Noname manuscript No.
(will be inserted by the editor)

Compressed k^d-tree for temporal graphs

Diego Caro · M. Andrea Rodríguez · Nieves R. Brisaboa · Antonio Fariña

the date of receipt and acceptance should be inserted later

Abstract Temporal graphs represent vertices and binary relations that change along time. The work in this paper proposes 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 (worst case) 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 and space. An experimental evaluation compares the ck^d-tree with k^d-tree (the d-dimensional extension of the k^2-tree) and with other up-to-date compressed data structures based on inverted indexes and Wavelet Trees, showing the potential use of the ck^d-tree for different types of temporal graphs.

Keywords multidimensional compact data structure · compact data structures for temporal graphs · time-varying graphs · evolving graphs

1 Introduction

A temporal graph is a graph whose connectivity between vertexes changes along time (i.e., edges can appear and disappear). They are useful for modeling data such as the evolution of friendship relations when a user adds or removes friends in online social networks, the dynamism of citation networks when new scientific articles are published, the time-varying connectivity between mobile devices when they change their base station, or the changes of links that appear or disappear in the Web graph. Using temporal graphs, it is possible to get not only the current, but also the historical state of the connectivity between vertexes. Formally, we consider a temporal graph as a set of contacts between vertices, indicating the time interval when the vertices are connected [NTM+13].

A temporal graph can be represented as several static graphs (or snapshots), storing the active edges for each time point[1] in the lifetime of the graph [FV02]. Main issue with this strategy is the use of space even if edges remain in the same state (active or inactive) for a long time. A strategy to overcome the space issue is to store differences between some snapshots (carefully chosen), but at the expenses of processing the differences at running time [RLK+11][dBBCR13][LBO+14]. Other strategies are based on storing events of activation or deactivation along time. With these methods, the state of an edge is recovered by counting how many events on the queried edge occur: an even number of times means that the edge is active, or inactive otherwise [CARB15].

Using a multidimensional approach, contacts can be seen as 4-tuples [BCFR14] represented with a 4D binary matrix. For 2-tuples, the k^2-tree [BLN14] achieves good results over Web graphs, binary relations, and raster data, because it is capable to represent in a small space clustered data. The k^d-tree [dBÁGB+13, dBR14] is the multidimensional version of the k^2-tree. Thus, a k^4-tree is the 4-tuples version of the k^d-tree [dBR14], which

D. Caro · M. A. Rodríguez
Computer Science Department, University of Concepción, Chile
E-mail: diegocaro@udec.cl; andrea@udec.cl

N. R. Birsaboa · A. Fariña
Database Lab, Facultade de Informática, University of A Coruña, Spain
E-mail: brisaboa@udc.es; fari@udc.es

[1] We use the term time instant and time point indistinctly.

shows to be inappropriate to compress temporal graphs that do not exhibit the clustering properties found for 2-tuples.

The work in this paper proposes modifications to the k^d-tree [dBÁGB+13, dBR14] with the aim to reduce the use of space for unclustered data while keeping good time performance. In particular, it proposes 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. We concentrate on temporal adjacency operations over nodes and edges (i.e., retrieval of direct/reverse neighbors, check if an edge is active, or obtain a snapshot of the graph), constrained to a specific time point or to a time interval.

The organization of this paper is as follows. Section 2 introduces main concepts useful to explain the proposed structures and summarizes related work. Section 3 presents the ck^d-tree, covering the description of the structure, its basic access operations, and its analytical cost in space. Section 4 shows how to represent temporal graphs with ck^d-tree and how the operations on temporal graphs can be solved as range search queries on ck^d-tree. Further improvements of ck^d-tree are given in Section 5. Section 6 provides the experimental evaluation of the proposed structures and its variants with respect to baselines and state-of-art structures using diverse types of temporal graphs. Finally, Section 7 gives conclusions and future research directions.

2 Preliminary concepts and related work

In this section, we introduce the definition of temporal graphs adopted in this work, summary existing structures for temporal graphs, and revise in more detail structures that are closely related to the structures proposed in this paper.

2.1 Temporal Graphs

Formally, a temporal graph is a set $\mathcal{C}$ of contacts between a set of vertexes V during a set of time points $\mathcal{T}$ (with total order) representing the *lifetime* of the graph. A contact of an edge $(u, v) \in E \subseteq V \times V$ is a quadruplet $c = (u, v, t, t')$, where $[t, t') \subset \mathcal{T} \times \mathcal{T}$ is the time interval when the edge (u, v) is active [NTM+13]. We denote $n = |V|$ the number of vertices, $m = |E|$ the number of edges, $\tau = |\mathcal{T}|$ the number of time points in the lifetime of the graph, and $c = |\mathcal{C}|$ the number of contacts. As Nicosia *et al.* [NTM+13] claim, we assume that there are no empty or overlapping intervals for a given edge. This is, given two contacts $(u, v, t_i, t'_i), (u, v, t_j, t'_j) \in C$, then $t_i < t_j$ if and only if $t'_i < t_j$. We call this temporal property of edges the *temporal constraint*. We named *aggregated graph* to the static graph composed by the set of edges E in the temporal graph. Figure 1 shows an example of a temporal graph.

In what follows, we will say that a contact (u, v, t, t') is active at a time point t_q, if $t_q \in [t, t')$, and it is active at a time interval $[t_q, t'_q)$, if $[t_q, t'_q)$ lays during $[t, t')$.

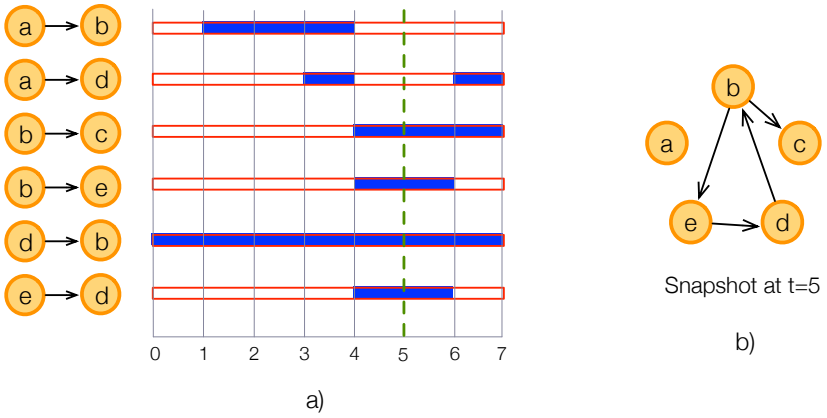

Fig. 1 Example of a temporal graph and a snapshot: a) a set of contacts among five nodes. b) the snapshot of active edges at time $t = 5$ corresponds to the dashed line on the set of contacts (Figure adapted from [NTM+13]).

Class	Operations defined for a time point
About vertices	DirectNeighbors(u,t): returns the adjacent active neighbors of u at a given time point t.
	ReverseNeighbors(v,t): gives the active reverse adjacent vertices of v at time t.
About edges	Edge$((u,v),t)$: true if the edge (u,v) is active at t, false otherwise.
	EdgeNext$((u,v),t)$: returns the instant of the next activation of (u,v) after t, or t if it is active; otherwise returns ∞.
About the graph	Snapshot(t): returns all active edges at a time point t.
	ActivatedEdges(t): return all edges that were activated at time point t.
About events	DeactivatedEdges(t): return all edges that were deactivated at time point t.
	ChangedEdges(t): return all edges that were activated or deactivated at time point t.

Table 1 Basic operations over a temporal graph constrained by a time point.

The operations over temporal graphs extend queries over static graphs with a temporal criteria [CARB15]. The queries can be divided into four classes: (1) queries about vertices, retrieving the active direct/reverse neighbors of a vertex, (2) queries about edges, checking if an edge is either active or inactive or retrieving the next activation time of an edge, (3) queries about the graph, retrieving the active or inactive state of all edges, and (4) queries about events/changes on edges, recovering the edges that were activated/deactivated or both. These queries can be constrained to a time point or time interval. For queries about vertices and edges in a time interval $[t,t')$, we consider two possible semantics: (1) a *strong* semantics, which retrieves contacts that are active during the interval $[t,t')$, and (2) a *weak* semantics, which retrieves contacts that occur within the interval $[t,t')$. Table 1 shows the definition of operations for a time point and Table 2 shows an example of operations on the graph in Figure 1a. For a detailed review on queries over temporal graphs see [CARB15, pp. 3–5].

Operation	Point $t_q = 1$	Interval $t_q = [3,5)$	
		Weak Sem.	Strong Sem.
DirectNeighbors of a	$\{b\}$	$\{b,d\}$	$\{\}$
ReverseNeighbors of b	$\{a,d\}$	$\{a,d\}$	$\{d\}$
Edge (a,b)	True	True	False
EdgeNext of (a,d)	3	-	
Snapshot	{(a,b), (d,b)}	-	
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 2 Examples of basic operations over the temporal graph in Figure 1. Table extracted from [CARB15].

Temporal graphs can be classified by the duration of its contacts [HS12]. If the duration of the contacts is always a time point, e.g., of the form $(u,v,t,t+1)$ with $(u,v) \in E$ and $t \in \mathcal{T}$, we say that the graph is a *point-contact* temporal graph[2];otherwise, we say that it is an *interval-contact* temporal graph. Temporal graphs can be also classified by its dynamism [DEGI10]. We say that a temporal graph is *fully-dynamic* if the contacts of any of its edges occur at many time points. 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, the *weak* semantics over a query 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 right-endpoint t' of the query interval gets all the active contacts. The same does the *strong* semantics, which can be answered by a time-point query over t. 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 and the activated edges at t are also the edges that are active at t. Similarly, DeactivatedEdges for a time instant t can be computed as Snapshot$(t-1)$, because the deactivated edges at time t are those that were activated at $t-1$.

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

2.2 Existing structures for temporal graphs

The simplest methods for representing temporal graphs are based on storing snapshots (or several static graphs) with the active edges for each time instant in the lifetime of the graph. An example of this representation is the *presence matrix* [FV02, p. 3], a binary matrix of size $m \times \tau$, where each cell (i, j) indicates the activation state of edge i at the time instant j. The main issue with this strategy is the excessive use of space for edges that remain in the same state (active/inactive) for long periods of time. This is due a new cell is required, although the state of a the following time instant is the same as the current time instant.

The *G* database* [LBO+14] is a distributed index based on the snapshot representation that solves the space issue by storing a new version of edges only when a new state is achieved. They keep, for each time instant, a pointer to the current active edges, represented by a set of adjacency lists. If a edge become active in next time instant they create a new adjacency list for storing the new edge. Then they update the time instant pointer to this new adjacency list, but also keep a pointer to the adjacency lists of the last instant. If none edge change its state in the next time instant, they point to the last time instant. Although this mechanism saves space, do not work very well if after some changes the current snapshot is exactly the snapshot before the last changes. The DeltaGraph [KD13] is also a distributed index that groups graphs in a hierarchical structure based on common edges. It is a similar idea, in the sense that store different version of edges, but it is capable to take into a account that future snapshots can be exactly the same occurred some changes before.

Another strategy is to store the temporal graph as a *log of events* [FV02, p. 6]. This *log* stores the events of activation/deactivation of edges along time, and contrary to the snapshot representation, the space usage depends on the number of contacts related to each edge. However, the main issue with this approach is the time required to obtain the state of an edge, as it is obtained by a sequential traversal of the events related to an edge.

Some attempts have been made to improve the time taking to process a *log of events*. The simplest strategy is to combine logs with snapshots. These snapshots act like a sampling of the state of the graph at some time instants. The FVF-Framework [RLK+11] obtain a set of representative snapshots through a clustering processing. Then, for each time instant, the store what changes with respect to the most similar representative snapshots. However, they are focused on solving other than adjacency queries over time.

The research on temporal graph compression is still in an early stage. Here, the space of the compressed strategies is measured in bits per contact (bpc), that is, the number of bits needed to have the graph in main memory divided by the number of contacts (time intervals when an edge was active).

A strategy based on adjacency lists have been proposed in [XFJ03]. The idea is to represent the graph as a set of adjacency lists, but for each neighbor, it stores a list of time intervals indicating when the edge was active. The compressed version of this idea is the EdgeLog, presented in [CARB15, p. 9]. Here, both adjacency lists and time intervals are stored with gap encoding. Although this strategy is very fast, it does not compress very well, and it also requires an external structure to retrieve reverse neighbors.

The Adjacency Log (EveLog) [CARB15, p. 10] is a compressed structure based on *log of events* idea. The EveLog models the *log* as two separated lists per vertex, one for representing the time instants and the other one for representing the edges related to the event. The list of time points is compressed with gap encoding, and the list of edges is compressed with a statistical model. The state of an edge is obtained by a sequential scanning of the adjacency log related to an edge. The issue with this approach is related with the time that takes the sequential scanning of the *log* used to retrieve the state of edges.

In [dBBCR13] both snapshots and logs were compressed in k^2-trees [BLN14]. The log is modeled as a differential, representing the edges that change between two consecutive time instants. The time performance of this strategy was improved with the ik^2-tree [GBBN14], which corresponds to a special arrangement of the bitmaps of each k^2-tree. This arrangement of bitmaps allows the retrieval of the state of an edge by counting how many times the edge appears in the log. Assuming that all edges are inactive at the beginning of time, 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. Thus, if it appears an even number of times, it means that the edge is inactive, otherwise it is active. The benefits of the ik^2-tree are its reduced space and the ability of answering reverse neighbors (predecessors of a vertex) using the same space. But, as we will show in the experimental section, the retrieval time depends on how many events are required to obtain the state of an edge.

The work in [CARB15] presents another strategy to solve the problem of processing a *log*, which is based on a fast counting method to retrieve the state of edges. They proposed CAS, a compact data structure based on a sequence composed by the concatenation of the *log of events* of all vertices. To retrieve the state of an edge, they also count how many times the edge appears in the log of events. For a fast counting, they use the Wavelet Tree [GGV03], a data structure capable to retrieve the frequency of apparitions of a symbol in logarithmic time, regardless the size of the sequence. The authors also proposed CET, which also models the *log of events* as a sequence ordered by time (instead of by vertex). They develop an extension of the Wavelet Tree for representing

sequences of multidimensional symbols, called Interleaved Wavelet Tree. The Interleaved Wavelet Tree allows the retrieval of the frequency of symbols in any component of a multidimensional symbol in logarithmic time. The state of edges is also recovered by counting how many times an edge appear. Although both structures are capable to answer reverse neighbors using the same space, only CET is capable to answer reverse and direct neighbors with the same time cost.

Brisaboa *et al.* [BCFR14] propose a structure based on the Compressed Suffix Array (CSA) for large alphabets [Sad03, FBN+12]. The temporal graph is represented as a sequence of 4-tuples, each of them representing a contact. The sequence is composed of four different alphabets, one per each component of the 4-tuple. These four disjoint alphabets produce a nice property used to compress the CSA, the first quarter of the suffix array always points to symbols in the second quarter, the second quarter always points to elements in the third quarter, and so on. To retrieve the state of an edge, they search a pattern composed by the edge. Then, they check if there is at least one contact that is active at the query time. This strategy requires more space than CAS and CET, but it allows the retrieval of direct and reverse neighbors with the same time cost.

2.3 Multidimensional compact data structures

A contact can be naturally represented as a cell in a 4D binary matrix, with two dimensions encoding edges and the others two encoding time intervals. In this section we review some compressed representation of binary matrices with two or more dimensions that can be used to represent temporal graphs. We start with the k^2-tree [BLN14, BLN09], which is 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], RDF databases [ÁGBFMP11] and temporal graphs [dBBCR13]. We also review 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.

2.3.1 The k^2-tree

Conceptually, the k^2-tree [BLN14, BLN09] corresponds to a MX Quadtree [Sam06] with a succinct representation of the tree shape. The tree is encoded using the generalized Jacobson's level-order [Jac89] [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 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[3]. 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). 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.

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

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.

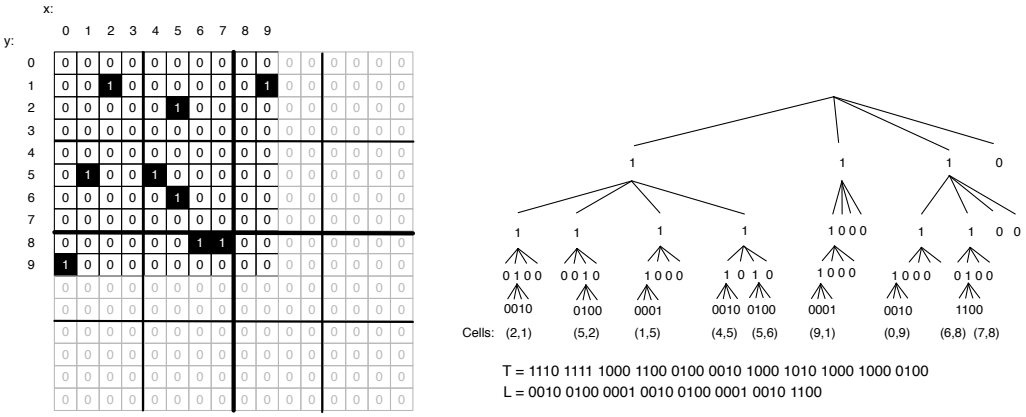

Fig. 2 A k^2-tree representation for a 10×10 binary matrix.

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. 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 zeroes (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.

2.3.2 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[4]. 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 3 shows an example of k^d-trees with $k = 2$.

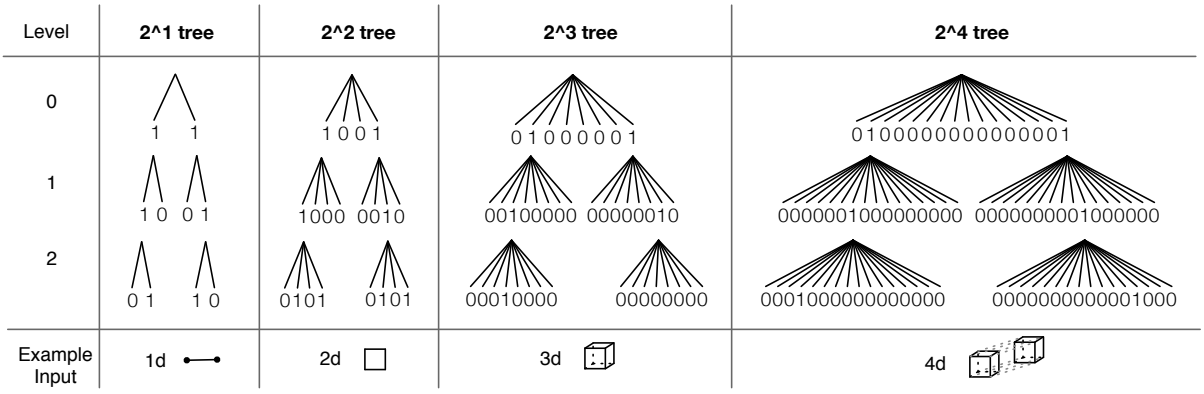

Fig. 3 Example of k^d-trees for 1, 2, 3 and 4 dimensions with $k = 2$. We include an example of the input: points in a line for 1D, cells in a square matrix for 2D, cells in a cube matrix 3D, and cells in a tesseract for 4D.

The 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 the cell. If the j-th bit is set to 1, it traverses down the k^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^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^d-tree, its space increases exponentially with the number of dimensions.

As the k^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.

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

2.3.3 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 4 shows an example of a ik^2-tree with $k = 2$ and three z_i values.

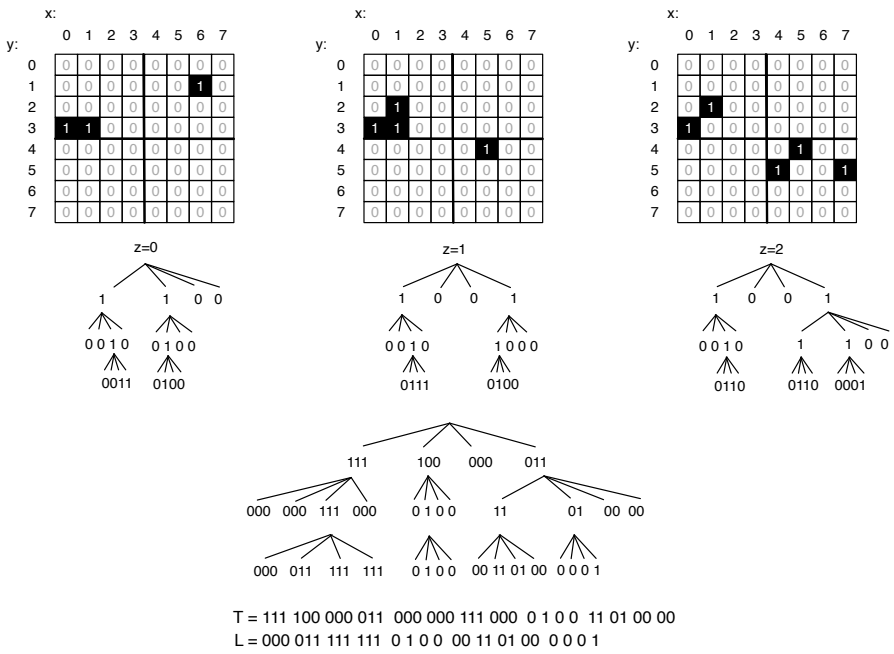

Fig. 4 An ik^2-tree representation of three binary matrices. Each binary matrix is encoded as a k^2-tree, which represents triples with a fixed value for the z component. The ik^2-tree is composed by the interleaved bits of the same branches of the three k^2-tree.

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$^\mathsf{d}$-tree outperforms in both space and time the ik^2-tree [dBR14].

2.4 k^2-trees for temporal graphs

The first approach to storing temporal graphs in k^2-trees is the differential k^2-tree [dBBCR13]. It 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.

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. Álvarez *et al.* [GBBN14, Gar14, dBR14] used this 3D encoding to store temporal graphs using the Interleaved k^2-tree. 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 $t \in [0, t_k]$. 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$^\mathsf{d}$-tree and how to deal with the sparseness due the d-dimensional space.

3 The Compressed k$^\mathsf{d}$-tree (ck$^\mathsf{d}$-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]):

$$\mathcal{H} = \log \binom{n^4}{m} \leq m \log \frac{n^4}{m} + O(m) \tag{1}$$

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 paper 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, where pointers are replaced by bitmaps. The succinct representation of the tree follows the generalized Jacobson's level-order [Jac89] [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^d-tree relies on the k^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.

3.1 Encoding the tree

The first step to compress the k^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^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^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.

3.2 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 $(p_1 - u_1, p_2 - u_2, ..., p_d - u_d)$. For example, the cell $(9, 1)$ in Figure 5 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^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 5 shows the compressed version of the k^2-tree in Figure 2.

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.

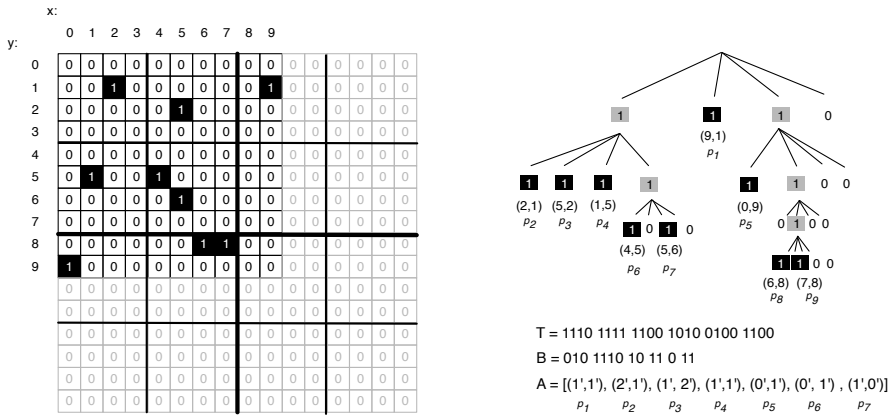

Fig. 5 A compressed version of the k^2-tree in Figure 2 with a binary matrix of size $n = 10$ and $k = 2$.

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. 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 $A_l[\mathsf{rank}_1(B_l, \mathsf{rank}_1(T_l, p))]$.

3.3 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 1) necessary to represent all binary matrices of size n^4 with m 1s (i.e., m contacts).

3.4 Construction

The construction algorithm for the ck^d-tree is based on the *inplace-construction* algorithm of the k^2-tree and k^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

$$i = \sum_{j=1}^{d} \left(\frac{p_j}{n/k} \mod k \right) \times k^{j-1}, \tag{2}$$

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

As in the k^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 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^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⟩)
```

Fig. 6 Algorithm for constructing the ck^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 2.

3.5 Orthogonal range search

The algorithm for range search in the ck^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 upper-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 7 presents the algorithm for the orthogonal range search, which retrieves the active cells in a region R. The algorithm is invoked with parameters 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: 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)
```

Fig. 7 Algorithm for orthogonal range search in the ck^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.

4 Using the Compressed k^d-tree for representing 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:

$$\mathcal{H} = \log \binom{n^2 \times \frac{\tau(\tau-1)}{2}}{c} \leq c \log \frac{n^2 \times t^2}{c} + O(c), \tag{3}$$

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.

4.1 Operations as range searches

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., $\mathsf{DirectNeighbors}(u,t)$, 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, $\mathsf{DirectNeighbors}(u,t)$ 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.

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

Table 3 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.

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[5]. 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 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 3 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.

4.2 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 3.3, 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 at 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.

4.3 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 it 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,

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

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.

5 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 2.3.1 are sensible, we must recall that they only work due to specific characteristics found in Web (static) graphs and raster data.

5.1 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, each of them using $k^{\lfloor d/2 \rfloor}$ bits. 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 8 shows the compressed version of a sparse internal node with one child.

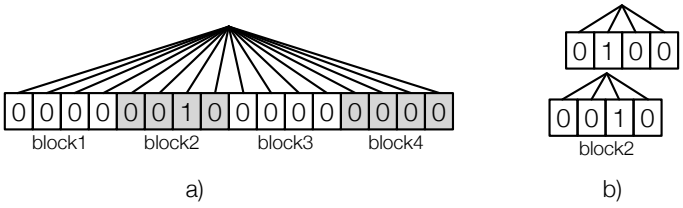

Fig. 8 Compression for a sparse node with few children: a) A sparse node in a k^4-tree with $k = 2$ and one child. b) The compressed version of the node in a).

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 8). In the traditional representation (Figure 8a), 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 8b), we only need to check 8 bits to find which one is the active child. Thus, while node compression reduces the space, 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$^{\mathsf{d}}$-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.

5.2 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$^{\mathsf{d}}$-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$^{\mathsf{d}}$-tree (bck$^{\mathsf{d}}$-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 $k - 1$ zeroes 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 9 shows the bck$^{\mathsf{d}}$-tree of the ck$^{\mathsf{d}}$-tree in Figure 5.

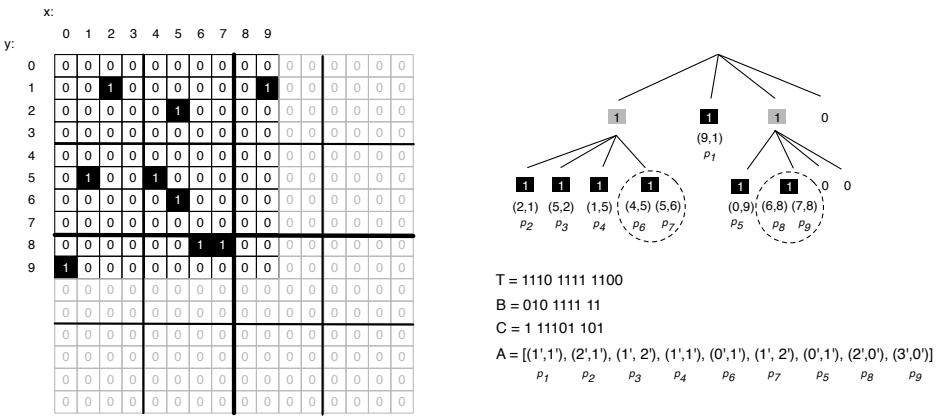

Fig. 9 The bucket version of the ck$^{\mathsf{d}}$-tree in Figure 5. Black leaves encode at most 2 cells (bucket size of $b = 2$). Two circles mark black leaves that were merged into a bucket.

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 Experimental evaluation

In this section we evaluate the performance of representative temporal graphs using the ck$^{\mathsf{d}}$-tree and the bck$^{\mathsf{d}}$-tree against the k$^{\mathsf{d}}$-tree and the ik^2-tree, which are considered the baselines for comparison. The k$^{\mathsf{d}}$-tree represents the graphs using the 4D and 3D tuples described in Section 4. For the ik^2-tree, we generate the corresponding ternary relations representing the neighboring changes of each contact. We also evaluate the performance against other compressed temporal graphs such as CAS, CET, EveLog, EdgeLog and the TGCSA.

6.1 Experimental framework

We ran several experiments on real and synthetic temporal graphs. Table 4 gives the main characteristics of these graphs: name, type, number of vertices, edges and contacts, length of lifetime, the number of contacts per edge, the space of a plain EdgeLog representation (4-byte and 8-byte integer for pointers), and the space of the entropy $\mathcal{H}$ defined in Section 3.

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 [6]. 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 a time granularity by *second*. It corresponds to the history dump of Wikipedia[7] of 2014-03-04. Wiki-Edit [Kun13] is a point-contact temporal graph, indicating the time when a user edits a Wikipedia article[8]. 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 searches 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 4 Description of temporal graphs used in the experimental evaluation. Columns Size and $\mathcal{H}$ indicate the theoretical space (in MBytes) of the plain EdgeLog and the entropy $\mathcal{H}$.

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 2.1 (i.e., by operations about

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

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

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

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. We used the Succinct Data Structure Library[9] (sdsl-lite) [GBMP14] to create the bitmaps in the k^d-tree, ck^d-tree and bck^d-tree, and the Compact Data Structure Library[10] (libcds) [CN08] for managing the arbitrary width arrays in ck^d-tree and bck^d-tree. For the k^2-tree and the ik^2-tree, we used the implementation gently provided by their authors[11].

All the experiments were conducted using $k = 2$ for the ck^d-tree, the bck^d-tree, the k^d-tree and the ik^2-tree.

6.2 Influence of 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[12], 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 use 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.

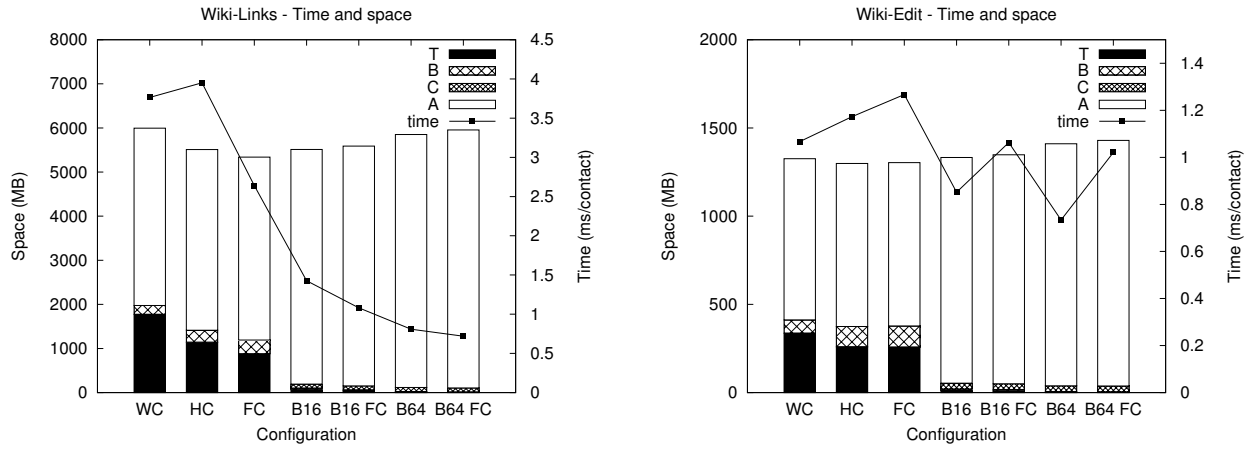

Fig. 10 Time and space results for different configurations of the Compressed k^d-tree over the temporal graphs I-Wiki-Links and P-Wiki-Edit. We include three variants of node compression: WC is the plain version, without node compression; HC and FC use node compression for Half and Full levels of the 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.

Figure 10 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

[9] Available at `https://github.com/simongog/sdsl-lite`.

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

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

[12] 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.

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 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$^{\mathsf{d}}$-tree, the node compression technique (Section 5.1) works better in *interval-contact* temporal graphs (4D matrices) than in *point-contact* and *incremental* graphs (3D matrices). This result also holds for bck$^{\mathsf{d}}$-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.

Now we compare our proposals the baseline k$^{\mathsf{d}}$-tree against the ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree. Figure 11 shows the space of the k$^{\mathsf{d}}$-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$^{\mathsf{d}}$-tree is reduced when the block size of the RRR bitmaps increases, but this is as the expenses of increasing the retrieval time. With a block size of 63 bits, the k$^{\mathsf{d}}$-tree gets its best configuration. In this case, ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-tree use 0.7 times the space of and are several times faster than the k$^{\mathsf{d}}$-tree.

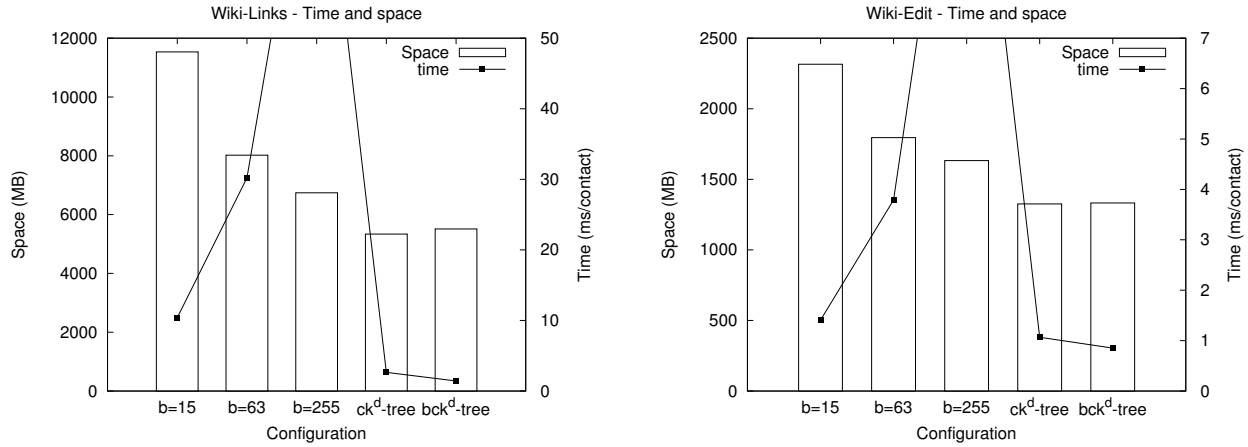

Fig. 11 Time and space results of the k$^{\mathsf{d}}$-tree using RRR compressed bitmaps with a block size of 15, 63, and 255 bits. The comparison use I-Wiki-Links and P-Wiki-Edit graphs. We include the space and time used by the ck$^{\mathsf{d}}$-tree (FC variant) and the bck$^{\mathsf{d}}$-tree (B16 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.

6.3 Space evaluation

We now compare our proposal against other structures. Space is measured in bits per contact (bpc), by dividing the total space of the structure by the number of contacts in the graph. Table 5 compares the ik^2-tree, the original k$^{\mathsf{d}}$-tree and the entropy $\mathcal{H}$ against our ck$^{\mathsf{d}}$-tree using node compression and grouping leaves into buckets (bck$^{\mathsf{d}}$-tree). We also included the space of the Snapshot k^2-tree (denoted as Snap.k^2-tree), which stores a k^2-tree of the active edges per each time instant, the TGCSA based on the compressed suffix array [BCFR14], the structures CAS and CET based on sequences, and the structures EdgeLog and EveLog based on events and adjacency lists, respectively [CARB15].

In this experiment, and in the following sections, we report the best configurations for each graph. The first two columns in Table 5 correspond to the space of our structures, the ck$^{\mathsf{d}}$-tree and the bck$^{\mathsf{d}}$-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[13].

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 TGCSA using a sampling size each 64 items on the Ψ array.

Dataset	ck^d-tree	bck^d-tree	k^d-tree	ik^2-tree	Snap.k^2-tree	CAS	CET	EveLog	EdgeLog	TGCSA	$\mathcal{H}$
I-Comm.Net	**26.0**	26.4	38.4	60.1	259.4	49.2	52.3	45.0	55.0	61.2	29.4
I-Powerlaw	**31.9**	32.6	48.8	73.0	2012.1	56.4	68.0	77.5	96.2	73.8	35.3
I-Wiki-Links	61.2	63.2	92.0	-	-	**34.5**	57.7	84.0	137.1	66.7	77.1
I-Yahoo-Netflow	**34.0**	36.1	47.4	-	-	48.6	62.9	103.7	150.5	62.7	57.5
G-Flickr-Secs	**46.1**	46.8	71.0	-	-	47.1	49.7	101.0	148.2	78.3	74.1
G-Flickr-Days	23.0	24.3	28.6	21.7	1724.0	**18.8**	31.8	74.2	134.2	50.6	32.2
P-Wiki-Edit	41.7	41.9	56.5	-	-	41.2	**38.3**	84.8	129.0	70.5	77.5
P-Yahoo-Session	47.1	45.2	46.0	-	-	**43.1**	49.1	131.8	200.9	66.6	65.8

Table 5 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 6.3 (Size is in bits/contact (bpc)).

As we can see, our proposals 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 G-Flickr-Days, because G-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, since by definition, all contacts have a duration of one time point. Notice that, as it is reported in [CARB15] CAS and EveLog are slow for reporting ReverseNeighbors, which is not a problem in our proposals.

The space used by the EdgeLog, EveLog and TGCSA is always greater than the entropy $\mathcal{H}$, regardless the type of the graph. The only exception is the *P-Wiki-Edit* graph on the TGCSA, 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.

6.4 Time evaluation

This section presents the time evaluation following the classification of operations defined in Section 2.1. 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.

6.4.1 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 the following sections, we chose the variant of the data structures reported in 6.3. We also added

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

three different bucket sizes (i.e., 2, 16, and 64) for the bck^d-tree, and three block sizes (15, 63, and 255) of the RRR bitmaps for the k^d-tree.

For this experiment, 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 12 and 13 shows 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 resume performance of all structures. In general, the time performance of Edge and EdgeNext is similar for all structures. The ck^d-tree and bck^d-tree are faster than the original k^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^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 TGCSA, although with a heavy penalty on space. In the other temporal graphs, the ck^d-tree and bck^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^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 resume the time performance of graphs with a short and a large lifetime. Figure 14 shows the evaluation for *weak* and *strong* semantics. The running time follows the same performance obtained in time instant evaluation. In ck^d-tree and bck^d-tree, the time using *strong* semantics tends to diminish when the interval size grows, but the opposite occurs using *weak* semantics. This happens because all contacts that overlap the interval must be recover. The exception to this rule is CAS, whose time performance increases with the interval size, regardless the semantics.

6.4.2 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 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 15 and 16 show the space-time tradeoff for DirectNeighbors and ReverseNeighbors using the best configuration for each graph (Section 6.3).

Our 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 TGCSA 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 TGCSA 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 6.4.1.

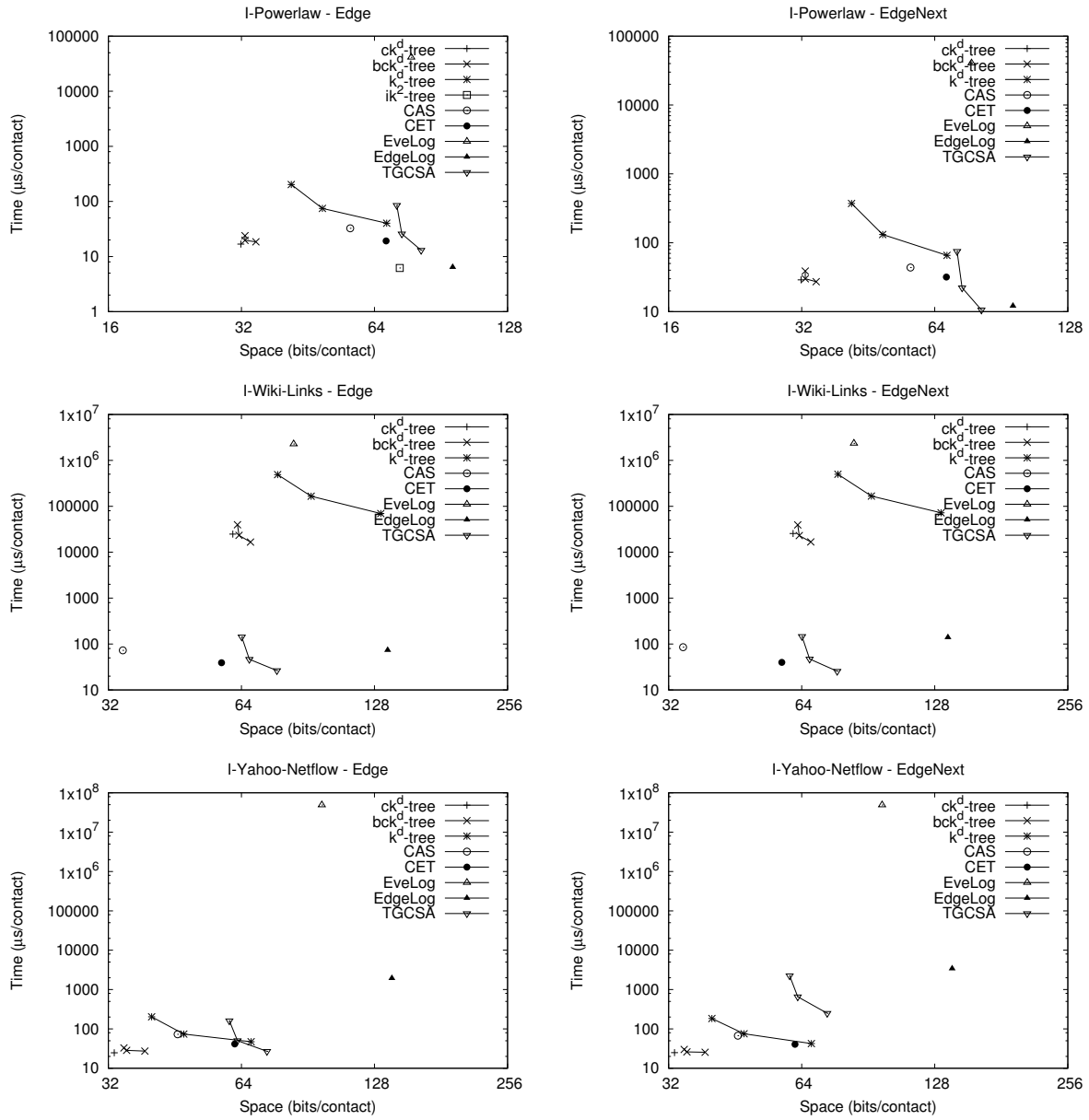

Fig. 12 Time and space performance of Edge and EdgeNext operations using the best configuration for graphs *I-Powerlaw*, *I-Wiki-Links*, and *I-Yahoo-Netflow*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

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 2.1). 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 17 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.

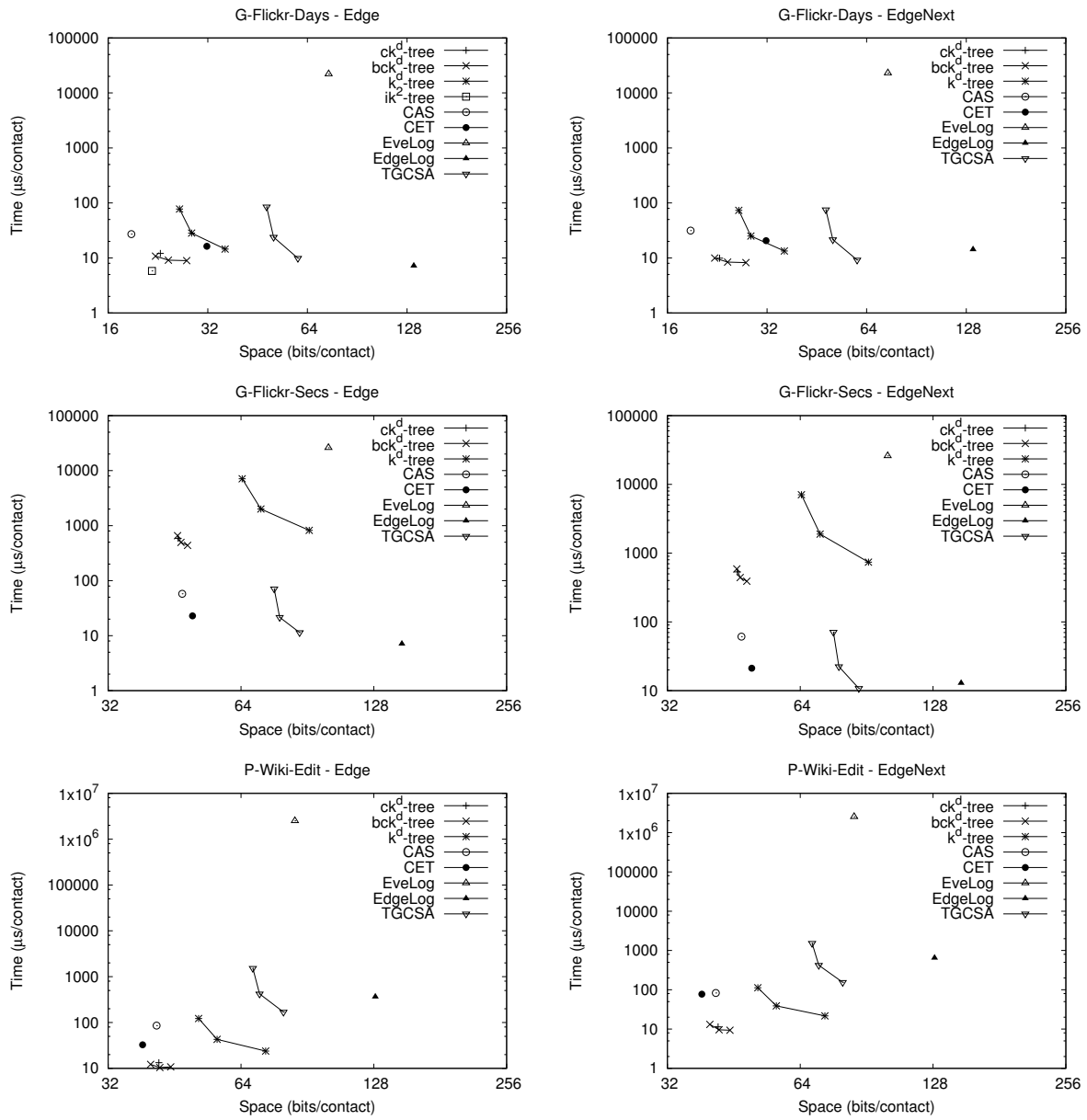

Fig. 13 Time and space performance of Edge and EdgeNext operations using the best configuration for graphs *G-Flickr-Days*, *G-Flickr-Secs*, and *P-Wiki-Edit*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

Like for time instant queries, the performance over the *I-Powerlaw* graph of our ck$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-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$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-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.

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 TGCSA do not change with both the semantics

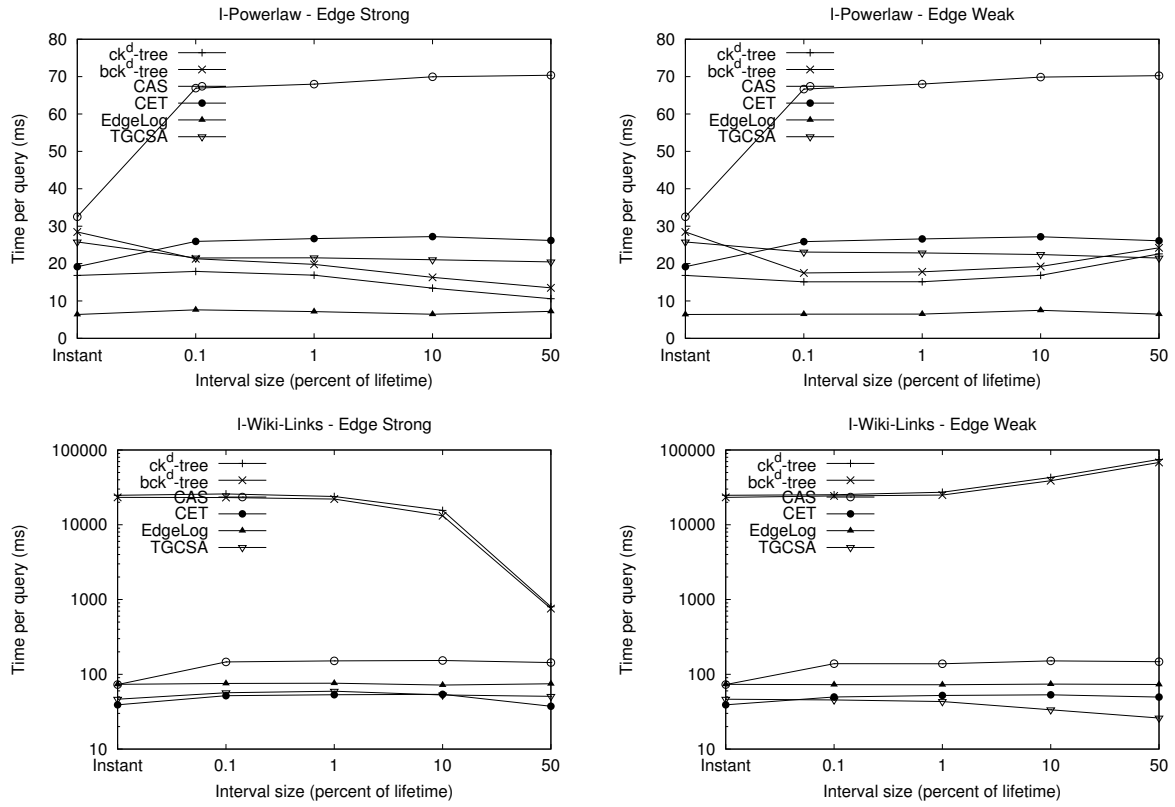

Fig. 14 Time performance of the interval version of Edge over the *I-Powerlaw* and *I-Wiki-Links* graphs. The image on the left shows the *strong* and on the right the *weak* semantics. Time performance is measured in μs per query.

and the size of the time interval. Indeed, they got the same performance of DirectNeighbors over a time instant in both temporal graphs.

6.4.3 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 19 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 18 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^d-tree and bck^d-tree is several times faster than sequence-based structures over *interval-contact* and *incremental* graphs. In *point-contact* graphs, CET outperforms our proposals. Taking into account the dynamism of *I-Powerlaw*, which has a constant number of active edges per time instant (Figure 19a), 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 19b. The performance of the ck^d-trees follows the same tendency of the TGCSA, 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^d-tree and bck^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.

6.4.4 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

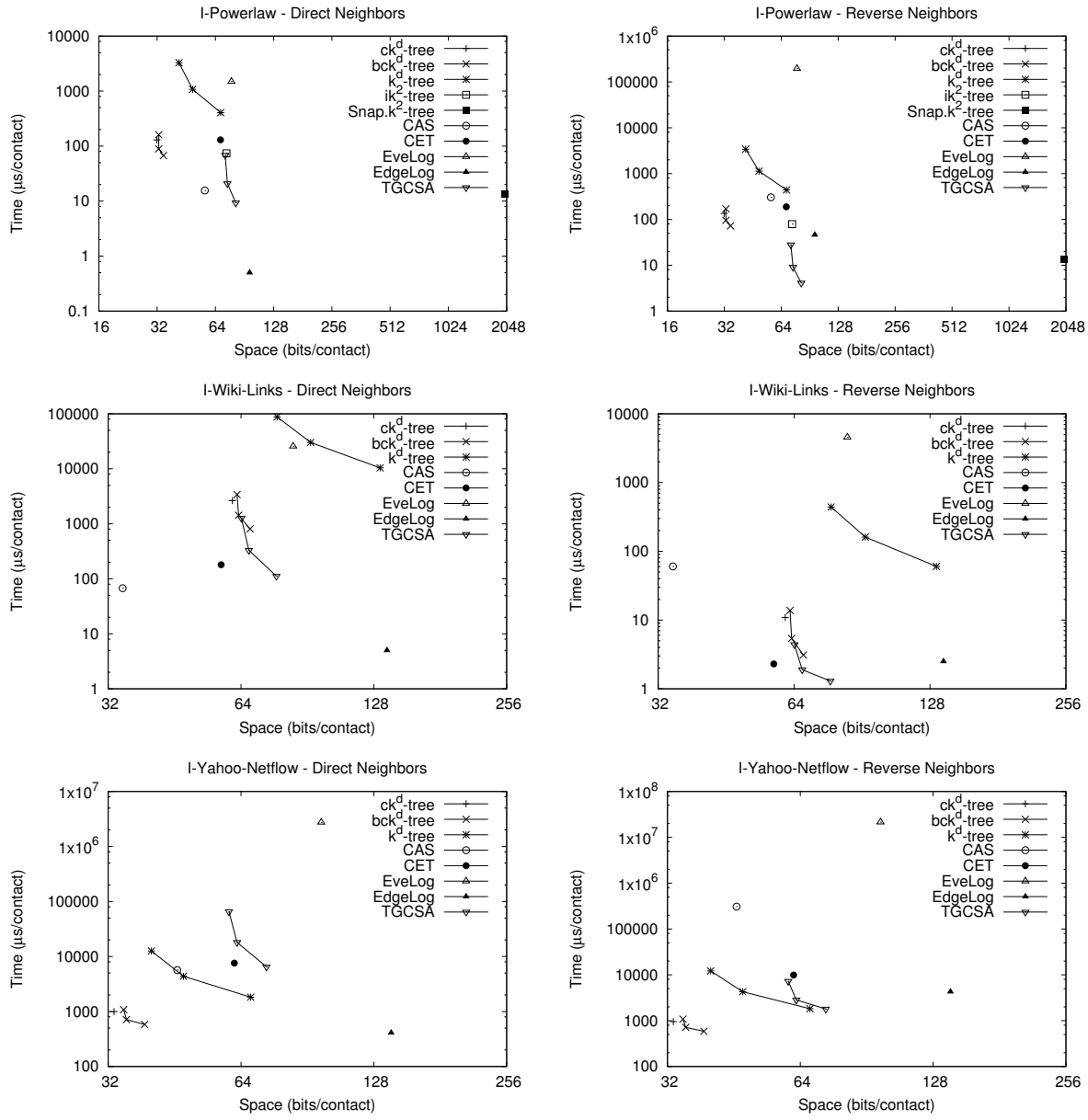

Fig. 15 Time and space performance of DirectNeighbors and ReverseNeighbors operations using the best configuration for graphs *I-Powerlaw*, *I-Wiki-Links*, and *I-Yahoo-Netflow*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

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, which corresponds 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^d-tree, bck^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 TGCSA 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 20 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*.

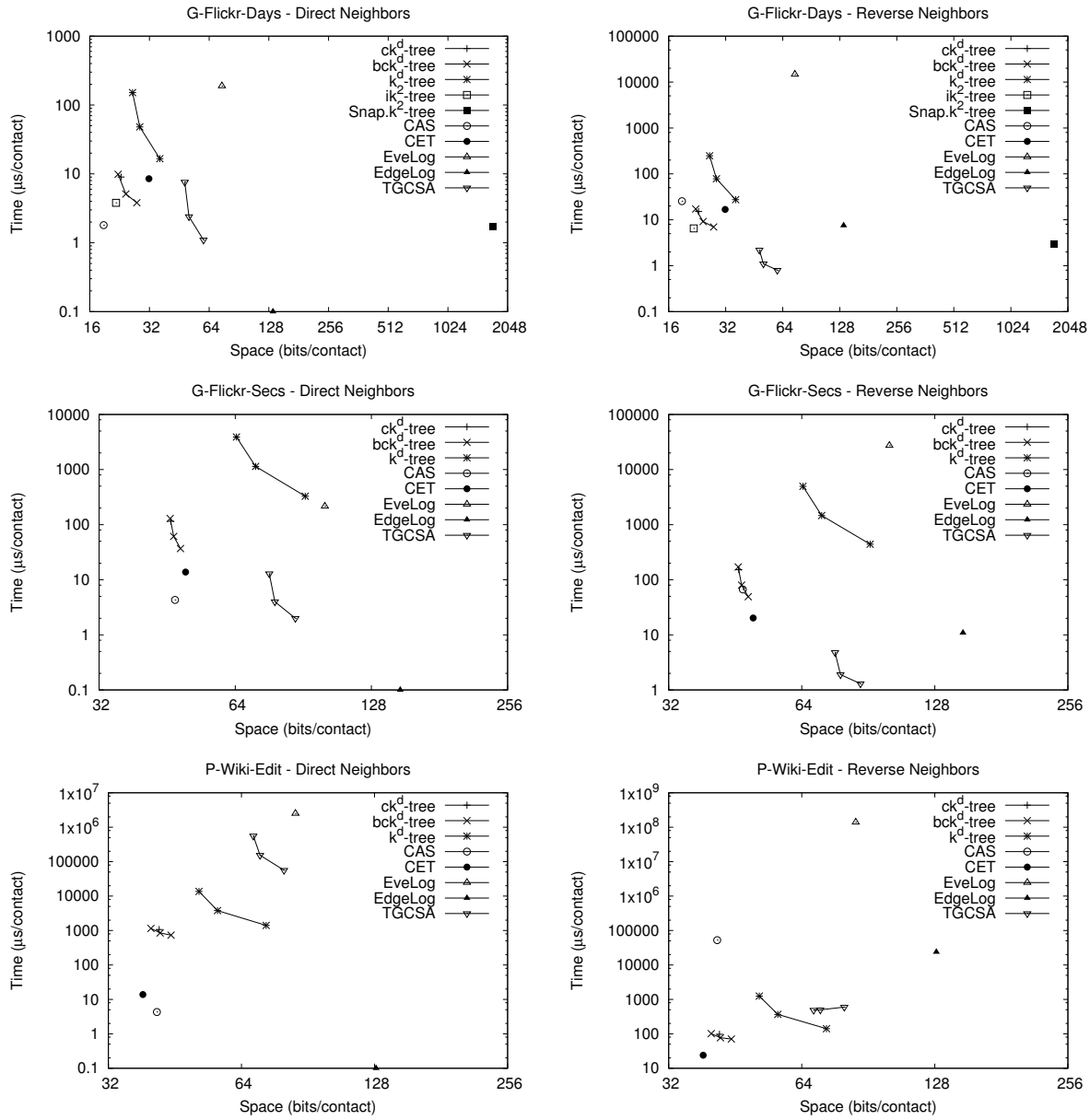

Fig. 16 Time and space performance of DirectNeighbors and ReverseNeighbors operations using the best configuration for graphs *G-Flickr-Days*, *G-Flickr-Secs* and *P-Wiki-Edit*. Time performance is measured in μs per contact reported and space in bits per contact (bpc).

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*.

6.4.5 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 19) 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, 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 4.3. Indeed, 42% of contacts in *I-Wiki-Links*

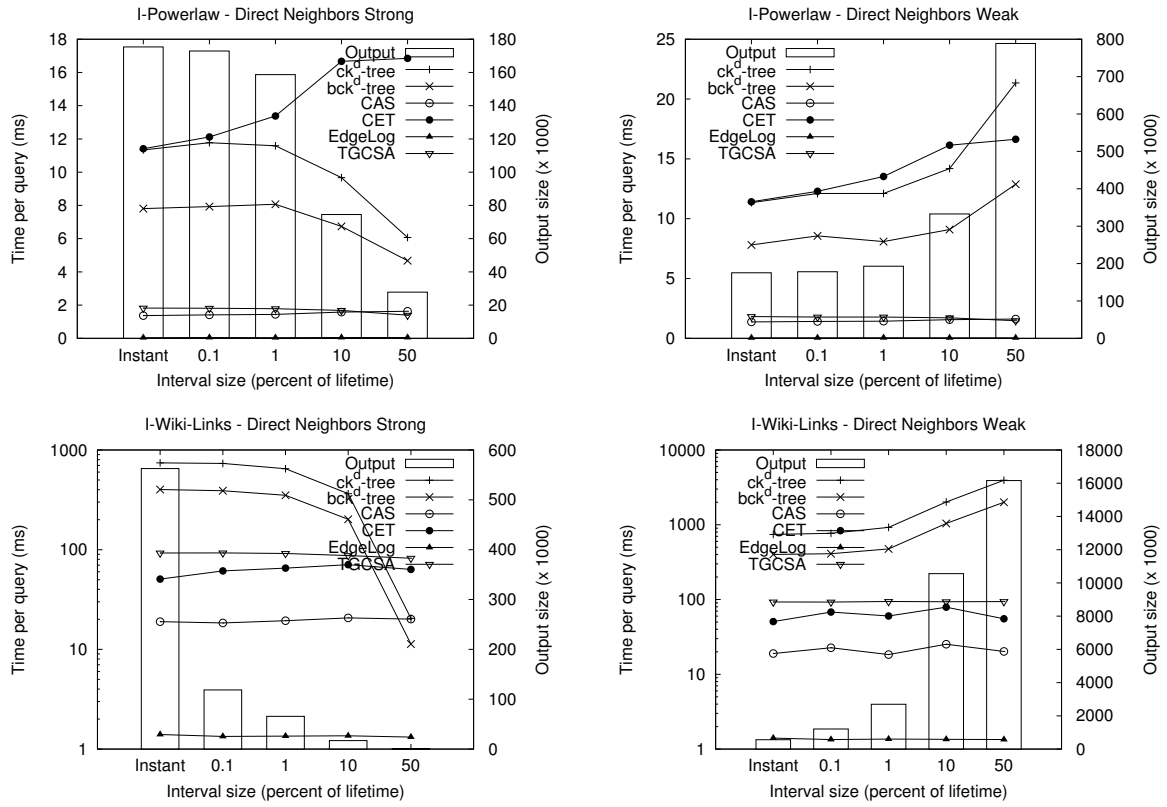

Fig. 17 Time performance of the interval version of the DirectNeighbors operation over the *I-Powerlaw* and *I-Wiki-Links* graphs. The image on the left shows the *strong* and on the right the *weak* semantics. Time performance is measured in *ms* per query.

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 6.4.5 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 6.4.2. 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.

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

Table 6 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.

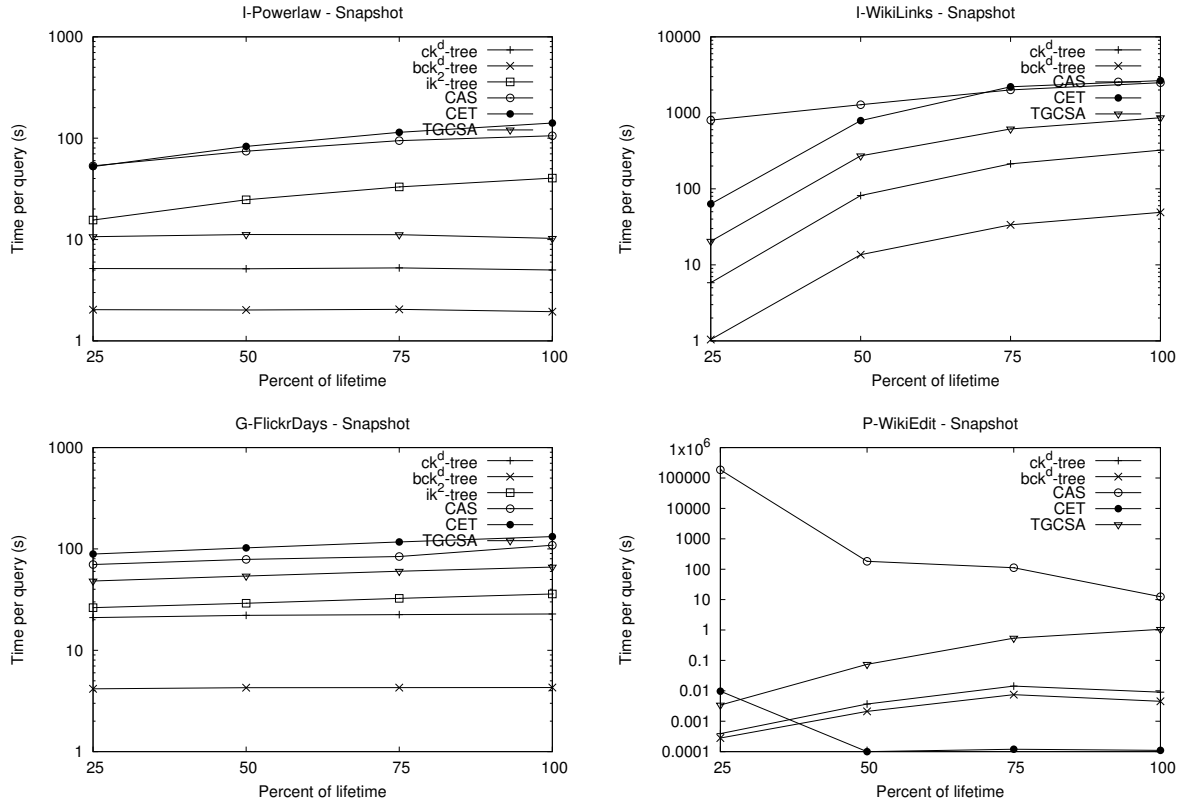

Fig. 18 Time comparison of Snapshot at the 25%, 50%, 75%, and 100% of the datasets' lifetime. (Time in *s* per query).

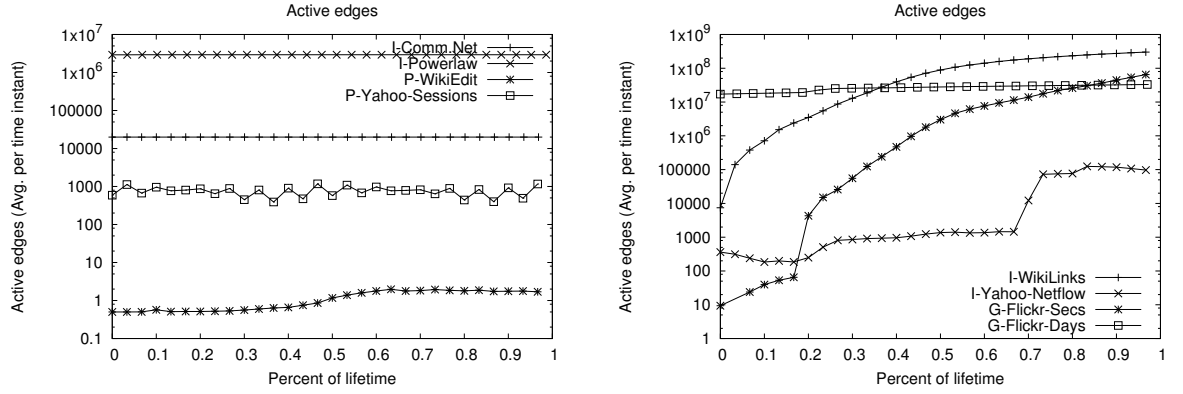

Fig. 19 Average number of active edges per time instant in all datasets. Figure on the left shows results for graphs *I-Comm.Net*, *I-Powerlaw*, *P-WikiEdit*, and *P-Yahoo-Sessions* with a uniform number of active edges. Figure on the right shows results for graphs *I-WikiLinks*, *I-Yahoo-Netflow*, *G-Flickr-Secs*, and *G-Flickr-Days* with an increasing number of active edges.

6.4.6 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 ck$^{\mathsf{d}}$-trees 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. This strategy is the same used in [CARB15] to evaluate the performance of the sequence-based structures.

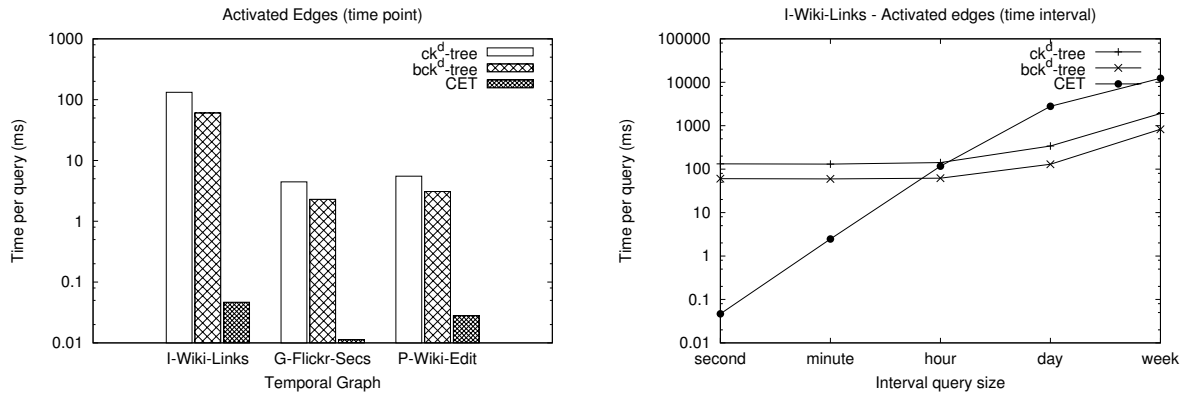

Fig. 20 Time performance of ActivatedEdges queries at time instants and during time intervals. Figure on the left shows the performance for time instants using *I-Wikipedia-Links*, *G-Flickr-Secs* and *P-Wiki-Edit* graphs. Figure on the right shows the performance of *I-Wikipedia-Links* over time intervals corresponding to a second, an hour, a day, and a week.

Table 7 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 Description of temporal graphs used in the sensitivity analysis

Before going further, we present the space used by structures in Table 8. The space of ck$^{\mathsf{d}}$-trees is under the entropy $\mathcal{H}$, but it is above the space used by EdgeLog. But, as we will see in what follows, and as Caro *et al.* [CARB15] pointed out, the main drawback of EdgeLog and EveLog is the time to process the list of events and neighboring neighbors. Other structures use more space than the ck$^{\mathsf{d}}$-tree structure.

Dataset	ck$^{\mathsf{d}}$-tree	bck$^{\mathsf{d}}$-tree	k$^{\mathsf{d}}$-tree	CAS	CET	EveLog	EdgeLog	TGCSA	$\mathcal{H}$
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 8 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 21 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$^{\mathsf{d}}$-tree and bck$^{\mathsf{d}}$-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 TGCSA works much slower than the other structures, contrary as we got in the time performance evaluation in Section 6.4.2.

In the second experiment, we evaluated the sensitivity with respect to the out degree of vertices. Figure 22 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

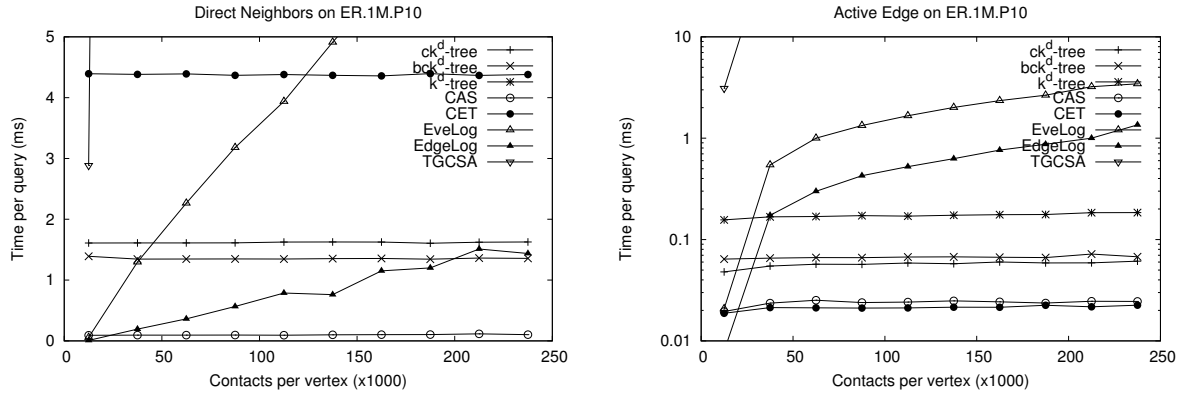

Fig. 21 Sensitivity analysis of DirectNeighbors and Edge operations in the ER.1M.P10 synthetic graph. Time per query (ms) v/s contacts per vertex.

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 TGCSA is poor, several times slower than the other structures.

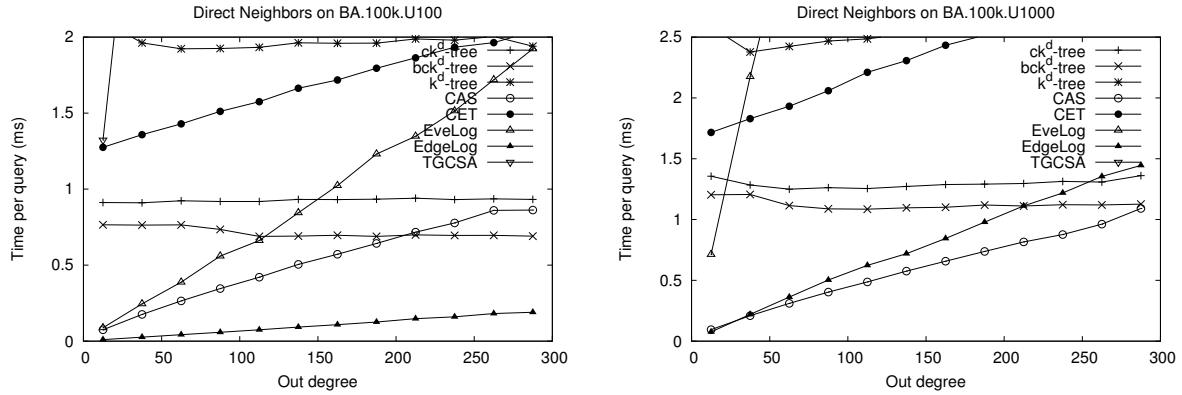

Fig. 22 Sensitivity analysis of answering DirectNeighbors queries over the BA.100k.U1000 and BA.100k.U100 synthetic graphs. Time per query (ms) v/s out-degree per vertex.

7 Conclusions and future work

This work proposed the Compressed k^d-tree (ck^d-tree), which is an improvement of the original k^d-tree. The main advantage of this structures its efficient use of space, several times better than the worst case of the k^d-tree when input data is sparse and does not share any clustering property. It also guarantees, without considering any regularity, an asymptotical space equal to the lower bound on the number of bits needed to represent a 4D binary matrix with m active cells. Although, this new structure can be used for any d-dimension space, it was specially adapted to represent temporal graphs in 4D and 3D.

With the proposed structures, the computation of all temporal-graph graphs can be defined in terms of only a orthogonal range search, a much simple mechanism than the ad-hoc programming of queries in CET, CAS, TGCSA, EdgeLog and EveLog structures. This is possible because each dimension (i.e., vertices and time instants) is treated in the same way. In addition, the performance for DirectNeighbors queries is the same than for ReverseNeighbors queries, as in the CET and TGCSA structures. Experiments show that the space of our data structure is near the 50% of the fastest version of the k^d-tree using compressed bitmaps. In addition, it can be several times faster than the previous proposals CET and CAS for queries that recover the state of the graph at a

given time instant. Also, it is several times smaller than the TGCSA and is also capable to recover all components of a contact on all operations.

We also compared the space usage of the proposed structures against the space used by the ik^2-tree for temporal graphs. In this case, the ck^d-tree uses up to 50% less space than the ik^2-tree, or the same space in the best case of the ik^2-tree (G-Flickr-Days datasets with a short lifetime). The same happens with respect to the time performance, where the ck^d-tree outperforms many times the ik^2-tree for all datasets except the G-Flickr-Days dataset, in which case both structures show the same 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.

Notice that the ck^d-tree can be used in other contexts where data is multidimensional such as triples in RDF, non-clustered 3D regions, or even to store and query the evolution of raster data. An extensive experimental evaluation in these contexts is left for future work. Also as future work, we need to check if we are able to compress the isolated cells in the buckets. Nowadays this is a current technology in many industrial implementations of B+-trees. Initial experiments reveal that the entropy of isolated cells in buckets could improve if we store them using xor-encoding [Hud09], but this is not enough because we need to have a fast decompression schema to maintain the performance. With respect to temporal graphs in 4D and 3D binary matrices, we believe that a node ordering considering the time activation/deactivation of edges could be useful to improve the clustering of the cells in the matrix. This mechanism has been useful for improving the compression and time for storing static graphs in k^2-tree. We also need to improve the partitioning method on temporal graphs sharing characteristics of *incremental* graphs (such as the Wiki-Links dataset), because the space is still far from the minimum obtained by other structures.

Acknowledgements Diego Caro and M. Andrea Rodríguez were partially funded by Fondef D09I1185. Diego Caro was also funded by CONICYT PhD scholarship and M. Andrea Rodríguez by Fondecyt 1140428 and MINECO (PGE and FEDER) grant TIN2013-46801-C4-3-R. Nieves Brisaboa and Antonio Fariña are funded by MINECO (PGE and FEDER) grants TIN2013-46238-C4-3-R, and TIN2013-47090-C3-3-P; CDTI, AGI and MINECO grant CDTI-00064563/ITC-20133062; ICT COST Action IC1302; and by Xunta de Galicia (co-founded with FEDER) grant GRC2013/053. We also thank to Diego Seco for his help in the preliminary discussions of the structures, to Gonzalo Navarro and Simon Gog for their suggestions on the improvement of the data structures and the experimental evaluation, and to Claudio Sanhueza from Yahoo! Labs who helps us with the P-Yahoo-Session dataset.

References

[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 *Proceedings of the Americas Conference on Information Systems (AMCIS)*. Association for Information Systems, 2011.

[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.

[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.

[BdBN12] Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Compressed dynamic binary relations. In *Data Compression Conference (DCC)*, pages 52–61. IEEE Computer Society, 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.

[BLN09] NievesR. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 18–30. Springer Berlin Heidelberg, 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.

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

[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.

[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.

[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.

[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. *Algorithms and Theory of Computation Handbook*, chapter Dynamic Graph Algorithms, pages 9–1 – 9–27. Chapman & Hall/CRC, 2010.

[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.

[FBN+12] 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.

[FV02] Afonso Ferreira and Laurent Viennot. A Note on Models, Algorithms, and Data Structures for Dynamic Communication Networks. Technical Report RR-4403, INRIA, 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.

[GBMP14] 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.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM/SIAM, 2003.

[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.

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

[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.

[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.

[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.

[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.

[RLK+11] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *The Proceedings of the VLDB Endowment (PVLDB)*, 4(11):726–737, 2011.

[RRR02] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'12*, pages 233–242, 2002.

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

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

[XFJ03] B. Bui Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar ram-cpu cache compression. In *Proc. ICDE'06*, page 59, 2006.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW'08*, pages 387–396, 2008.