

Before comparing space and time against the baselines, we report how the node compression and the size of buckets affect the representation of temporal graphs using 3D and 4D matrices. We evaluated seven different configurations of the ck^d-tree and bck^d-tree for *I-Wiki-Links* and *P-Wiki-Edit* graphs: WC is the plain version, without node compression; HC and FC use node compression for Half and Full levels of the tree, respectively; B16 and B64 use a bucket size of 16 and 64 cells[8], respectively, without node compression; and B16 FC and B64 FC use a bucket size of 16 and 64 with full node compression, respectively. We considered the space usage by the tree and black leaves (bitmaps T_l and B_l), the encoding of the isolated cells (array A), and the space to store the bucket size of each black leaf (bitmaps C_l). All the bitmaps in ck^d-tree and bck^d-tree used the interleaved bit-vector of the sdsl-lite with a block size of 1024 bits. We also report here the time to retrieve direct neighbors in ms per output contact.

Figure 7.1 shows the benefits of using node compression and buckets of black leaves in 3D and 4D matrices. Comparing WC against HC and FC, we can observe that effectively the space is reduced when more levels of the tree use node compression. This is due to the shrinking of the bitmap T in both 3D and 4D matrices. In 4D matrices, the improvement of space also produces an improvement in time, achieving best results when all levels of the tree use node compression. Conversely, the reduction of space in 3D matrices is at the expenses of increasing the retrieval time. Comparing B16 and B64 configurations, we observe a slightly increment in space, but an important improvement of time with respect to WC, HC, and FC variants. In 4D matrices the retrieval time is improved 3.7 times over the FC configuration, and 1.5 times over the WC variant. The improvement is directly related to the bucket size, with better time using a larger block size.

When we combine the bucket variant with node compression (B16 FC and B64 FC), we do not obtain the space reduction of node compression. This is because the bitmap T is already small when using B16 and B64 configurations. However, in 4D matrices we inherit

[6] Available at `https://github.com/fclaude/libcds`.

[7] The structures were implemented by Susana Ladra (k^2-tree), Guillermo de Bernardo and Sandra Álvarez (ik^2-tree) and Antonio Fariña (TG-CSA).

[8] In this section we will use B with a suffix to denote the size of the bucket in the bck^d-tree, and b (without a suffix) to denote the block size in bitmaps.

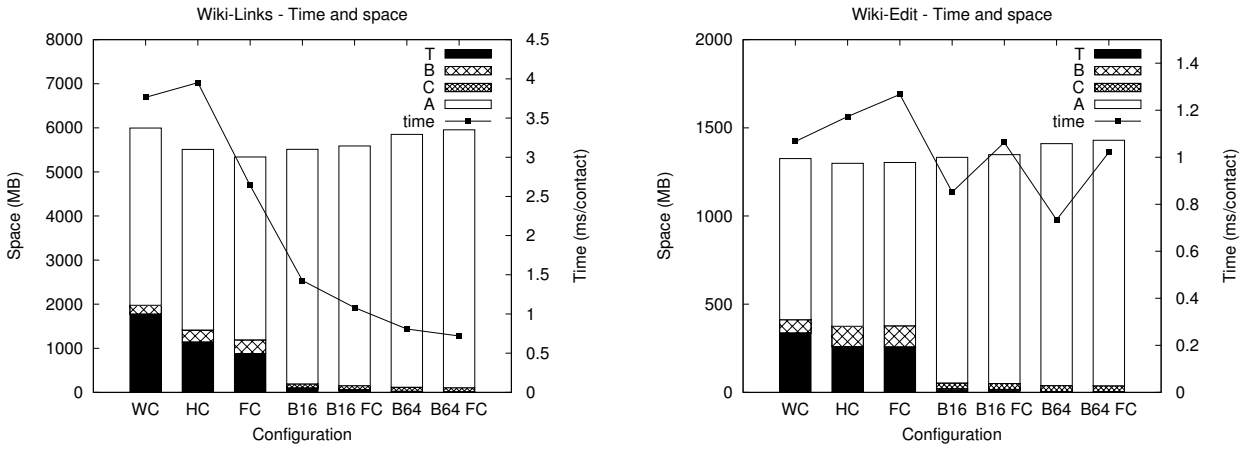

Figure 7.1: Time and space results for different configurations of the Compressed k^d-tree over the temporal graphs I-Wiki-Links and P-Wiki-Edit. We include three variants of node compression: WC is the plain version, without node compression; HC and FC use node compression for Half and Full levels of the tree, respectively; B16 and B64 use a bucket size of 16 and 64 cells, respectively; and B16 FC and B64 FC use a bucket size of 16 and 64 with full node compression, respectively. We separate the space requirements of the different components of the structure. We also report the time to retrieve direct neighbors in *ms* per output contact.

the improvement in retrieval time due to node compression. In the same way, we inherit the increase in retrieval time for 3D matrices.

In summary, when using ck^d-tree, the node compression technique (Section 6.4.1) works better in *interval-contact* temporal graphs (4D matrices) than in *point-contact* and *incremental* graphs (3D matrices). This result also holds for bck^d-tree if the bucket size is 2. When the bucket size is greater than 16, node compression does not work as expected, because the bitmap T already codifies a small tree.

7.3.2 Space comparison

We first compare our proposals ck^d-tree and bck^d-tree against the baseline k^d-tree. Figure 7.2 shows the space of the k^d-tree using RRR compressed bitmaps [RRR02] with a block size of 15, 63, and 255 bits over the *I-Wiki-Links* and *P-Wiki-Edit* graphs. As expected, the space of the k^d-tree is reduced when the block size of the RRR bitmaps increases, but this is at the expenses of increasing the retrieval time. With a block size of 63 bits, the k^d-tree gets its best configuration. In this case, ck^d-tree and bck^d-tree use 0.7 times the space and are several times faster than the k^d-tree.

Table 7.3 compares the ik^2-tree, the original k^d-tree and the entropy $\mathcal{H}$ against our ck^d-tree using node compression and the bck^d-tree. The table also includes the space of the Snapshot k^2-tree, which stores a k^2-tree of the active edges per each time instant, and the space usage for the TG-CSA based on the compressed suffix array [BCFR14], the structures CAS and CET based on sequences, and the structures EdgeLog and EveLog based on events

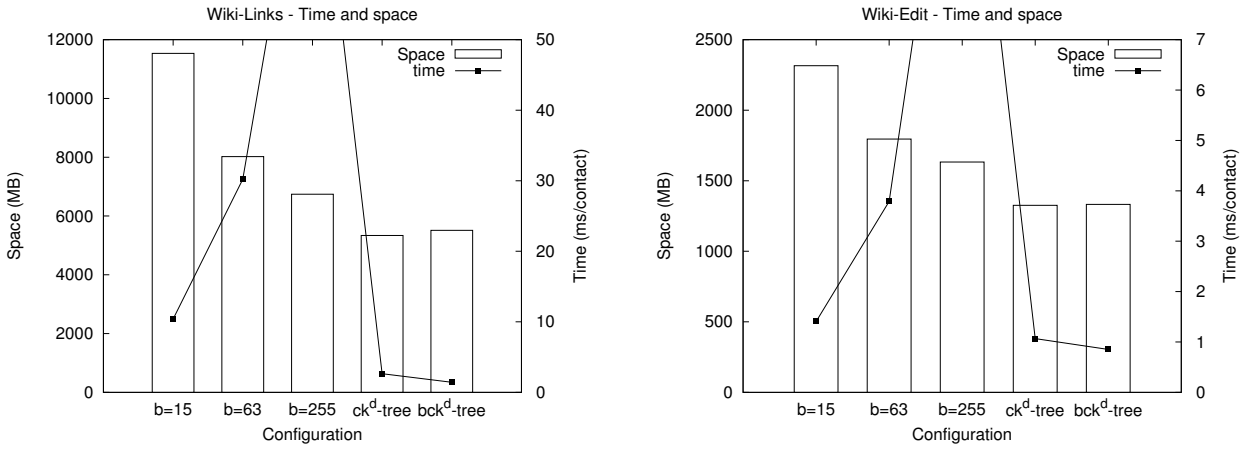

Figure 7.2: Time and space results of the k^d-tree using RRR compressed bitmaps with a block size of 15, 63, and 255 bits. The comparison use I-Wiki-Links and P-Wiki-Edit graphs. We include the space and time used by the ck^d-tree (FC variant) and the bck^d-tree (B16 WC). WC is the plain version, without node compression, while FC uses node compression for all levels of the tree. We also report the time to retrieve direct neighbors in *ms* per output contact.

and adjacency lists, respectively [CARB15]. Space is measured in bits per contact (bpc), by dividing the total space of the structure by the number of contacts in the graph.

In the experiment, and in the following sections, we report the best configurations for each graph. The first two columns of the table correspond to the space of our structures, the ck^d-tree and the bck^d-tree. For both structures we used uncompressed bitmaps with sampling every 1024 bits. We used the FC variant for *I-Comm.Net*, *I-Powerlaw*, *I-Wiki-Links*, and *I-Yahoo-Netflow*, the HC variant for *P-Yahoo-Sessions*, and the WC variant for *G-Flickr-Days*, *G-Flickr-Secs*, and *P-Wiki-Edit*. For the bck^d-tree, we report the bucket size $B16$ without node compression (WC variant) for all datasets.

The third column denotes the space used by the baseline, the k^d-tree using RRR compressed bitmaps with a block size of 63 bits. The fourth column corresponds to the space of the ik^2-tree. The fifth column is the space used by the Snapshot k^2-tree if we store the graph as several static graphs (snapshots), each of them containing the active edges for each time instant in the lifetime. The ik^2-tree and the Snapshot k^2-tree use uncompressed bitmaps with sampling each 640 bits (i.e., using 5% of extra space). The creation of the ik^2-tree and the Snapshot k^2-tree structures failed in some cases, which are marked with a dash[9].

The following four columns correspond to the structures CAS and CET, using the libcds implementation of RRR compressed bitmaps, with a block size of 15 bits, and the EveLog and EdgeLog using the PForDelta integer compressor [ZHNB06, ZLS08] with a block size of 128 elements and 32 elements, respectively. The last column is the space used by the TG-CSA using a sampling size each 64 items on the Ψ array [BCFR14].

[9]The program used to create the structures failed when the lifetime of the graph is greater than 10,000 instants.

Dataset	ck^d-tree	bck^d-tree	CAS	CET	EveLog	EdgeLog	TG-CSA	k^d-tree	ik^2-tree	Snap.k^2-tree	$\mathcal{H}$
I-Comm.Net	**26.0**	26.4	49.2	52.3	45.0	55.0	61.2	38.4	60.1	259.4	29.4
I-Powerlaw	**31.9**	32.6	56.4	68.0	77.5	96.2	73.8	48.8	73.0	2012.1	35.3
I-Wiki-Links	61.2	63.2	**34.5**	57.7	84.0	137.1	66.7	92.0	-	-	77.1
I-Yahoo-Netflow	**34.0**	36.1	47.1	62.9	103.7	150.5	62.7	47.4	-	-	57.5
G-Flickr-Secs	**46.1**	46.8	47.1	49.7	101.0	148.2	78.3	71.0	-	-	74.1
G-Flickr-Days	23.0	24.3	**18.8**	31.8	74.2	134.2	50.6	28.6	21.7	1724.0	32.2
P-Wiki-Edit	41.7	41.9	41.2	**38.3**	84.8	129.0	70.5	56.5	-	-	77.5
P-Yahoo-Session	47.1	45.2	**43.1**	49.1	131.8	200.9	66.6	46.0	-	-	65.8

Table 7.3: Space used by ours ck^d-tree and bck^d-tree against other structures for temporal graphs. The configurations used for each dataset is detailed in Section 7.3 (Size is in bits/contact (bpc)).

As we can see, the ck^d-tree and bck^d-tree obtain better space in half of the graphs, and they are always better than the entropy $\mathcal{H}$. The compression ratio is, in average, 74% of the k^d-tree and several times better than the Snapshot k^2-tree. The k^d-tree has better space than the ck^d-tree in the *P-Yahoo-Session* graph, however, our bck^d-tree gets better space. The only case when ik^2-tree achieves better space than ck^d-tree is in the I-Flickr-Days, because I-Flickr-Days has a short lifetime of 134 instants, which is the best case for ik^2-tree.

With respect to the sequence-based structures, CET obtains best space in *P-Wiki-Edit*, while CAS does in *I-Wiki-Links* and *G-Flickr-Days*, with a compression ratio of 91%, 68%, and 82% of the space used by the ck^d-tree, respectively. As *I-Wiki-Links* and *G-Flickr-Days* have a growing number of active edges, CET and CAS do not need to store the neighboring changes that deactivate edges at the end of the lifetime [CARB15]. Also, as *P-Wiki-Edit* is a *point-contact* temporal graph, the structures do not require to store the neighboring changes that deactivate edges, because by definition, all contacts have a duration of one time point. The CAS and EveLog are slow for reporting ReverseNeighbors, which is not a problem in our multidimensional proposals (CET, TG-CSA and ck^d-tree).

The space used by the EdgeLog, EveLog and TG-CSA is always greater than the entropy $\mathcal{H}$, regardless the type of the graph. The only exception is the *P-Wiki-Edit* graph using the TG-CSA structure, which uses 90% of the entropy space. As we will show in the time evaluation, however, these structures are very fast for answering most of the operations.

7.4 Time evaluation

This section presents the time evaluation following the classification of operations defined in Section 3.1.1. Each subsection includes an evaluation for time instants and for time intervals under both *weak* and *strong* semantics. We also include a subsection to evaluate the hybrid representation of the ck^d-tree for *interval-contact* graphs. In addition, we report a sensitive analysis to check how the number of contacts per edge and per vertex impact the time evaluation of all the structures.

7.4.1 Active edge retrieval and next activation

In this section we study the efficiency of checking if an edge is active at a time-point and during a time interval, and the efficiency of obtaining the time instant of the next activation of an edge (EdgeNext operation). In this experiment, and in the following sections, we choose

the variant of the data structures reported in 7.3. We also added three different bucket sizes (i.e., 2, 16, and 64) for the bck$^{\mathsf{d}}$-tree, and three block sizes (15, 63, and 255) of the RRR bitmaps for the k$^{\mathsf{d}}$-tree.

We generated 2,000 queries by a random selection of 2,000 contacts from the graphs. For each selected contact (u, v, t_a, t_b), we used the source and target vertices (u, v), and the beginning of the time interval t_a to perform the Edge and EdgeNext operations. The time performance is measured in μs per contact reported and space is measured in bits per contact (bpc).

Figures 7.3 and 7.4 show the performance of Edge and EdgeNext operations using the best configuration for each graph. We selected graphs *I-Powerlaw*, *I-Wiki-Links*, *I-Yahoo-Netflow*, *G-Flickr-Days*, *G-Flickr-Secs*, and *P-Wiki-Edit* as the representative datasets because they reflect the performance of all structures. In general, the time performance of Edge and EdgeNext is similar for all structures. The ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree are faster than the original k$^{\mathsf{d}}$-tree baseline, even in the slowest configurations. The effect of the bucket size works as expected, decreasing the time to recover the state of an edge as the size of the bucket increases. The ik^2-tree has the best performance for the dataset that we were able to create. The best case of ik^2-tree, the G-Flickr-Days graph, is 65% faster than our ck$^{\mathsf{d}}$-tree. The sequence-based methods CET and CAS only have a good performance in graphs with a growing number of active edges with a long lifetime (*G-Flickr-Secs* and *I-Wiki-Links*). This also occurs with TG-CSA, although with a heavy penalty on space. In the other temporal graphs, the ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree show the best tradeoff. The EdgeLog is several times faster than the EveLog, although both are almost four times heavier in space than ck$^{\mathsf{d}}$-tree. The bad performance of EveLog is due to the traversal of the log of events.

We evaluated the time performance for Edge operations for time interval using *weak* and *strong* semantics. The experiment includes four different interval sizes, corresponding to the 0.1%, 1%, 10% and 50% of the lifetime of the graph. We generated 2,000 queries by selecting the source and target vertices and a time instant (as in the begining of this section). For the interval size, we set the query interval by defining the end of the interval as the starting time instant plus the percentage of the lifetime. With this method, we ran 2,000 queries per interval over the same set of vertices. Time performance is measured in μs per query. For completeness, we also added the running time of the Edge operation over a time instant. We omitted the EveLog structure because it is several times slower than any other structure.

The evaluation is made over *I-Powerlaw* and *I-Wiki-Links* graphs, as they reflect the time performance of graphs with a short and a large lifetime. Figure 7.5 shows the evaluation for *weak* and *strong* semantics. The running time follows the same performance obtained in the evaluation of time-instant queries. In ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree, the time using *strong* semantics tends to diminish when the interval size grows, but the opposite occurs when using *weak* semantics. This happens because all contacts that overlap the interval must be recovered. The exception to this rule is CAS, whose time performance increases with the interval size for both *weak* and *strong* semantics.

7.4.2 Direct and reverse active neighbors

This section presents the evaluation of the time performance to retrieve the set of direct and reverse neighbors that are active at a time instant and during a time interval. For this experiment, we ran 2,000 queries randomly chosen from the set of contacts, following

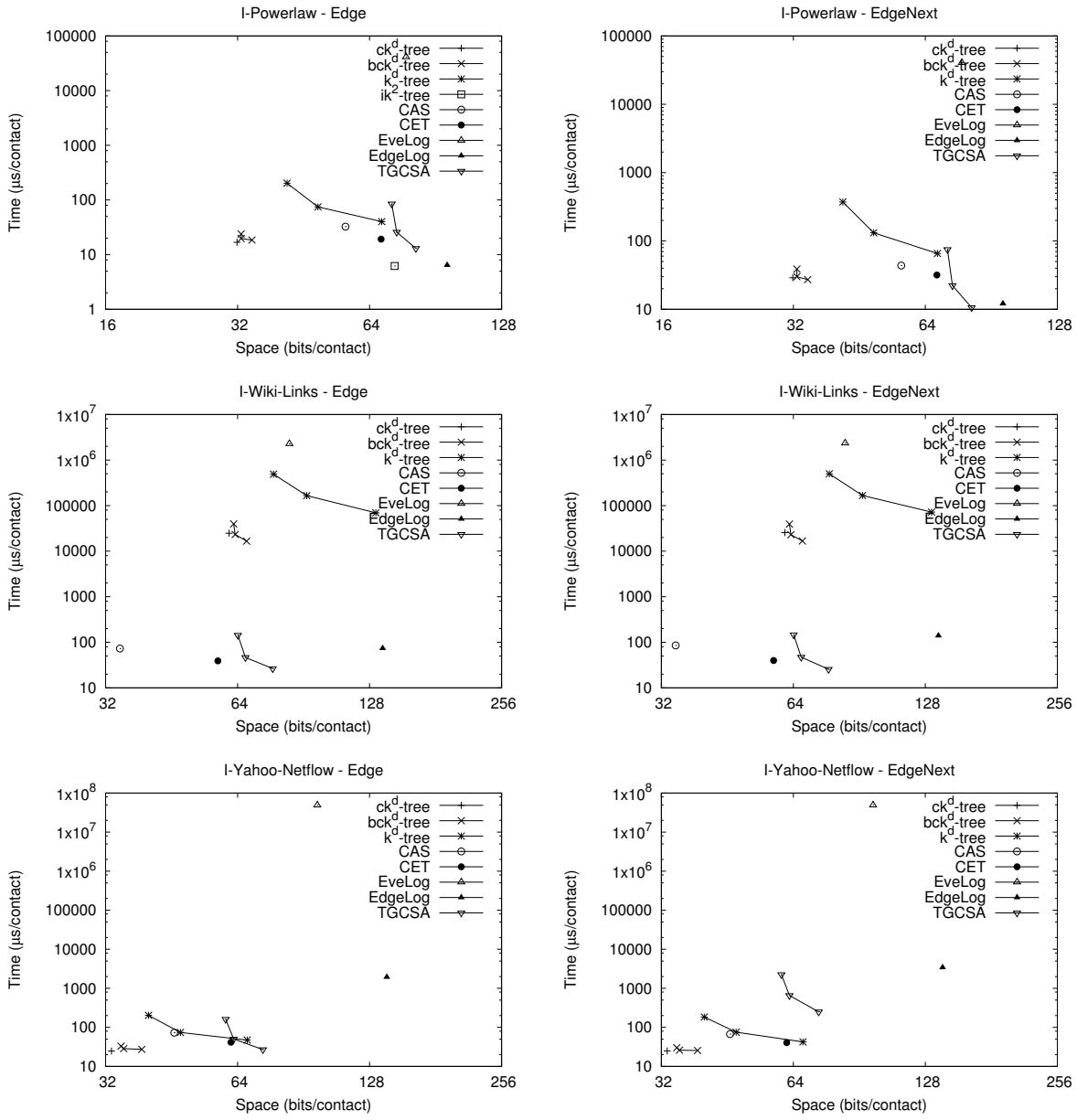

Figure 7.3: Time and space performance of Edge and EdgeNext operations using the best configuration for graphs *I-Powerlaw*, *I-Wiki-Links*, and *I-Yahoo-Netflow*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

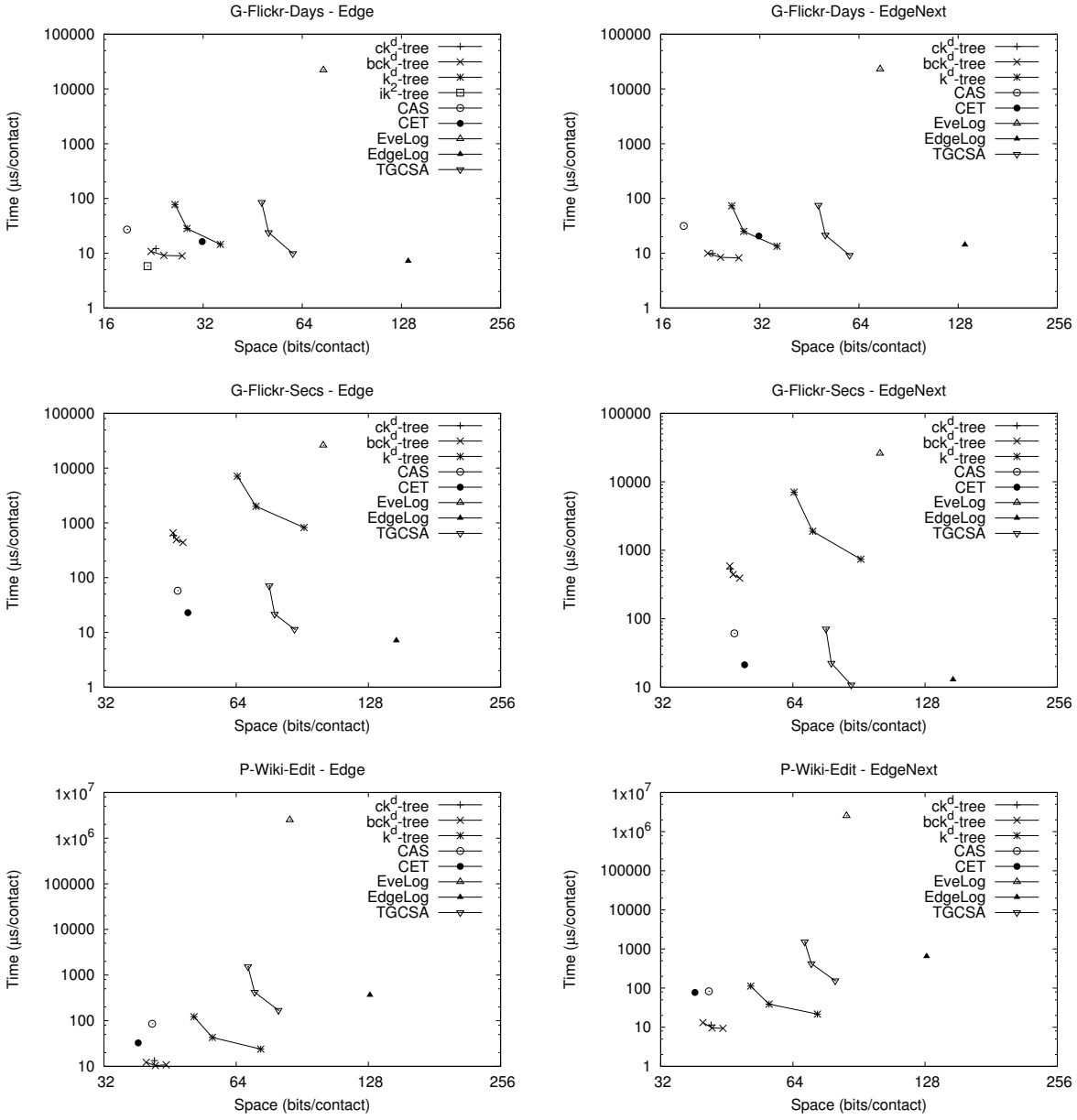

Figure 7.4: Time and space performance of Edge and EdgeNext operations using the best configuration for graphs *G-Flickr-Days*, *G-Flickr-Secs*, and *P-Wiki-Edit*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

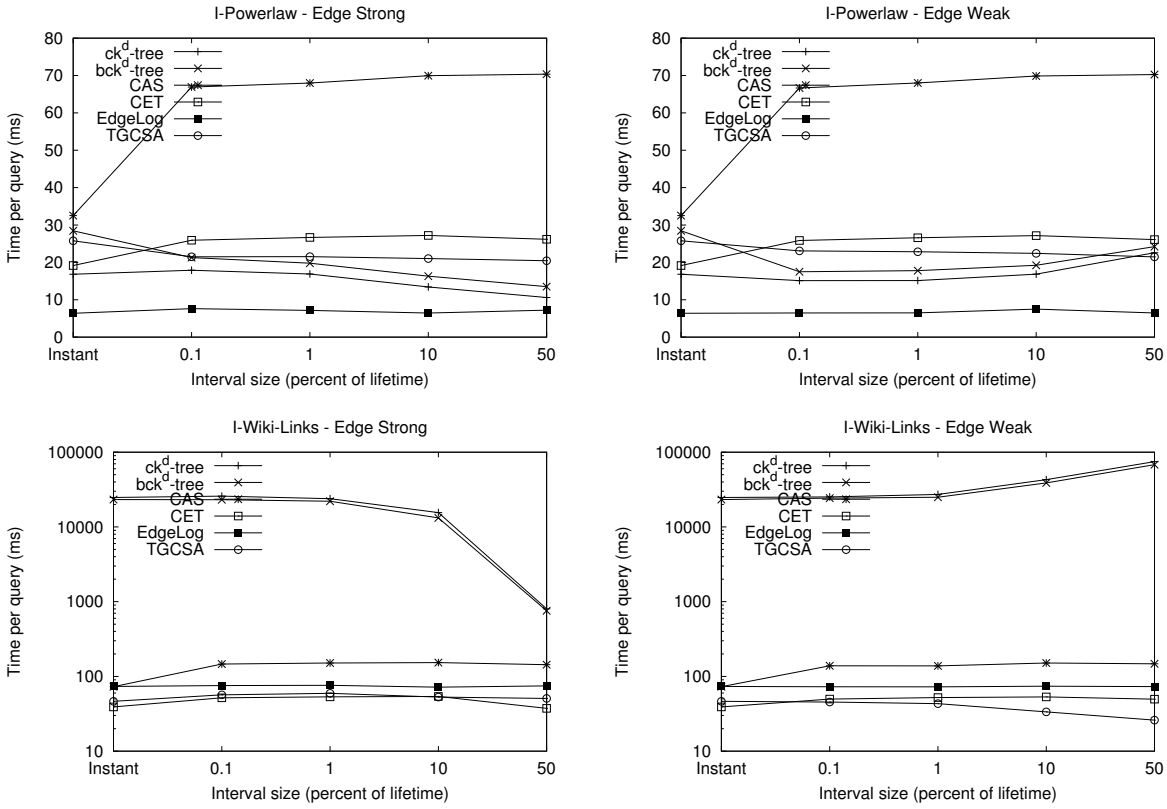

Figure 7.5: Time performance of the interval version of Edge over the *I-Powerlaw* and *I-Wiki-Links* graphs. The image on the left shows the *strong* and on the right the *weak* semantics. Time performance is measured in μs per query.

the same strategy used in the previous section, but only selecting the source vertex of each contact. We measured the time performance in μs per contact reported, and space in bits per contact (bpc). Figures 7.6 and 7.7 show the space-time tradeoff for DirectNeighbors and ReverseNeighbors over a time instant using the best configuration for each graph (see Section 7.3).

The ck^d-tree and bck^d-tree always outperform the k^d-tree for at least one order of magnitude. In the graphs where ck^d-tree achieves the smallest space, it also achieves a good time performance. The variation of the bucket size for the bck^d-tree works as expected, reducing the retrieval time with larger buckets at the expenses of increasing the size of the data structure. CAS and CET have a good performance in graphs with a growing number of active edges with a long lifetime. However, CAS is very slow for answering ReverseNeighbors operations. The modifications made to CET for representing *point-contact* graphs works very well, achieving the best performance in these graphs. As we mentioned before, EdgeLog has a good time performance but it requires at least four times the space of the smallest structure. The TG-CSA structure has a good time performance for DirectNeighbors and ReverseNeighbors on *incremental* graphs.

As the figures show, the time performances of ck^d-tree, bck^d-tree, and CET are similar for both DirectNeighbors and ReverseNeighbors operations. Small variations in time performance are due to the number of contacts in the output. In *I-Powerlaw* and *G-Flickr-Days*, the ck^d-tree and bck^d-tree are 3 to 9 times slower than storing the Snap.k^2-tree in fraction of the space. CET and TG-CSA have similar time performance, but they use more space than the ck^d-tree.

Regarding time interval queries, we also evaluated the time performance under *weak* and *strong* semantics. The experiment included four different interval sizes, corresponding to the 0.1%, 1%, 10% and 50% of the lifetime of the graph. We generated 2,000 queries following the same strategy for the interval evaluation in Section 7.4.1. Time performance is measured in *ms* per query. To see the effect of the semantics we also report the number contacts in the output, and the time per query for a time instant. We ran the experiments over the *I-Powerlaw* and *I-Wiki-Links* graphs, as they reflect the performance of graphs with short and long lifetimes.

Notice that we are not evaluating the performance of interval queries on *incremental* graphs, because they can be reduced to time instant queries (as we pointed out in Section 3.1). We skipped the results of interval ReverseNeighbors operations because the time performance of ck^d-tree, bck^d-tree, and CET are similar to DirectNeighbors operations. The bad performance of CAS in ReverseNeighbors remains poor for time interval queries.

Figure 7.8 shows the time comparison of ck^d-tree, bck^d-tree, CAS, and CET. As expected, the output size of the *strong* semantics decreases as the interval size increases. The opposite occurs for the *weak* semantics, where the output size always increases or remains the same.

Like for time instant queries, the performance over the *I-Powerlaw* graph of our ck^d-tree and bck^d-tree is better than CAS, but slower than CET in *weak* and *strong* semantics. In the *I-Wiki-Links* graph, the time performance is similar to the obtained in the time instant, except for large intervals. In *strong* semantics, the time performance of CAS and CET tend to increase with the interval size. This is due to the nature of the data structures, based on a counting method to retrieve the state of edges. Because ck^d-tree and bck^d-tree search for contacts with a time interval greater or equal to the query, they do not suffer of this problem and the time performance diminishes with the interval size.

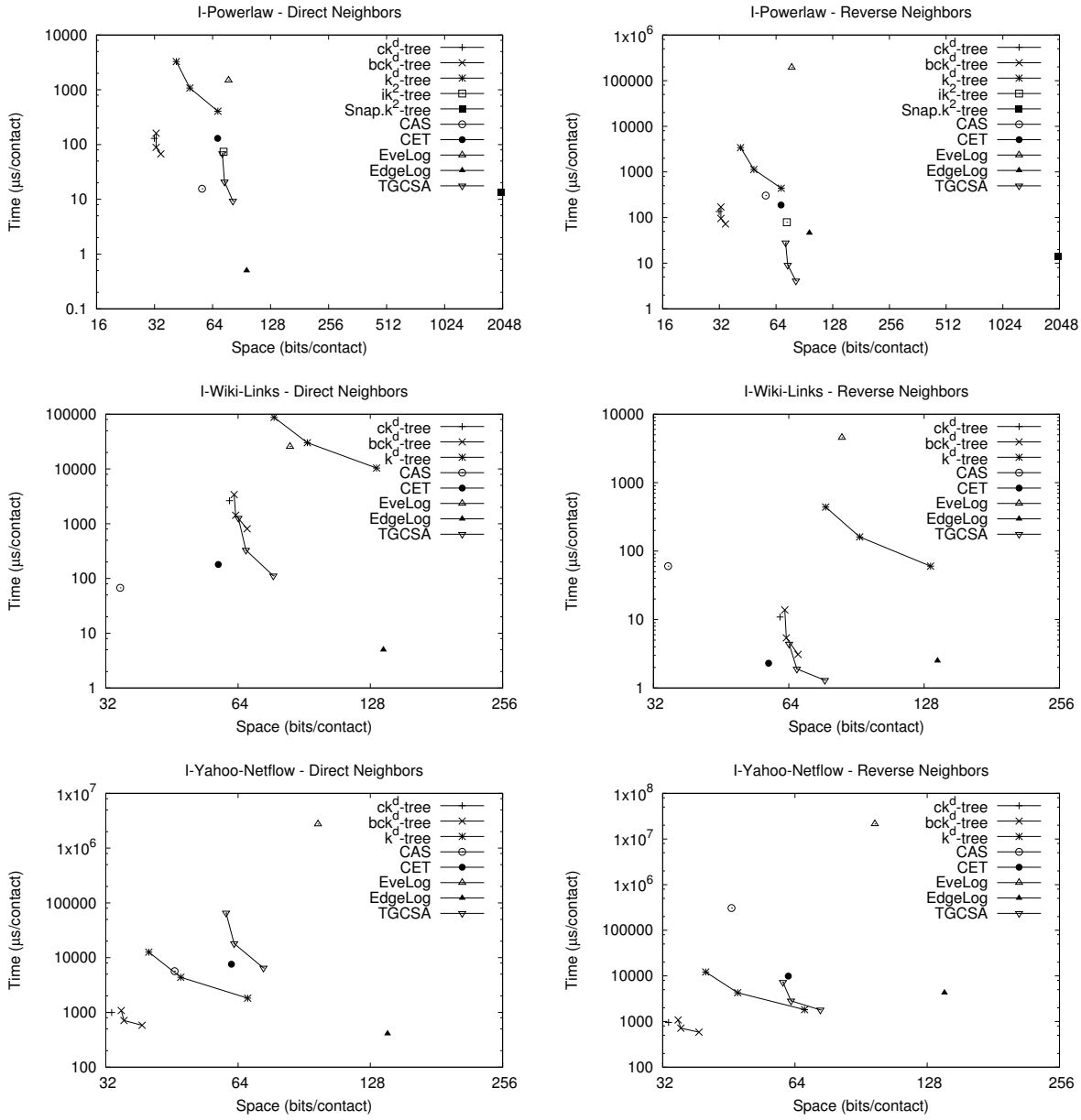

Figure 7.6: Time and space performance of DirectNeighbors and ReverseNeighbors operations over a time instant using the best configuration for graphs *I-Powerlaw*, *I-Wiki-Links*, and *I-Yahoo-Netflow*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

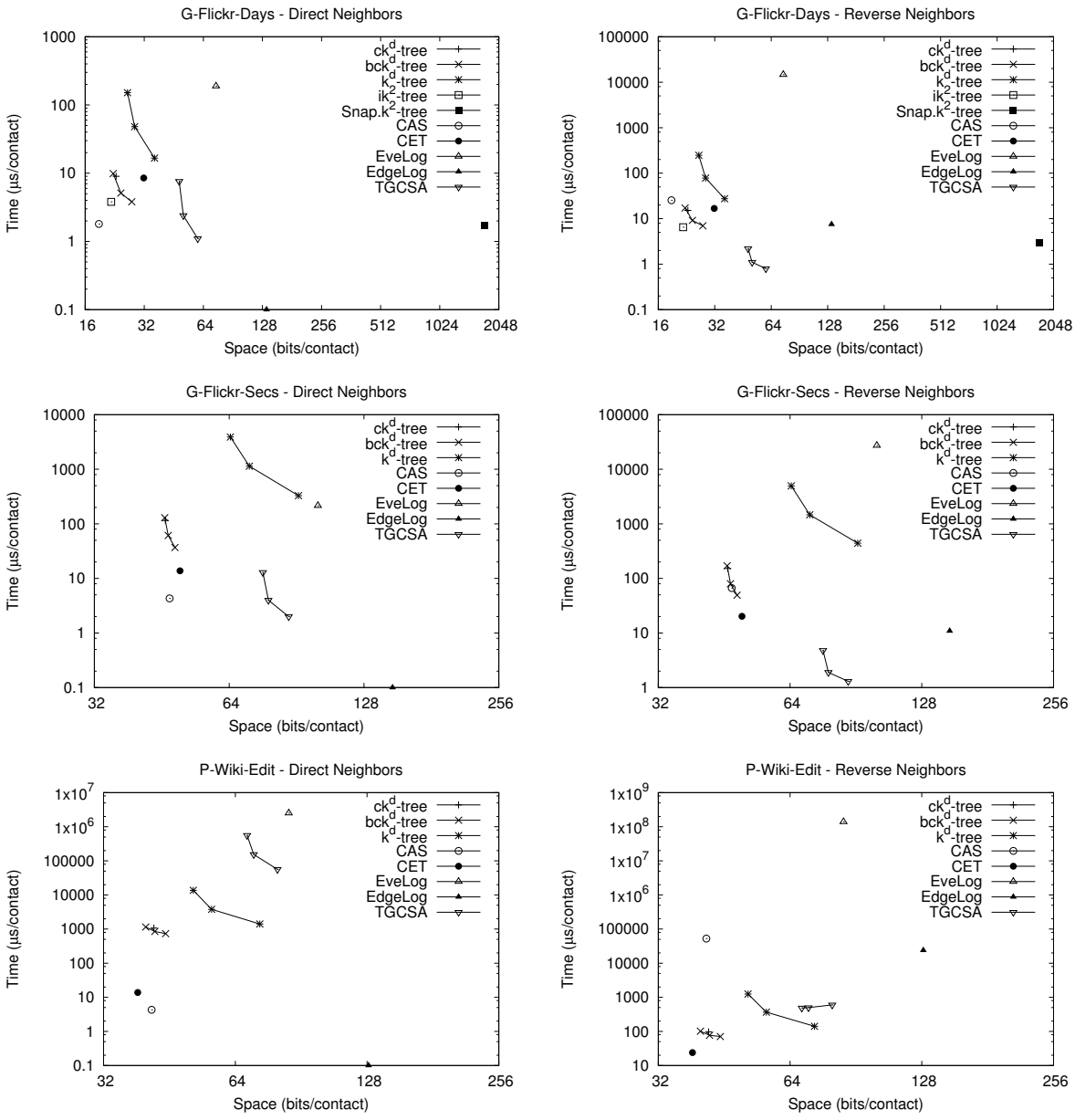

Figure 7.7: Time and space performance of DirectNeighbors and ReverseNeighbors operations over a time instant using the best configuration for graphs *G-Flickr-Days*, *G-Flickr-Secs* and *P-Wiki-Edit*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

Using the *weak* semantics, the time tend to increase with the size of the interval in ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree. At the largest query interval, the 50% of the lifetime, ck$^{\mathsf{d}}$-tree obtains the worst performance. We checked that this increase is not related to the extra step used to remove duplicated vertices by also running the query without the extra step and obtaining a similar performance. In this case, the performance of CET and CAS do not change with the size of the interval. The performance of EdgeLog and TG-CSA do not change with both the semantics and the size of the time interval. Indeed, they got the same performance of DirectNeighbors over a time instant in both temporal graphs.

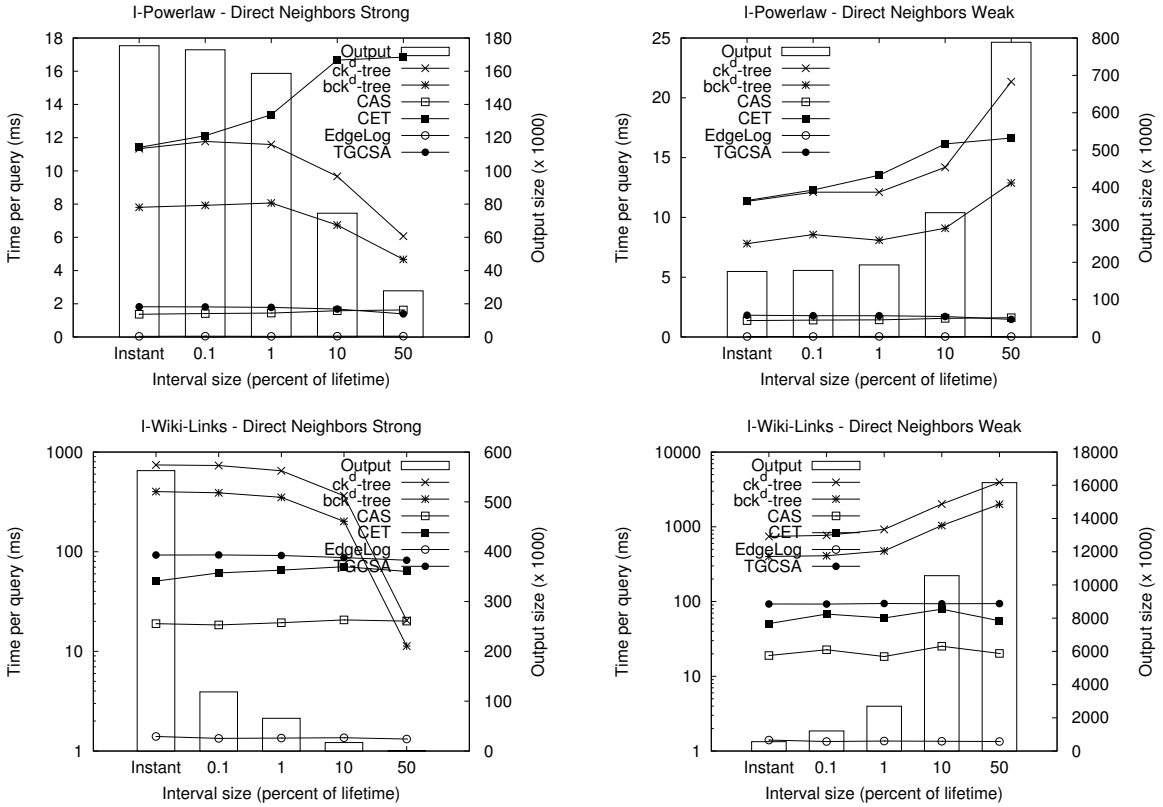

Figure 7.8: Time performance of the interval version of the DirectNeighbors operation over the *I-Powerlaw* and *I-Wiki-Links* graphs. The image on the left shows the *strong* and on the right the *weak* semantics. Time performance is measured in *ms* per query.

7.4.3 Snapshot retrieval

We studied the performance of retrieving the set of all active edges at a certain time-point (Snapshot operation). We compared the average retrieval time in four different instants: the 25%, 50%, 75%, and the 100% of the lifetime of the temporal graphs. Figure 7.10 provides the average number of active edges per time instant, that is, the expected output size. For the ik^2-tree we computed the operation by retrieving the state of all cells that ever changed their status active/inactive before the query time. The performance is measured as the time

to perform the query in seconds.

Figure 7.9 shows the time performance over graphs *I-Powerlaw*, *I-WikiLinks*, *G-FlickrDays*, and *P-WikiEdit*. As it can be seen, the time performance of ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree is several times faster than sequence-based structures over *interval-contact* and *incremental* graphs. In *point-contact* graphs, CET outperforms all the structures. Taking into account the dynamism of *I-Powerlaw*, which has a constant number of active edges per time instant (Figure 7.10a), it is clear that the counting method of the ik^2-tree, CAS, and CET does not scale well with the lifetime of the graph. Nevertheless, this does not seem to be a real issue in graphs with a growing number of active edges, such as *I-WikiLinks* and *G-FlickrDays* shown in Figure 7.10b. The performance of the ck$^{\mathsf{d}}$-trees follows the same tendency of the TG-CSA, stable for *I-Powerlaw* and growing for *I-WikiLinks* and *G-FlickrDays* graphs.

Regarding the nature of the *P-WikiEdit* graphs, with unit duration of contacts, it is not surprising the poor results obtained by ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree, because few contacts are active per time instant. However, the query time is very fast, requiring between 5-10μs to retrieve these contacts.

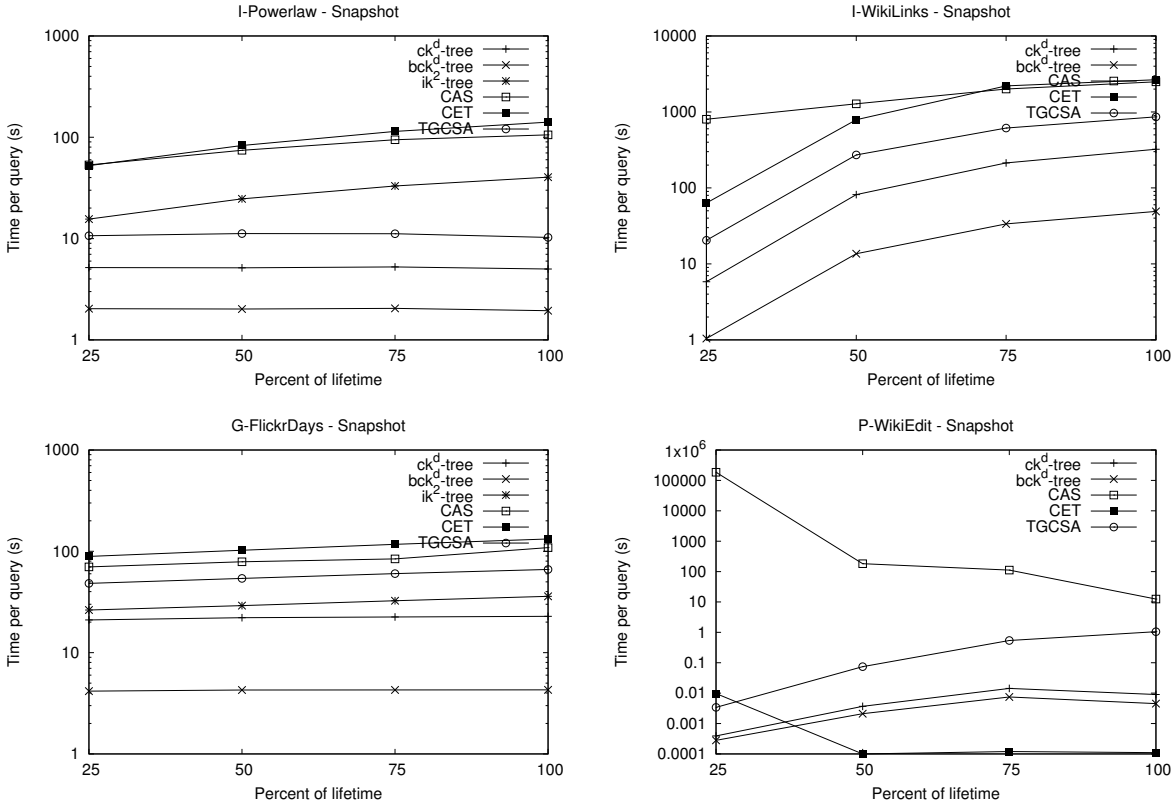

Figure 7.9: Time comparison of Snapshot at the 25%, 50%, 75%, and 100% of the datasets' lifetime. (Time in s per query).

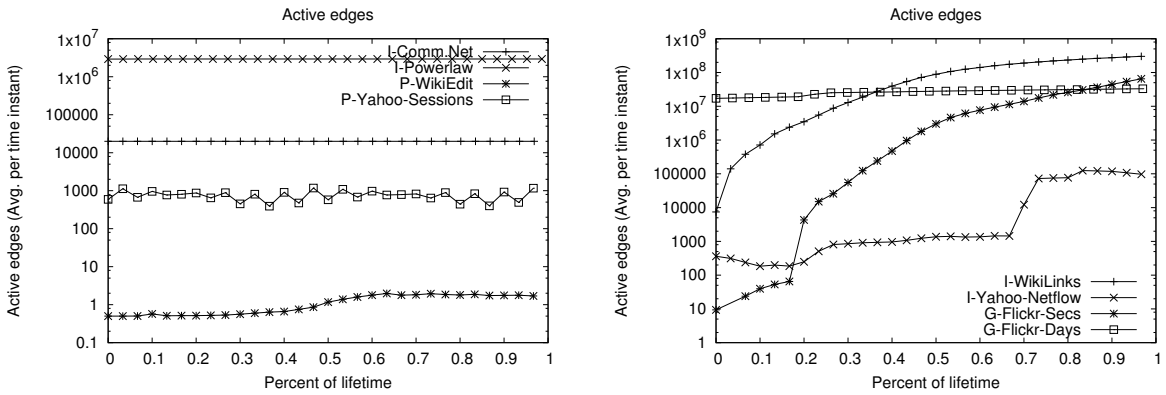

Figure 7.10: Average number of active edges per time instant in all datasets. Figure on the left shows results for graphs *I-Comm.Net*, *I-Powerlaw*, *P-WikiEdit*, and *P-Yahoo-Sessions* with a uniform number of active edges. Figure on the right shows results for graphs *I-WikiLinks*, *I-Yahoo-Netflow*, *G-Flickr-Secs*, and *G-Flickr-Days* with an increasing number of active edges.

7.4.4 Events on edges

This section shows the performance of ActivatedEdges queries, retrieving the set of edges that have been activated at a time instant or during a time interval. For the evaluation we generated 2,000 random time instants, uniformly distributed over the lifetime of the corresponding graph. The experiments were performed for each time instant, and also over four different sizes of time intervals, corresponding to a minute, an hour, a day, and a week. The performance is measured as the average time to perform a query in μs. The evaluation only considered ck$^{\mathsf{d}}$-tree, bck$^{\mathsf{d}}$-tree, and CET. In other data structures, such as CAS, EdgeLog and EveLog, the performance is very poor for this type of query, while for TG-CSA and ik^2-tree these queries have not been implemented. We are only reporting here the time performance of the ActivatedEdges queries, because the strategy (the search range) is analogous to obtain DeactivatedEdges and ChangedEdges queries.

Figure 7.11 shows the performance over graphs with a time granularity of one second, i.e., over *I-Wikipedia-Links*, *G-Flickr-Secs*, and *P-Wiki-Edit* graphs. As it can be seen, CET is the fastest structure for retrieving edges activated at time instants. These results hold for the three types of temporal graphs: *interval-contact*, *point-contact*, and *incremental*.

When the interval size is one hour, the time performance of CET is the same than of ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree. In larger time intervals (for a day and a week), ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree achieve the best performance. Although we only present the time interval results for *I-Wikipedia-Links*, the same performance holds for *G-Flickr-Secs* and *P-Wiki-Edit*.

7.4.5 Effect of partitioning contacts in *interval-contact* graphs

The similar growth of active edges of *I-Wiki-Links* and *G-Flickr-Secs* graphs (see Figure 7.10) made us think that *I-Wiki-Links* share some properties of *incremental* graphs. Similarly, the *I-Yahoo-Netflow* graph has a low number of active edges per time instant,

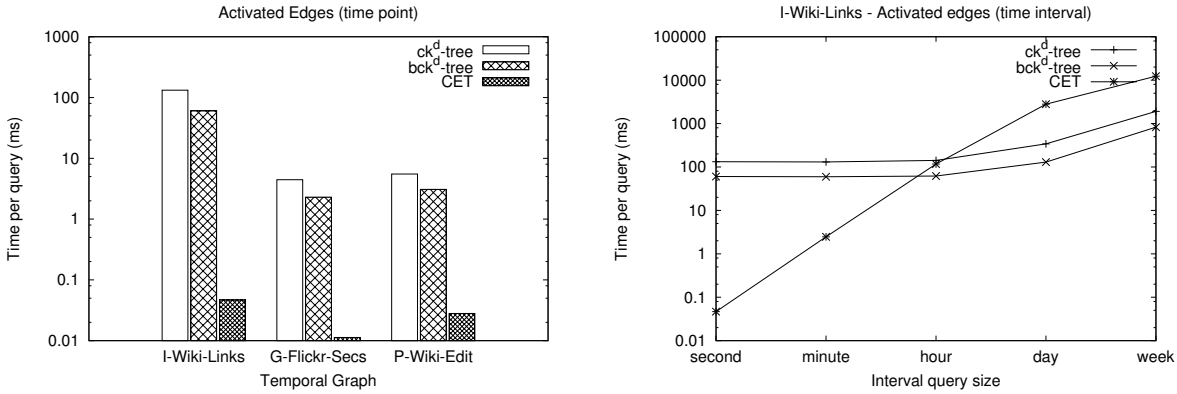

Figure 7.11: Time performance of ActivatedEdges queries at time instants and during time intervals. Figure on the left shows the performance for time instants using *I-Wikipedia-Links*, *G-Flickr-Secs* and *P-Wiki-Edit* graphs. Figure on the right shows the performance of *I-Wikipedia-Links* over time intervals corresponding to a second, an hour, a day, and a week.

which also suggests that some contacts are active by only one time instant as in *point-contact* graphs. This makes the *I-Wiki-Links* and *I-Yahoo-Netflow* graphs be good candidates to test the usefulness of the partition technique proposed in Section 6.3.3. Indeed, 42% of contacts in *I-Wiki-Links* belong to the *incremental* class (i.e., there are 307,690,160 active edges at the end of the lifetime), and 89% of contacts in *I-Yahoo-Netflow* are active during a time instant. Consequently, we partitioned contacts of these graphs into a set of 3D tuples representing contacts in the *incremental* and *point-contact* classes, and a set of 4D tuples representing other types of contacts.

Table 7.4.5 shows the space and time performance of the 4D and the 3D+4D representation. The time performance is measured as the average time per query to run the DirectNeighbors operations as in Section 7.4.2. The space usage is in bits per contact (bpc). The improvement ratio of the partitioning with respect to the 4D representation in *I-Wiki-Links* is 0.89 and in *I-Yahoo-Netflow* is 0.8 in average. In *I-Wiki-Links* the time also improves, between 0.90 and 0.95 times the performance of the 4D representation. Queries on the *I-Yahoo-Netflow* graph using the ck$^{\mathsf{d}}$-tree with the partition of contacts run 10% slower than using the ck$^{\mathsf{d}}$-tree with the 4D representation, but run in similar time with the bck$^{\mathsf{d}}$-tree. Although the idea of dividing the contacts of a temporal graph by its temporal behavior sounds simple, it works very well on practice.

7.4.6 Sensitivity analysis with respect to out degree and number of contacts

This section shows a sensitivity analysis in order to check how the time to answer queries in the structures depends on the number of direct neighbors (out degree in the aggregated graph) and the number of contacts per vertex. For this propose, we generated three synthetic temporal graphs by selecting a degree distribution of the aggregated graph (i.e., the static graph composed by the edges of the temporal graph), and assigning a number of contacts to each edge.

Dataset	Structure	Space (bpc)			Time (ms/query)		
		4D	3D+4D	Ratio	4D	3D+4D	Ratio
I-Wiki-Links	ck^{d}-tree	61.2	54.8	0.89	742.7	702.0	0.95
	bck^{d}-tree	63.2	56.5	0.89	401.4	360.3	0.90
I-Yahoo-Netflow	ck^{d}-tree	33.0	26.6	0.81	90.0	99.1	1.10
	bck^{d}-tree	35.7	28.3	0.79	64.3	63.5	0.99

Table 7.4: Improvement of using the partitioning of contacts in *I-Wiki-Links* and *I-Yahoo-Netflow* graphs. The *interval-contact*, *incremental*, and *point-contact* graphs were represented by using a 4D, 3D+4D binary matrices, respectively.

Table 7.5 shows the main characteristics of the synthetic graphs. In BA.100k.U1000 and BA.100k.U100, the aggregated graph corresponds to a Powerlaw degree distribution created from the Barabási-Albert model [AB02] and a uniform number of contacts per edge (1000 and 100 contacts per edge, respectively). The ER.1M.P10 graph follows a uniform degree distribution on the aggregated graph generated from the Erdös-Rényi model [AB02]. The number of contacts per edge follows a Pareto distribution with $\alpha = 1.0$.

Dataset	Type	Vertices (n)	Edges (m)	Lifetime (τ)	Contacts (c)	c/m	Size (MB)	$\mathcal{H}$ (MB)
BA.100k.U1000	Interval	100,000	941,408	100,000	941,408,000	1000	7198	4160
BA.100k.U100	Interval	100,000	941,408	100,000	94,140,800	100	734	453
ER.1M.P10	Interval	1,000,000	10,001,583	1,000,000	122,731,342	12.27	1104	780

Table 7.5: Description of temporal graphs used in the sensitivity analysis

Before going further, we present the space used by structures in Table 7.6. The space of ck^{d}-trees is under the entropy $\mathcal{H}$, but it is above the space used by EdgeLog. However, as we will see in what follows, the main drawback of EdgeLog and EveLog is the time to process the list of events and neighboring vertices. Other structures use more space than the ck^{d}-tree structure.

Dataset	ck^{d}-tree	bck^{d}-tree	k^{d}-tree	CAS	CET	EveLog	EdgeLog	TG-CSA	$\mathcal{H}$
BA.100k.U1000	29.9	31.2	45.3	37.1	43.6	30.4	**18.2**	64.5	37.1
BA.100k.U100	36.6	37.6	57.5	46.5	46.6	36.7	**26.5**	67.1	40.4
ER.1M.P10	44.5	47.1	70.5	54.7	51.5	68.9	**42.8**	73.6	53.5

Table 7.6: Space used by compressed data structures for temporal graphs in the sensitivity analysis. (Size is in bits/contact (bpc)).

In the first experiment, we evaluated the sensitivity of the structures with respect to the number of contacts of each vertex. Figure 7.12 shows the average time per query to retrieve DirectNeighbors and Edge operations, grouped by the number of contacts per vertex in the ER.1M.P10 graph (which is a variable number of contacts per vertex). The ck^{d}-tree and bck^{d}-tree, as well as CAS and CET, do not vary their time performance with respect to the number of contacts of the vertex in the query. On the contrary, both EdgeLog and EveLog increase their time to answer queries as the number of contacts grows. Notice that in this evaluation, the TG-CSA works much slower than the other structures, contrary as we got in the time performance evaluation in Section 7.4.2. These results are in concordance with the analytical comparison of Section 7.1, as the retrieval time of the TG-CSA depends on

the number of contacts per vertex.

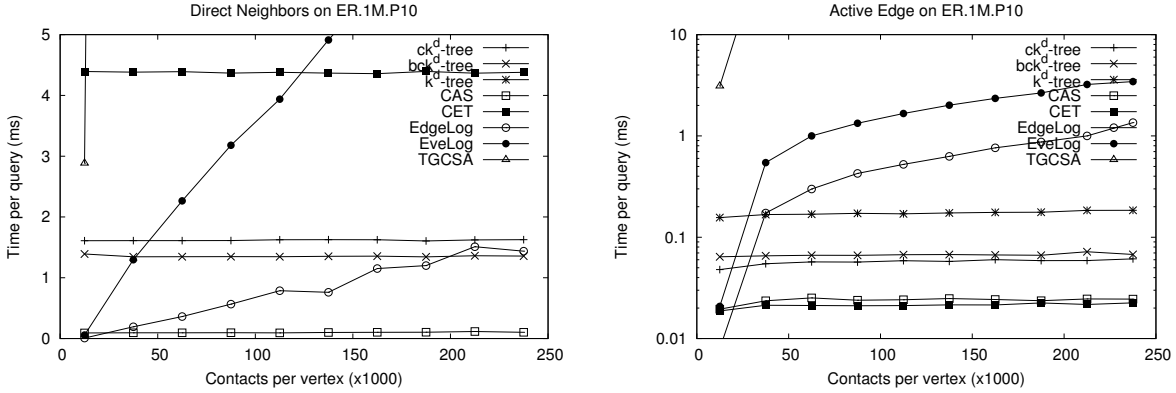

Figure 7.12: Sensitivity analysis of DirectNeighbors and Edge operations in the ER.1M.P10 synthetic graph. Time per query (ms) v/s contacts per vertex.

In the second experiment, we evaluated the sensitivity with respect to the out degree of vertices. Figure 7.13 shows the time to answer DirectNeighbors queries over the BA.100k.U1000 and BA.100k.U100 graphs, grouped by the out degree on vertices. The time of ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree is stable on both graphs. The time increases for the BA.100k.U1000 graph because this graph has ten times the number of contacts per vertex of BA.100k.U100, but it is still stable on the out degree per vertex. As expected, the time to answer queries by the CAS and CET effectively increases with the out-degree. The results obtained in the first experiment hold for EdgeLog and EveLog, the time increases with the number of contacts per vertex if we compare the slope in the curves of the BA.100k.U1000 and BA.100k.U100 graphs. As before, the performance of the TG-CSA is poor, several times slower than the other structures.

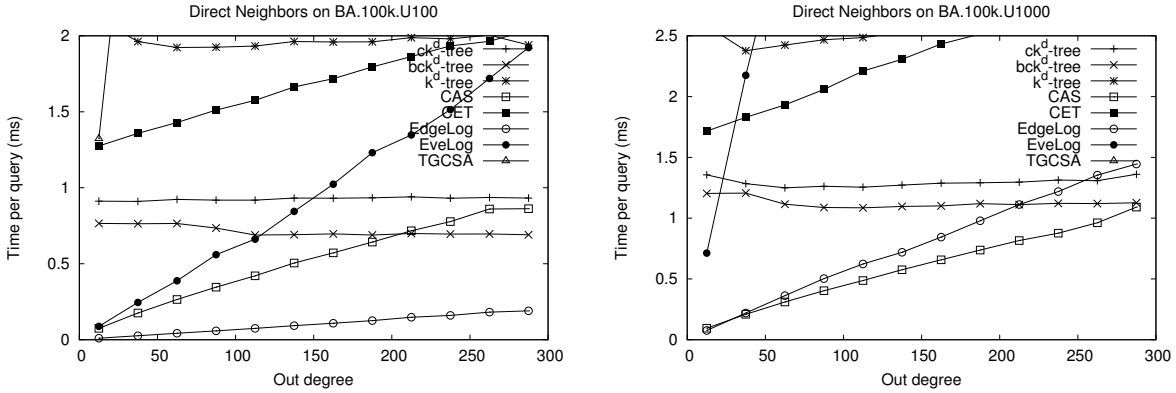

Figure 7.13: Sensitivity analysis of answering DirectNeighbors queries over the BA.100k.U1000 and BA.100k.U100 synthetic graphs. Time per query (ms) v/s out degree per vertex.

7.5 Final comments

The experiments in this work showed that the space used by even a simple and uncompressed representation of an adjacency log is several times less than compressed data structures for a snapshot-based approach. Indeed EdgeLog is very efficient in time to obtain direct neighbors, but at the expense of doubling or quadrupling the space of CAS, CET and ck$^{\mathsf{d}}$-tree in some cases. On edge-related queries, the linear approach followed by EdgeLog and EveLog, i.e., the sequential scanning of the log, is clearly outperformed by the logarithmic approach followed by CAS and CET. Regarding the 3D and 4D representations, the space used by ck$^{\mathsf{d}}$-tree is near the 50% of the fastest version of the k$^{\mathsf{d}}$-tree using compressed bitmaps. In addition, it can be several times faster than CET and CAS for queries that recover the state of the graph at a given time instant. Also, it is several times smallest than the TG-CSA and is also capable of recovering all components (i.e. source and target vertices and the time interval) of a contact on all operations.

Although ck$^{\mathsf{d}}$-tree ensures a space close to the information-theoretic lower bound, do not always obtain the best compression ratio. For example, in the I-Wiki-Links graph, CAS uses half of the space of ck$^{\mathsf{d}}$-tree. This could be related with the size of time intervals of each contact or the size of the lifetime, both large in I-Wiki-Links. However, in other datasets with short lifetime and short time intervals such as G-Flickr-Days and P-Yahoo-Session, CAS also gets the best space. A further research in the characterization of the dynamics of temporal graphs could help us to understand this difference on the compression ratio.

We also compared the space usage of the proposed structures against the space used by ik^2-tree for temporal graphs. In this case, ck$^{\mathsf{d}}$-tree uses up to 50% less space than ik^2-tree, or the same space in the best case of the ik^2-tree (G-Flickr-Days datasets with a short lifetime). With respect to the time performance, the ck$^{\mathsf{d}}$-tree outperforms 2-4 times the ik^2-tree on Snapshot queries for all datasets except the G-Flickr-Days dataset, in which case both structures show a similar time performance. In the sensitivity analysis, we show that the time performance of the operations does not depend on the number of contacts per vertices nor the out degree of the aggregated graph, as it happens in the other structures.

Among the data structures, some work better than others depending on the kind of queries. The experiments show that CAS is lighter and faster than CET for direct neighbors. However, CET, ck$^{\mathsf{d}}$-tree and the TG-CSA have the same performance on both direct and reverse neighbors. So, a careful decision must be made to choose the one for a specific type of queries.

The greatest improvement obtained of the compact sequence representations is when retrieving the state of a single edge, because CAS and CET do not necessarily recover all the history of activation/deactivation of the edge, they just count how many events related with the edge have occurred. The only exception is the ReverseNeighbors query in CAS, which needs to track each contact related to the chosen vertex.

Experiments show the exceptional running time of CET when we want to retrieve which edges change their state, or have been activated/deactivated at a time point. To obtain changes over long time intervals, the ck$^{\mathsf{d}}$-tree is a better choice. The other data structures work poorly, because they need to traverse all vertices, and for each vertex, recovering the temporal log and looking for what changed at a given time.

Chapter 8

Conclusions and future work

8.1 Summary of contributions

The main goal of this thesis was to create space and time efficient compressed representations of temporal graphs. Our main aim is to replace the basic strategy of representing temporal graphs as several snapshots. This work explored different compact structures for temporal graphs conceptualized as temporal adjacency logs, as multidimensional sequences, and as cells in a 4-dimensional binary matrix. The experimental evaluation shows that the use of compression techniques or compact and self-indexed data structures provides not only interesting savings in space but also good time performance for answering a broad set of useful queries.

We started by proposing basic compact representations that use text compressed indexes. They are based on inverted indexes, strategies referred as EdgeLog and EveLog. Access operations in EdgeLog and EveLog traverse the temporal log that contains activation/deactivation of edges along time. We also proposed the TG-CSA, a compact data structure based on the Compact Suffix Array, which stores the graph as a sequence of 4-dimensional tuples. Hence, operations are answered as a pattern matching problem.

Then, we proposed two new strategies to represent temporal graphs, CAS and CET, that make use of compact and self-indexed data structures. In particular, CAS uses a Wavelet Tree and CET uses a Interleaved Wavelet Tree, where Interleaved Wavelet Tree is a novel compact and self-indexed data structure capable of representing sequences of a multidimensional alphabet. The main advantage of both strategies is that they are capable of checking the state of edges in logarithmic time, instead of performing a sequential search over the temporal log.

We finally proposed the Compressed k^d-tree (ck^d-tree), which is an improvement of the original k^d-tree. The main advantage of this structure is the efficient use of space, several times better than the worst case of the k^d-tree when input data is sparse and do not share any clustering property. It also guarantees, without considering any regularity, a space close to the information-theoretic lower bound for representing a multidimensional binary matrix with m active cells. Although, this new structure can be used for any d-dimensional space, it was specially adapted to represent temporal graphs in 4 and 3 dimensions. We designed some enhancements to improve both the compression and the time performance of the ck^d-tree.

We analyzed some temporal properties of special temporal graphs to improve the space and time performance of the proposed structures. For instance, taking into account that in *incremental* temporal graphs, once a contact becomes active, it remains active until the end of the lifetime of the graph, thus, it is not necessary to store the deactivation of the edge. Similarly, we took into account that the time interval in *point-contact* temporal graphs is always a time instant. In both cases, this allow us to reduce the sequences in CAS and CET, as well as to use a 3D representation for the ck^d-tree.

The analytical comparison and the experimental evaluation show that there is no a best

data structure to answer all types of queries on temporal graphs. When space is not a concern and there are few contacts per edge, all queries can be solved fast with the basic strategy based on a log of time intervals per edge, that is, with EdgeLog. Instead, if the space is of concern, CAS is a good candidate to answer direct neighbors and edge queries. For a good space-time trade-off for all queries, ck^{d}-tree and CET are the best candidates.

8.2 Future work

This section presents future lines of work we expect to address in order to continue the work that has been developed in this thesis. We describe for each contribution of the thesis, the most relevant possibilities to continue our work.

- In this thesis we proposed the TG-CSA, a compact data structure based on the Compressed Suffix Array of Sadakane [Sad02, Sad03]. The graph is represented as a sequence composed by the concatenation of 4-tuples, each of them representing a contact. As future work we would like to explore more encodings of Ψ (such as the one used in [FBN^{+}12b]), because it represents around 80-90% of the size of the TG-CSA. Also, we would like to explore the adaptation of other compressed self-indexes (e.g., FM-Index, LZ-index, among others) [NM07, FGNV09].

- We introduced an extension to the Wavelet Tree called Interleaved Wavelet Tree, which main goal is to represent sequences whose of multidimensional symbols. As a future work, we would like to explore the use of this structure in other areas such as compact binary relations [BCN13] and spatial indexes for point grids [BLNS09, FGN14].

- We proposed the ck^{d}-tree, which is an improvement in the space of the k^{d}-tree when cells in a multidimensional binary matrix are sparse. Thus, it can be used in other domains where cells or points do not share clustering properties. Regarding information in 2D, the ck^{d}-tree should be compared against a compressed representation of non-clustered quadtrees [GGNL^{+}15, FGN14], and against compact binary relations [BCN13]. In 3D, the structure can represent triples in RDF [GBBN14], non-clustered 3D regions, or even can store and query the evolution of raster data [dBR14].

- We introduced the bck^{d}-tree, which is an enhancement of the ck^{d}-tree that groups leaves into buckets. The main advantage of this strategy is a time performance improvement. However, this is accompanied by an increase of space usage. As a future work, we need to check if we are able to compress the data inside the buckets. Nowadays this is a current strategy in many industrial implementations of secondary memory B+-trees. Initial experiments reveal that the entropy of leaves in buckets could improve if we store them using xor-encoding [Hud09], but this may not be enough because we need to have a fast decompression schema to maintain the performance.

- We would like to explore techniques used to reduce the space on static graphs to temporal graphs. For instance, we can update the node ordering considering the time activation/deactivation of edges, which could be useful to improve the clustering of the cells in the matrix or the high-order entropy in the sequence representation. This mechanism has been useful for improving the compression and time for storing static

graphs in k^2-tree. Similarly, we can explore the compact representation of subgraphs (such as bicliques), that has been extensively used to improve the performance on social graphs [HN13], and to take into account that usually, there are some correlated spatio-temporal changes [CBL08] in some vertices of temporal graphs.

- In the evaluation chapter we notice that there is not a single structures that achieve the best compression ratio on all temporal graphs. As a future work we would like to study in what degree the dynamic properties of temporal graphs affect the compression ratio and the retrieval times of adjacency operations.

- The multidimensional representation used in TG-CSA and ck^d-tree opens the possibility to add more information to temporal graphs. For instance, they can be used for representing temporal multi graphs, where many overlapping time intervals can be related to an edge. Similarly, considering the temporal dimension as labels over edges, we could use all the data structures presented in this thesis for representing labeled graphs. For instance, we could label edges, as well as vertices, with some properties.

- We would also like to explore the usage of the structures presented in this thesis for the computation of temporal metrics based on spatio-temporal paths, or *journeys* [NTM$^+$13, TLS$^+$13, CFQS12]. As a future research line, we would like to develop compact data structures specifically designed for computing temporal metrics. For example, we could take into account that temporal paths [WCH$^+$14, HCW14] over two vertices share much information in common, as many vertices can be part of the same path. This idea has been explored with success for improving the computation of metrics over each snapshot of temporal graphs [RLK$^+$11].

- In this thesis we computed adjacency operations using one processor (i.e., CPU). As future work, we want to explore how to take advantage of parallelism and distributed computation environments, such as the work in [LBO$^+$14, HG13], in order to speed up the adjacency operations, as well as another complex queries over temporal graphs. We would also like to exploit the SIMD instructions to perform parallel arithmetic operations on the d-dimensions of the ck^d-tree and bit parallelism [GGNL$^+$15].

- With the structures we open new possibilities of encoding information in multiple dimensions. We would like to extend the work of ck^d-tree, TG-CSA and Interleaved Wavelet Tree to apply compression into another multidimensional data structures (e.g., *kd*-trees or range trees [BCKO08, Sam06]).

- We described all the representations of temporal graphs as static data structures. As future work, we would like to implement the dynamic version of the proposed data structures, considering that contacts can arrive along time. In the following subsection we sketch how dynamism could be handled for the proposed structures.

Dynamism The dynamism on temporal graphs can appear when considering that the lifetime is always growing. In this sense, the set of edges is also expected to grow along time, as well as, the set of vertices. Because the parity property still holds in the dynamic scenario, all data structures can be modified to meet this requirement. Modifications in EdgeLog are related with adding new neighboring vertices on the aggregated graph, and

keeping updated the list of time intervals of each edge. New edges and new time intervals need to keep the order of lists for the d-gaps compression. In EveLog, new neighboring changes should be added at the end of each adjacency log. The compression of the list of vertices depends on the frequency of each vertex along time, thus the dynamic version of ETDC [BFNP10] is required.

The dynamic solutions of CAS and CET rely on dynamic techniques developed to handle the growing length of the sequence and size of the alphabet in the Wavelet Tree [Mak12]. When the set of vertices is fixed, dynamism affects the size of the sequence S_{CET}, because new neighboring changes should be added at the end of the sequence. A varying length on S_{CET} can be handled by changing the static bitmaps by a dynamic version [Mak12] to the Interleaved Wavelet Tree. Dynamic bitmaps should be also used to deal with the growing size of B_{CET}. Changes on the set of vertices imply to increase the size of the alphabet in the Interleaved Wavelet Tree representing S_{CET}. As the height of the Interleaved Wavelet Tree is $2 \log n$, each time the size of the vertices is doubled, the Interleaved Wavelet Tree adds two new levels in the tree. Dynamism in CAS is more restricted because the alphabet in S_{CAS} combines vertices and time points. Therefore, only a growing lifetime is allowed by inserting symbols in the Wavelet Tree of S_{CAS} using dynamic bitmaps. Note that in this case, insertions can occur in the middle of the sequence representation used by CAS.

The dynamic version of the ck$^{\mathsf{d}}$-tree should manage creation and deletion of cells by using the same principles developed for the dynamic version of the k^2-tree [BdBN12]. The idea is to replace the bitmaps by a dynamic version [Mak12], in order to be able to insert and delete the three kind of nodes: white, gray and black. Assuming that insertions of cells are made one at a time, we need to manage two cases of node transformation: (i) a white leaf can be converted into a black leaf and (ii) a black leaf can be converted into a gray node. The deletion of a cells require to manage more cases. For instance, the deletion of a black leaf introduces an update on its parent, that can be converted into a black leaf (but it may require to update gray internal nodes upwards the tree). To manage the movements of black leaves, we also need to update the unary paths of isolated cells. Thus, the arrays A_l must be updated to a dynamic version [Mak12].

In terms of a temporal graph over a ck$^{\mathsf{d}}$-tree, we need to manage the growing size of the dimensions and create a special mark for contacts that remain active until the current time instant. For the growing size of the dimensions, we only need to add a new root (a new gray node) on top of the tree. The active contacts can be encoded as a tuple $(u, v, t, 0)$, where the 0 indicates that the contact is still active. Once a new neighboring change is added, the tuple must be updated to (u, v, t, t') to indicate the deactivation of the contact.

Appendix A

Publications and other research results

Publications

Journals

- D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," Information Systems, vol. 51, pp. 1–26, Jul. 2015.
- D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Compressed k^{d}-tree for temporal graphs,". To be submitted to Knowledge and Information Systems.
- N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez, "A Compressed Suffix-Array Strategy for Temporal-Graph Indexing". Work in progress.

International Conferences

- N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez, "A Compressed Suffix-Array Strategy for Temporal-Graph Indexing," presented at the 21st International Symposium on String Processing and Information Retrieval, Ouro Preto, Brazil, 2014, vol. 8799, pp. 77–88.
- G. D. Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodriguez, "Compact Data Structures for Temporal Graphs," presented at the Data Compression Conference (DCC), 2013, p. 477.

International Workshops

- D. Caro, "A compressed hexatree for temporal-graph indexing... or how to compress the k^4-tree," presented at the SPIRE 2014 Workshop on Compression, Text, and Algorithms (WCTA 2014), Ouro Preto, Brazil.

Research stays

- November 2012, Research stay at the French Naval Academy Research Institute (IRENav in Brest, France) under supervision of Prof. Christophe Claramunt.
- November 2012, January-March 2014, Research stay at the Database Lab (LBD) of the University of A Coruña (Spain) under supervision of Prof. Nieves R. Brisaboa.

Bibliography

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, Jan 2002.

[AD09] Alberto Apostolico and Guido Drovandi. Graph Compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[ÁGBFMP11] Sandra Álvarez-García, Nieves R. Brisaboa, Javier D. Fernández, and Miguel A Martínez-Prieto. Compressed k2-Triples for Full-In-Memory RDF Engines. In *AMCIS*, 2011.

[AM04] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *ADC '04: Proceedings of the 15th Australasian database conference*. Australian Computer Society, Inc, January 2004.

[Ama15] Amazon.com. Amazon elastic compute cloud (amazon ec2). `http://aws.amazon.com/ec2/`, 2015.

[AS99] Srinivas Aluru and Fatih E Sevilgen. Dynamic Compressed Hypertoctrees with Application to the N-Body Problem. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, December 1999.

[AZAL04] Eytan Adar, Li Zhang, Lada A. Adamic, and Rajan M. Lukose. Implicit structure and the dynamics of blogspace. *Workshop on the Weblogging Ecosystem*, 13(1), 2004.

[BAhM12] Jérémy Barbay, Luca Castelli Aleardi, Meng he, and J. Ian Munro. Succinct Representation of Labeled Graphs. *Algorithmica*, 62(1-2), February 2012.

[BCFR14] Nieves R. Brisaboa, Diego Caro, Antonio Fariña, and Andrea Rodríguez. A Compressed Suffix-Array Strategy for Temporal-Graph Indexing. In Edleno Moura and Maxime Crochemore, editors, *Lecture Notes in Computer Science*, pages 77–88. Springer International Publishing, 2014.

[BCGJ11] Francesco Bonchi, Carlos Castillo, Aristides Gionis, and Alejandro Jaimes. Social Network Analysis and Mining for Business Applications. *ACM Trans. Intell. Syst. Technol.*, 2(3):22:1–22:37, 2011.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[BCN13] Jérémy Barbay, Francisco Claude, and Gonzalo Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19 – 37, 2013.

[BdBN12] Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Compressed Dynamic Binary Relations. In *Data Compression Conference (DCC), 2012*, pages 52–61, 2012.

[BDM+05] David Benoit, Erik D Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BFNP07] Nieves R. Brisaboa, Antonio Fariña, Gonzalo Navarro, and José R Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[BFNP10] Nieves R. Brisaboa, Antonio Fariña, Gonzalo Navarro, and José R. Paramá. Dynamic lightweight text compression. *ACM Transactions on Information Systems (TOIS)*, 28(3), 2010.

[BGIMSR07] Jérémy Barbay, Alexander Golynski, J Ian Munro, and S Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, November 2007.

[BHK85] F. Warren Burton, Matthew M. Huntbach, and John G. Kollias. Multiple generation text files using overlapping tree structures. *The Computer Journal*, 28(4):414–416, 1985.

[BKMK90] F. Warren Burton, John G. Kollias, D. G. Matsakis, and V. G. Kollias. Implementation of overlapping b-trees for time and space efficient representation of collections of similar files. *The Computer Journal*, 33(3):279–280, 1990.

[BLN09] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-Trees for Compact Web Graph Representation. In *SPIRE '09: Proceedings of the 16th International Symposium on String Processing and Information Retrieval*. Springer-Verlag, August 2009.

[BLN13] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management: an International Journal*, 49(1), January 2013.

[BLN14] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[BLNS09] Nieves R. Brisaboa, Miguel Rodríguez Luaces, Gonzalo Navarro, and Diego Seco. A new point access method based on wavelet trees. In *Proceedings of the International Workshop on Semantic and Conceptual Issues in GIS (SeCoGIS)*, volume 5833 of *Lecture Notes in Computer Science*, pages 297–306. Springer, 2009.

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

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

[BM99] Andrej Brodnik and J. Ian Munro. Membership in Constant Time and Almost-Minimum Space. *SIAM Journal on Computing*, 28(5), May 1999.

[BWJ98] Claudio Bettini, X.Sean Wang, and Sushil Jajodia. A general framework for time granularity and its application to temporal reasoning. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):29–58, 1998.

[BXFJ03] Binh-Minh Buin-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003.

[CARB15] Diego Caro, M Andrea Rodríguez, and Nieves R. Brisaboa. Data structures for temporal graphs based on compact sequence representations. *Information Systems*, 51:1–26, July 2015.

[CBHS05] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. *Proceedings of the 14th international conference on World Wide Web*, pages 613–622, 2005.

[CBL08] Jeffrey Chan, James Bailey, and Christopher Leckie. Discovering correlated spatio-temporal changes in evolving graphs. *Knowledge and Information Systems*, 16(1), July 2008.

[CFQS11] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. In *Proceedings of the 10th international conference on Ad-hoc, mobile, and wireless networks*, pages 346–359, Berlin, Heidelberg, 2011. Springer-Verlag.

[CFQS12] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-Varying Graphs and Dynamic Networks. *arXiv.org*, cs.DC, February 2012.

[Cha86] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, 1986.

[Cha88] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[CIK+12] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, M. Sohel Rahman, German Tischler, and Tomasz Walen. Improved algorithms for the range next value problem and applications. *Theoretical Computer Science*, 434:23–34, 2012.

[Cla83] Kenneth L Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 226–232. IEEE, 1983.

[Cla96] David Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, January 1996.

[CM07] J. Shane Culpepper and Alistair Moffat. Compact set representation for information retrieval. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 4726 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2007.

[CMG09] Meeyoung Cha, Alan Mislove, and P. Krishna Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *International World Wide Web Conference (WWW)*, pages 721–730. ACM, 2009.

[CN08] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5280 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2008.

[CN10a] Francisco Claude and Gonzalo Navarro. Extended Compact Web Graph Representations. In T Elomaa, H Mannila, and P Orponen, editors, *Algorithms and Applications (Ukkonen Festschrift)*, pages 77–91. Springer, 2010.

[CN10b] Francisco Claude and Gonzalo Navarro. Fast and Compact Web Graph Representations. *ACM Trans. Web*, 4(4):16:1–16:31, 2010.

[CN10c] Francisco Claude and Gonzalo Navarro. Self-Indexed Grammar-Based Compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[CN12] Francisco Claude and Gonzalo Navarro. The wavelet matrix. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2012.

[CNO15] Francisco Claude, Gonzalo Navarro, and A Ordóñez. The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets. *Information Systems*, 47:15–32, 2015.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of Information Theory*. John Wiley & Sons, November 2012.

[dBÁGB+13] Guillermo de Bernardo, Sandra Álvarez-García, Nieves R. Brisaboa, Gonzalo Navarro, and Oscar Pedreira. Compact querieable representations of raster data. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 8214 of *Lecture Notes in Computer Science*, pages 96–108. Springer, 2013.

[dBBCR13] Guillermo de Bernardo, Nieves R. Brisaboa, Diego Caro, and M. Andrea Rodríguez. Compact data structures for temporal graphs. In *Data Compression Conference (DCC)*, page 477. IEEE, 2013.

[dBR14] Guillermo de Bernardo Roca. *New data structures and algorithms for the efficient managementof large spatial datasets*. PhD thesis, Universidade da Coruña, December 2014.

[DEGI10] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. *Algorithms and theory of computation handbook*, pages 9–9, 2010.

[DM11] M Dehmer and A Mowshowitz. A history of graph entropy measures. *Information Sciences*, 2011.

[dMNZBY00] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *Transactions on Information Systems (TOIS*, 18(2), April 2000.

[EGS05] David Eppstein, Michael T Goodrich, and Jonathan Z Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*. ACM Request Permissions, June 2005.

[FB74] Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[FBN+12a] A. Fariña, N. Brisaboa, G. Navarro, F. Claude, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM TOIS*, 30(1):article 1, 2012.

[FBN+12b] Antonio Fariña, Nieves R. Brisaboa, Gonzalo Navarro, Francisco Claude, Ángeles S Places, and Eduardo Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems*, 30(1):1–34, February 2012.

[Fer02] Afonso Ferreira. On models and algorithms for dynamic communication networks: The case for evolving graphs. *In Proc ALGOTEL*, 2002.

[FGN14] Arash Farzan, Travis Gagie, and Gonzalo Navarro. Entropy-bounded representation of point grids. *Computational Geometry: Theory and Applications*, 47(1), January 2014.

[FGNV09] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13:12, 2009.

[FMMN07] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20, 2007.

[FV02] Afonso Ferreira and Laurent Viennot. A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Technical report, MASCOTTE - INRIA Sophia Antipolis / Laboratoire I3S , HIPERCOM - INRIA Rocquencourt, 2002.

[Gar82] I Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, pages 1–6, 1982.

[Gar14] Sandra Alvarez Garcia. *Compact and Efficient Representations of Graphs*. PhD thesis, Universidade da Coruña, December 2014.

[GBBN14] Sandra Alvarez Garcia, Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Interleaved K2-Tree: Indexing and Navigating Ternary Relations. In *2014 Data Compression Conference (DCC)*, pages 342–351. IEEE, 2014.

[GBMP14a] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the 13th International Symposium on Experimental Algorithms*. Springer-Verlag New York, Inc, June 2014.

[GBMP14b] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[GBYR15] Venkat N Gudivada, Ricardo Baeza-Yates, and Vijay V Raghavan. Big Data: Promises and Problems. *Computer*, 48(3):20–23, 2015.

[GBYS92] Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. *Information Retrieval*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[GGMN05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)(Greece, 2005)*, pages 27–38, 2005.

[GGNL+15] T Gagie, J González-Nova, S Ladra, Gonzalo Navarro, and D. Seco. Faster Compressed Quadtrees. In *Proc. 25th Data Compression Conference (DCC)*, pages 93–102, 2015.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, January 2003.

[GHV05] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *ESWC'05: Proceedings of the Second European conference on The Semantic Web: research and Applications*. Springer-Verlag, May 2005.

[GHV07] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Introducing time into RDF. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, 2007.

[GNP12] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, April 2012.

[Gog11] Simon Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, University of Ulm, November 2011.

[Gol66] S Golomb. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[GP13] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, May 2013.

[GRLK12] Manuel Gomez-Rodriguez, Jure Leskovec, and Andreas Krause. Inferring Networks of Diffusion and Influence. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(4):21:1–21:37, 2012.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 14(2):47–57, June 1984.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *the thirty-second annual ACM symposium*, pages 397–406, New York, New York, USA, 2000. ACM Press.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[HCW14] Silu Huang, James Cheng, and Huanhuan Wu. Temporal Graph Traversals: Definitions, Algorithms, and Applications. *arXiv.org*, January 2014.

[HG13] Imranul Hoque and Indranil Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *the First ACM SIGOPS Conference*, pages 1–17, New York, New York, USA, 2013. ACM Press.

[HN13] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and Information Systems (KAIS)*, pages 1–35, 2013.

[Hol08] Lawrence B. Holder. Graph-based Temporal Mining of Metabolic Pathways with Microarray Data. In *ACM SIGKDD Workshop on Data Mining in Bioinformatics (BIOKDD'08)*, August 2008.

[HS12] Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, 519(3):97–125, 2012.

[Hud09] Beno it Hudson. Succinct Representation of Well-Spaced Point Clouds. *Audio, Transactions of the IRE Professional Group on*, pages –, September 2009.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the IRE*, pages 1098–1101, 1952.

[Jac89a] Guy Jacobson. Space-efficient static trees and graphs. *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554, 1989.

[Jac89b] Guy Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.

[KD13] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *International Conference on Data Engineering (ICDE)*, pages 997–1008. IEEE Computer Society, 2013.

[KM97] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 107–121, 1997.

[Kun13] Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web Companion*, WWW '13 Companion, pages 1343–1350, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13), November 1999.

[Lab14] Yahoo! Labs. Yahoo! network flows data, version 1.0. `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`, 2014.

[LBO+14] Alan G Labouseur, Jeremy Birnbaum, Paul W Olsen, Sean R Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, March 2014.

[LM00] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. In *Proceedings of the IEEE*, pages 1722–1732, 2000.

[LOH05] JeongKyu Lee, JungHwan Oh, and Sae Hwang. STRG-Index: spatio-temporal region graph indexing for large video databases. In *the 2005 ACM SIGMOD international conference*, page 718, New York, New York, USA, 2005. ACM Press.

[LOH13] Alan G Labouseur, Paul W Olsen, Jr, and Jeong-Hyon Hwang. Scalable and Robust Management of Dynamic Graph Data. *The VLDB Journal The International Journal on Very Large Data Bases*, pages 1–6, 2013.

[Lue78] George S. Lueker. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 28–34. IEEE Computer Society, 1978.

[Mak12] Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012.

[Man01] Giovanni Manzini. An analysis of the Burrows—Wheeler transform. *Journal of the ACM (JACM)*, 48(3), May 2001.

[McC76] Edward M McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[MD12] Abbe Mowshowitz and Matthias Dehmer. Entropy and the Complexity of Graphs Revisited. *Entropy*, 14(12):559–570, December 2012.

[MHN84] Takashi Matsuyama, Le Viet Hao, and Makoto Nagao. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing*, 26(3):303–318, 1984.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN06] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.

[MR97] J I Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. *Foundations of Computer Science*, 1997.

[MR01] J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, January 2001.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008.

[Mun96] J. Ian Munro. Tables. In V Chandru and V Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer Berlin / Heidelberg, 1996.

[Nav07] Gonzalo Navarro. Compressing Web Graphs like Texts. Technical Report TR/DCC-2007-2, Department of Computer Science, University of Chile, 2007.

[Nav12] Gonzalo Navarro. Wavelet Trees for All. *Combinatorial Pattern Matching*, 2012.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *Computing Surveys (CSUR*, 39(1):2, April 2007.

[NP12] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *SEA'12: Proceedings of the 11th international conference on Experimental Algorithms*. Springer-Verlag, June 2012.

[NR08] Gonzalo Navarro and Luís Russo. Re-pair Achieves High-Order Entropy. In *Data Compression Conference, 2008. DCC 2008*, page 537, 2008.

[NST98] Mario A. Nascimento, Jefferson R O Silva, and Yannis Theodoridis. Access structures for moving points. Technical Report TR-33, 1998.

[NST99] Mario A. Nascimento, Jefferson R O Silva, and Yannis Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 171–188, London, UK, UK, 1999. Springer-Verlag.

[NTM+13] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal Networks*, Understanding Complex Systems, pages 15–40. Springer Berlin Heidelberg, 2013.

[Pag99] Rasmus Pagh. Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, July 1999.

[PJT00] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 395–406. Morgan Kaufmann, 2000.

[PUS08] Andrea Pugliese, Octavian Udrea, and V S Subrahmanian. Scaling RDF with Time. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*. ACM, April 2008.

[RBR12] Miguel Romero, Nieves R. Brisaboa, and M. Andrea Rodríguez. The smo-index: a succinct moving object structure for timestamp and interval queries. In *ACM Sigspatial International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS)*, pages 498–501. ACM, 2012.

[RLK+11] Chenghui Ren, Eric Lo, Ben Kao, Xiaojin Zhu, and Reynold Cheng. On Querying Historical Evolving Graph Sequences. *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB 2011)*, 4(11), 2011.

[RP71] R Rice and J Plaunt. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communications*, 19(6):889–897, December 1971.

[RRR02] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.

[RRS07] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), November 2007.

[RS06] J Ian Munro Rao and S Srinivasa. Succinct Representation of Data Structures. In *Handbook of DATA STRUCTURES and APPLICATIONS*, pages 1–22. Chapman and Hall/CRC, September 2006.

[Sad00] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, volume 1969 of *Lecture Notes in Computer Science*, pages 410–421. Springer, 2000.

[Sad02] Kunihiko Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. pages 410–421. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2002.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of algorithms*, 48(2):294–313, September 2003.

[Sam06] Hanan Samet. *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann, 2006.

[SB15] Diego Seco and Jeremy Barbay. Personal communication, 2015.

[SG06] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. ACM Request Permissions, January 2006.

[Sha48] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal, The*, 27(3):379–423, 1948.

[Sim95] G Simonyi. Graph entropy: a survey. *Combinatorial Optimization*, 1995.

[SN09] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *SODA '10: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–16. Society for Industrial and Applied Mathematics, October 2009.

[SO07] Kunihiko Sadakane and Daisuke Okanohara. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[SQF+11] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frederic Amblard. Time-Varying Graphs and Social Network Analysis: Temporal Indicators and Metrics. In *3rd AISB Social Networks and Multiagent Systems Symposium (SNAMAS)*, pages 32–38, April 2011.

[ST99] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.

[SWYZ02] Falk Scholer, Hugh E Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes For fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM Request Permissions, August 2002.

[TB09] Jonas Tappolet and Abraham Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, pages 308–322, Berlin, Heidelberg, 2009. Springer-Verlag.

[TL10] Lei Tang and Huan Liu. Graph mining applications to social network analysis. *Managing and Mining Graph Data*, pages 487–513, 2010.

[TLS+13] John Tang, Ilias Leontiadis, Salvatore Scellato, Vincenzo Nicosia, Cecilia Mascolo, Mirco Musolesi, and Vito Latora. Applications of temporal graph metrics to real-world networks. In *Temporal Networks*, Understanding Complex Systems, pages 135–159. Springer Berlin Heidelberg, 2013.

[TP01] Yufei Tao and Dimitris Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *International Conference on Very Large Databases (VLDB)*, pages 431–440. Morgan Kaufmann, 2001.

[WCH+14] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path Problems in Temporal Graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Automata Theory*, pages 1–11. IEEE, 1973.

[WF90] David S Wise and John Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9(3):282–296, 1990.

[WMB99] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing Gigabytes*. Compressing and Indexing Documents and Images. Morgan Kaufmann, 1999.

[XHL90] Xiaomei Xu, Jiawei Han, and Wei Lu. RT-Tree: An Improved R-Tree Index Structure for Spatiotemporal. *Proceedings of the 4th Inter. Sym. on Spatial Data Handling*, 2:1040–1049, 1990.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE Computer Society, April 2006.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*. ACM, April 2008.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2), July 2006.