Cache-Friendly Compressed Boolean Matrices

Antonio Fariña[1], Adrián Gómez-Brandón[1,3], Asunción Gómez-Colomer[2,3], and Gonzalo Navarro[2,3]

[1] Universidade da Coruña, Spain
[2] Department of Computer Science, University of Chile, Chile
[3] Millennium Institute for Foundational Research on Data (IMFD), Chile
fari@udc.es,asuncion.gomez@ug.uchile.cl,adrian.gbrandon@udc.es,
gnavarro@uchile.cl

Abstract. We introduce a new compressed representation of sparse Boolean matrices that enjoys reference locality properties. We build on an existing representation based on LOUDS-deployed cardinal trees, and design one based instead on DFUDS. While this brings various complications, we show that the resulting matrix representation is considerably faster to carry out sums and multiplications, with speedups of up to 60%.

1 Introduction

Sparse binary matrices arise in a number of applications, including the representation of labeled graphs (e.g., in graph databases, social networks, etc.) and Machine Learning [7]. Recent work [1, 2], aiming at solving regular path queries on graph databases, showed that a representation based on k^2-trees [4], which exploits sparsity and clustering of the 1s in the matrix, efficiently supported a general Boolean matrix algebra including transposition, sums and other pointwise operations, multiplication, and transitive closure. In this paper we focus on better representations to support such matrix algebra.

The k^2-tree is a cardinal tree of arity k^2 with a bounded height. Its original representation [4] is analogous to the so-called LOUDS deployment [10], which represents *ordinal* trees in levelwise node order, using 2 bits per node. The *cardinal* LOUDS version, instead, uses k^2 bits per *internal* node to store a *signature* telling which children exist. The sparse matrices referenced above used $k = 2$, so the representation uses 4 bits per internal node. While efficient for navigation, because it relies on the most basic primitives for bitvectors, *rank* and *select* [5, 11],[4] this levelwise representation suffers from poor reference locality.

A way to improve reference locality is to replace LOUDS by a variant of the so-called DFUDS deployment of *ordinal* trees [3], which also uses 2 bits per node but deploys the nodes in depth-first order. The existing *cardinal* DFUDS-based representations [3] [12, Sec. 8.3.1], however, would use $k^2 + 2$ bits per (internal and leaf) node, 6 in our case.[5] This is much more than the space used by LOUDS.

[4] This adds $o(1)$ bits per bit of the representation, which we are disregarding to streamline the discussion.

[5] They [3] [12, Sec. 8.4] introduce representations using $O(\log k)$ bits per node, but those are likely to be practical only for large enough k values.

The cardinal DFUDS representation consists of the ordinal DFUDS plus the LOUDS signatures in depth-first order. The ordinal DFUDS is used to navigate the tree, whereas the LOUDS signatures mark which children exist. A further burden of cardinal DFUDS when used on k^2-trees is that the ordinal DFUDS sequence needs to explicitly represent all leaves, even those at the maximum possible depth, and thus we must also include their LOUDS signatures (k^2 zeros). The LOUDS levelwise representation, instead, can ignore those empty signatures because they would appear contiguously at the end of the sequence. This sums to $k^2 + 2$ bits per node, instead of LOUDS' k^2 bits per internal node.

In this paper we note and exploit the fact that *the ordinal DFUDS representation of a node is a function of its LOUDS signature*, so as to navigate the tree using, essentially, the node signatures deployed in DFUDS' depth-first order, and getting rid of the ordinal DFUDS sequence. Further, we manage to remove the empty signatures of the existing cardinal DFUDS representation, at the price of extending the signatures of internal nodes by one bit. As a consequence, our representation uses only $k^2 + 1$ bits per *internal* node. Since, in this representation, subtrees are deployed compactly in the sequence, tree traversals are more cache-friendly. We dub our representation *kache-trees*.

Our kache-trees use the signatures of (sequences of) nodes and transform them on the fly to their ordinal DFUDS representations, then resorting to the classic DFUDS navigation implemented over rmM-trees [13]. Note that this is in the line of the theoretical proposal of Farzan and Munro [8], yet our representation is specialized to k^2-trees, practical and simple to implement.

In order to evaluate the impact of the improved locality of reference, we implemented our kache-trees and used them to solve sums and multiplications on Boolean matrices. Our kache-trees turn out to be considerably faster — offering speedups of up to 60% — on the denser matrices, which are the most time-consuming to multiply. This indeed turns out to be a consequence of cache misses, which are reduced by about a half. On sums, where both representations traverse the trees sequentially with few cache misses, our representation performs similarly on the denser matrices and speeeds up by up to 60% on the sparser ones, because there are more cases of copying whole submatrices to the output, and kache-trees deploy those submatrices contiguously.

2 LOUDS, DFUDS, and k^2-trees

Rooted trees can be classified into two main categories: ordinal and cardinal. Ordinal trees distinguish only the order among children, whereas cardinal trees have a fixed set of possible children, each of which can be present or not. LOUDS and DFUDS are two popular succinct representations of ordinal trees, which use $2n + o(n)$ bits of space for an ordinal tree of n nodes and support various navigation operations, like going to a node's parent or child, in constant time.

2.1 LOUDS representation of ordinal trees

LOUDS (Level-Order Unary Degree Sequence) [10] represents an ordinal tree of n nodes using a single bitvector B, where the nodes are deployed in level order. This representation encodes each tree node having c children ($c = 0$ for leaves) with its so-called *description*, $\texttt{1}^c\texttt{0}$. The identifier of a node is the position where its description starts in B. Bitvector B is prepended with `10` to avoid certain special cases. It is easy to show that the length of B is $2n + 1$.

LOUDS supports basic operations, such as navigating to the parent and children of a node v, in a constant amount of basic *rank* and *select* primitives, which run themselves in $O(1)$ time if one spends $o(n)$ further space [5, 11]. For example, the t-th child of v is $\texttt{child}(v, t) = select_0(B, rank_1(B, v - 1 + t)) + 1$.

2.2 DFUDS representation of ordinal trees

DFUDS (Depth-First Unary Degree Sequence) [3] deploys the same node descriptions of LOUDS, yet in depth-first order. It regards the bits as parentheses; the description of a node with c children is '$(^c)$'. The bit sequence is now prepended with '$(\,)$' to prevent special cases, and B is of length $2n + 2$.

An advantage of this representation is that all nodes within a subtree are deployed contiguously in the bitvector. Together with the use of parentheses, this enables many more operations not supported in LOUDS. The DFUDS operations build mostly on two basic primitives apart from *rank* and *select*:

$$\texttt{fwd\_search}(B, i, d) = \min\{j > i, \texttt{excess}(B, j) = \texttt{excess}(B, i) + d\} \cup \{n + 1\}$$
$$\texttt{bwd\_search}(B, i, d) = \max\{j < i, \texttt{excess}(B, j) = \texttt{excess}(B, i) + d\} \cup \{0\},$$

where $\texttt{excess}(B, j)$ is the number of opening minus closing parentheses in $B[1..j]$. For example, $\texttt{child}(v, t)$ is calculated as $\texttt{fwd\_search}(B, \texttt{succ}_0(B, v) - t, -1) + 1$, where $\texttt{succ}_0(B, v) = select_0(rank_0(B, v - 1) + 1)$.

Those primitives are more complex to implement than *rank* and *select*. A popular implementation uses $o(n)$ further bits to store a so-called rmM-tree [13]. This is a perfect binary tree whose leaves cut B into blocks of b parentheses, storing (at least) the minimum and the total excess of the block. Internal rmM-tree nodes store the same summary information on the subsequence spanned by its descendant leaves. It is easy to build the rmM-tree bottom-up in linear time.

To compute $\texttt{fwd\_search}(B, i, d)$ using the rmM-tree, we begin by scanning the block of B that contains the i-th parenthesis. Within this block, we search for the smallest position $j > i$ where the excess is $\texttt{excess}(B, i) + d$. If we do not find it, we use the rmM-tree to move upwards and locate the nearest ancestor whose subtree contains the excess sought, based on its stored minimum and total excess. Then, we move downward towards the child that contains the position we are looking for, until reaching the rmM-tree leaf that contains the answer. This process ends by scanning that leaf, where j is finally found.

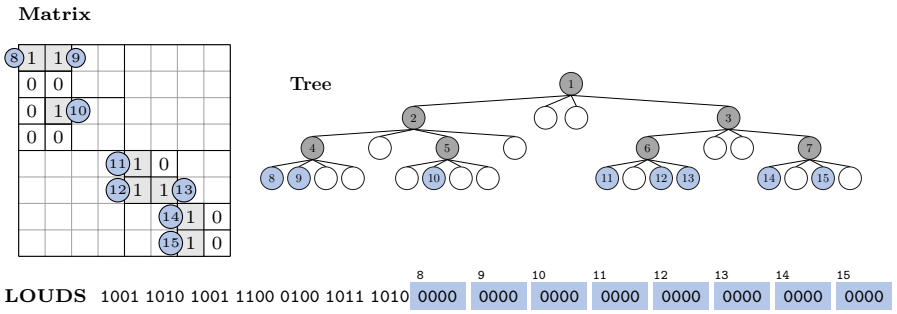

Fig. 1. An 8×8 matrix (left, above), its corresponding k^2-tree (right, above), and its cardinal LOUDS representation (below); the higlighted leaves 0000 are not represented.

2.3 K^2-trees and cardinal LOUDS

The k^2-tree [4] is a compact representation of binary $l \times l$ matrices by recursively dividing them into k^2 quadrants of equal area. Non-empty quadrants are further subdivided recursively, up to height $h = \lceil \log_k l \rceil$, where the submatrices represent individual cells (if l is not a power of k, the matrix is padded with 0s up to side k^h). The k^2-tree excels in representing sparse and clustered matrices.

The k^2-tree is a cardinal tree of arity k^2, and is represented, as in LOUDS, by deploying the nodes levelwise. This time, however, each node is represented with a *signature* of k^2 bits that indicates with `1`s which children of the node exist (i.e., which of its submatrices are nonempty). The nodes are identified by the position of their signature (counting number of signatures, not of bits), and the navigation operations are even simpler than for ordinal LOUDS. For example, the formula to descend to a(n existing) child is $\texttt{child}(v,t) = rank_1(B, k^2(v-1)) + t + 1$.

The k^2-tree is a particular case of the cardinal LOUDS representation in that, since the height is fixed to h, there is no need to store the last level of leaves, as they are always empty. Therefore, the representation of a k^2-tree uses k^2 bits per *internal* node, without using any bits for the last-level leaves.

Fig. 1 illustrates a matrix with eight 1s along with its corresponding k^2-tree representation for $k = 2$. We also show the cardinal LOUDS representation of the tree, highlighting the final leaves that are omitted from the representation.

2.4 DFUDS representation of cardinal trees

Cardinal trees can also be represented with a DFUDS-inspired representation [3] [12, Sec. 8.3.1]. This stores the original DFUDS for ordinal trees (using the rmM-tree we described above) along with an additional bitvector, S, that contains the k^2-bit signatures of the nodes in depth-first order. The DFUDS sequence provides the navigation, while the signatures indicate which children of the current node exist. The DFUDS node v corresponds to the $rank_0(B, v)$-th signature.

Fig. 2 illustrates this representation on our running example. Note that leaf signatures `0000` are now spread along the sequence and are not easily discarded.

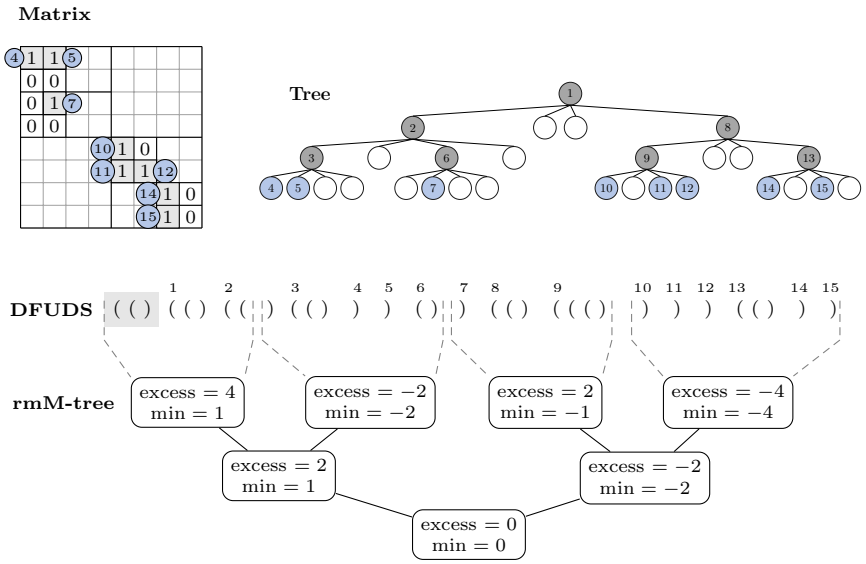

Fig. 2. Cardinal DFUDS representation of the same k^2-tree of Fig. 1, using block size $b = 8$ for the rmM-tree.

3 LOUDS-Based Matrix Algebra

A sparse Boolean matrix representation [1, 2] builds on the cardinal LOUDS described in Section 2.3. The authors implement an algebra where matrices are input to and returned from operators in this format. We focus on two of the most popular operations: the Boolean sum and product.

To compute the sum $A + B$, they sequentially traverse the representation of both k^2-trees, in levelwise order. The algorithm uses a queue of tasks of types: *copy* tasks, which just copy nodes from either A or B to the output and enqueue the tasks of copying their children, and *merge* tasks, which output the bitwise-or of both signatures and enqueue the tasks of copying or merging the corresponding children of A and B (depending on whether only A, or only B, or both, have each child). While this one-pass procedure is very efficient, we notice that copying a whole subtree to the output when the other operand's subtree is empty, is done signature by signature, as the nonempty subtree is not deployed in contiguous form in the LOUDS format.

For the multiplication $A \times B$, they use the standard recursive algorithm:

$$\begin{pmatrix} A_0 & A_1 \\ A_2 & A_3 \end{pmatrix} \times \begin{pmatrix} B_0 & B_1 \\ B_2 & B_3 \end{pmatrix} = \begin{pmatrix} A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \end{pmatrix}.$$

Zero-bit children in k^2-trees indicate empty submatrices, in which case the multiplication is avoided. The sums are implemented as explained; sums with empty submatrices are also converted into copies. Those copies are done level by level because, again, subtrees are not deployed contiguously in LOUDS.

4 Our Cardinal DFUDS Representation for k^2-trees

We introduce a new k^2-tree representation, named *kache-tree*, based on cardinal DFUDS, improving the one seen in Section 2.4. An advantage of a DFUDS-based representation, compared to the standard LOUDS-based one, is that the signatures are concatenated in depth-first order, and thus every subtree is deployed in a contiguous sequence and its traversal becomes more cache-friendly. A disadvantage of the structure of Section 2.4 is that, in addition to the n node signatures, it adds the $2n$-bits ordinal DFUDS sequence of the same tree in order to support navigation.

4.1 The key idea

Our new representation gets rid of those $2n$ bits of the ordinal DFUDS representation, together with the empty signatures of last-level leaves. It only retains the sequence of signatures of internal nodes in depth-first order. Unlike in cardinal LOUDS, it is not immediate how to navigate those depth-first-ordered signatures without the help of the ordinal DFUDS sequence.

A first insight to solving this challenge is given in the following lemma.

Lemma 1. *In a k^2-tree, the ordinal DFUDS description of a node is a function of its signature and of the depth of the node.*

Proof. Let the signature of node v have c 1s; therefore v has c children. If the depth of v is not that of the last level of the internal nodes, its DFUDS description is '$(^c\,)$' (i.e., c opening parentheses followed by a closing one). Otherwise, v is the parent of c last-level leaves and its DFUDS description is '$(^c\,)\,)^c$', that is, the same as an internal node with c children followed by the c closing parentheses of the leaves. □

An example of those cases can be seen in Fig. 3. The signature of node 1 is '`1010`' and, since the node is not at the last level of internal nodes, its DFUDS representation is '(( )'. The signature of node 3 is '`1100`' and, since it is at the last level of internal nodes, its DFUDs representation is '(( )))'.

4.2 The data structure

In order to distinguish both cases, we will prepend the node signatures with an additional bit that is `1` when the node is at the last depth of internal nodes, and `0` otherwise. With those enhanced signatures, we can obtain the DFUDS representation of a node *on the fly*, without having to spend *two* bits per *node*. Instead, we spend *one* bit per *internal node*. In addition, we do not represent the empty signatures of last-level leaves.

In our example above, the enhanced signature of node 1 would be '`01010`' and that of node 3 would be '`11100`'. The cardinal LOUDS representation of this tree uses 28 bits (7 4-bit signatures, see Fig. 1), the original cardinal DFUDS

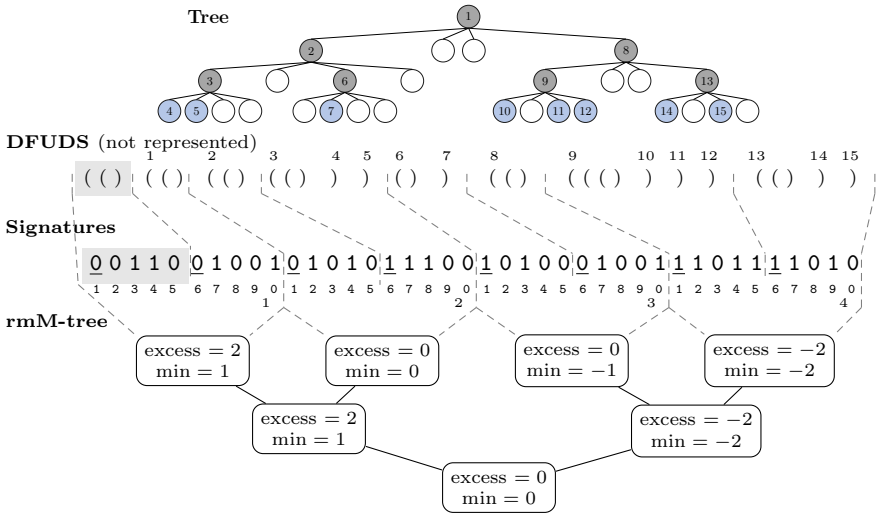

Fig. 3. The kache-tree representation of our running example, using blocks of $b = 2$ signatures for the rmM-tree leaves. Note that those leaves cover an irregular number of DFUDS parentheses (those parentheses are not represented in the kache-tree).

representation uses 92 bits (15 4-bit signatures plus 32 from ordinal DFUDS, see Fig. 2), and our new representation uses 40 bits (8 5-bit signatures, see Fig. 3).

As we presented in Section 2.2, operations such as *fwd_search* on DFUDS need to scan a subsequence of parentheses to compute its *excess* and *minimum excess*. To speed up that computation in our sequence of signatures and remain cache-friendly, we precompute a small universal table $T[0, 2^{15} - 1]$ that is independent of the sequence itself. Each entry $T[x]$ contains the field e (excess) and m (minimum excess) for x, where x is the number that corresponds to a sequence of 3 signatures of 5 bits. Hence, reading 15 bits, we can compute the *excess* and *minimum excess* of three consecutive nodes simultaneously. The total space of T is 64 KB, which allows it fit in cache. A smaller table $t[0, 2^5 - 1]$ handles individual signatures, storing the same fields e and m, a field c with the number of 1s in the signature, and a field p with the actual parentheses sequence represented by the signature.

Table T allows us simulate the efficient scanning of parentheses performed by *fwd_search*. As seen in Section 2.2, that operation is sped up by an rmM-tree built on the DFUDS sequence. In our approach, each rmM-tree leaf covers a fixed number b of nodes, that is, a variable number of parentheses. Apart from that, our rmM-tree works just as the original one.

The construction of rmM-tree, assuming that b is a multiple of 3, scans consecutive subsequences of $\frac{b}{3}$ chunks of 15 bits. From the $T[\cdot]$ fields of the chunks, the excess and minimum excess of each leaf is computed. The fields of the internal nodes are then completed as for the original rmM-tree; see Section 2.2.

4.3 Forward search

In the DFUDS format, operation $\texttt{fwd\_search}(i, d)$ requires finding relative excess d forwards from parenthesis position i. In our representation, this may imply starting the search from the middle of a signature. Consequently, we redefine the operation as $\texttt{fwd\_search}(i, d, \delta)$, meaning that the scan should start from the δth parenthesis inside the ith signature, and that we must return the signature that contains the corresponding parenthesis position. We then proceed as follows:

1. We read the ith signature, x, obtain its DFUDS representation from $t[x].p$, and scan it from the δth parenthesis. We return the answer i directly if the difference d is found within $t[x].p$; otherwise we update d by subtracting the excess of the traversed parentheses.
2. We scan the rest of the corresponding rmM-tree leaf, from the $(i+1)$th signature onwards, by chunks of 3 signatures, skipping chunks y where $d < T[y].m$ and in that case updating d to $d - T[y].e$.
3. If $d \geq T[y].m$ for some chunk y, we scan y signature-wise, using $t[x].m$ and $t[x].e$, until finding the one containing the answer, and return it.
4. If we scan the whole rmM-tree leaf without finding the answer, we continue the navigation on the rmM-tree in the standard way.
5. The rmM-tree sends us to another leaf where we resume the scanning as in points 2 and 3. This time the answer should be found.

To compute $\texttt{child}(i, t)$, the tth child of node i (with $0 \leq t < 4$), we find with $t[\cdot].c$ the number of children of i, and compute $j = \texttt{fwd\_search}(i, -1, c-t-1)$, so we know that the parenthesis preceding the tth child of i is at the jth signature. This implies that the signature of the tth child of i is the $(j+1)$th signature.

5 Solving Matrix Sums and Multiplications

In this section, we address the algebraic operations of sum and multiplication.

5.1 Boolean sum

Given two kache-trees A and B representing matrices of the same size (so the trees have the same maximum depth), we can compute the Boolean sum $S = A+B$ with a sequential scan of the signatures of both kache-trees. We start with two read pointers a and b at the beginning of the signatures of A and B, and a write pointer s at the beginning of the signatures of the resulting kache-tree S. Then, we proceed as follows:

1. The bitwise-or of the next 5 bits starting at a and b is written into the result at position s. The three pointers are then advanced to the next signature. Note that the bit that marks last-level internal nodes must be the same in both signatures because we are summing two matrices of the same size.
2. If the obtained signature is not in the last-level of internal nodes, we check if each child exists in A and B.

– In case the child appears in both, we continue recursively generating the signatures of S.
– If the child occurs only in A, the next m signatures of the subtree of a are copied to S. The last signature of that subtree is computed as $j = \texttt{fwd\_search}(a, -1, 0)$, so $m = j - i + 1$. Finally, a and s increase by m.
– Otherwise, the child occurs only in B, so we append to S the corresponding subtree of B and increase the pointers b and s as above.

Note that every time a subtree is copied to the final result, we just need to copy a consecutive subsequence of bits. This is the main difference from the k^2-tree, where the subtree representation is broken into different portions along the levels. Therefore, the k^2-tree requires a more resource consuming level-order traversal to copy each subtree [2, Sec. 5].

5.2 Boolean multiplication

Regarding Boolean product, $M = A \times B$, we use the recursive divide-conquer algorithm of Section 3. In each step, we need to solve eight multiplications of the form $M_k = A_i \times B_j$, where A_i and B_j are submatrices of A and B, respectively. In the kache-tree, since both A and B support $\texttt{child}(i, t)$ operations, these multiplications can be computed recursively. Once we obtain those partial results, we have to combine them according to Section 3, by applying the sum operation.

For efficiency, we do not build the rmM-trees of the partial results M_k. As a consequence, during the sum of partial results, we cannot use rmM-trees to compute $\texttt{fwd\_search}(i, -1, 0)$ when copying a whole subtree. Instead, we compute it with a sequential scan of the sequence, that is, the same procedure that was used to scan the leaves of the rmM-tree (see Section 4.3). This does not increase the times much because we will traverse the sequence anyway in order to copy it. We only build the final rmM-tree of the result, as described in Section 4.2, so that the output of the operation is a fully-functional kache-tree.

6 Experimental Results

We now compare our proposal (`kc-tree`, with $b = 1024$) with the previous solution, `k`^{2}`-tree` [2], evaluating their performance at performing sums and multiplications of Boolean matrices. The implementations are in C/C++, and the `g++` compiler was used with options `-std=c++11 -O3`. All our experiments ran on an Intel(R) i7-8700k @4.70GHz, which has 6 cores (12 siblings), with 64GB of RAM. Each core has 32KB+32KB of L1- plus 300KB of L2-cache. The last-level cache has 12MB. We ran experiments under two different scenarios.

– The first one operates over synthetic matrices. We created 16 sets of $v \times v$ matrices, for $v \in \{1000, 4000, 8000, 16000\}$, with 1s uniformly distributed with densities $d \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. For the sets with $v \in \{1000, 4000, 8000\}$ we created 19 matrices and for those with $v = 16000$ we generated 9 matrices. Within each set, we performed both sum and multiplication of every pair of consecutive matrices, $M_i + M_{i-1}$ and $M_i \times M_{i-1}$.

		k^2-tree				kc-tree (b=1024)			
	v	$d=10^{-1}$	$d=10^{-2}$	$d=10^{-3}$	$d=10^{-4}$	$d=10^{-1}$	$d=10^{-2}$	$d=10^{-3}$	$d=10^{-4}$
Sum	1,000	5.10	1.07	0.14	0.02	4.99	0.97	0.12	0.01
	4,000	77.94	14.76	2.16	0.28	77.67	13.84	1.69	0.19
	8,000	320.86	58.57	8.33	1.19	310.63	55.08	6.49	0.74
	16,000	1,284.38	237.13	33.75	4.57	1,244.32	219.63	25.72	2.85
Mult	1,000	3,355.56	126.61	3.61	0.12	2,105.61	100.31	3.11	0.10
	4,000	213,325.00	8,337.57	234.56	6.71	134,149.00	6,568.05	199.20	6.24
	8,000	1,709,220.00	67,233.60	1,902.69	53.75	1,073,840.00	52,758.70	1,607.03	49.48
	16,000	13,863,200.00	537,481.00	15,331.70	432.80	8,588,380.00	423,232.00	12,953.30	396.39
Space	1,000	82.33	16.59	2.70	0.54	97.21	19.45	3.02	0.47
	4,000	1,311.54	261.71	40.08	5.59	1,551.01	309.34	47.21	6.42
	8,000	5,243.45	1,045.94	159.51	21.69	6,201.42	1,236.88	188.47	25.46
	16,000	20,969.54	4,182.88	637.10	86.05	24,801.23	4,947.04	753.32	101.57

Table 1. Experiments on the synthetic scenario. Average times for sum and multiplication are shown in msec; the average space of the source matrices is in KB.

– In the second scenario, we took four real matrices of varying size from the web graphs at https://law.di.unimi.it/datasets.php, namely `cnr-2000`, `eu-2005`, `uk-2014-host`, and `enwiki-2023`. To account for typical operations of interest we computed, for each matrix M, $M + M^T$ (which connects the nodes regardless of the direction), $M \times M$ (which detects paths of length 2, i.e., node pairs (x, y) such that $x \to z \to y$ occurs in the graph for some node z), and $M \times M^T$ (which detects nodes pointing to the same node, i.e., node pairs (x, y) such that $x \to z \leftarrow y$ occurs in the graph for some node z).

Synthetic scenario. Table 1 shows the average elapsed times needed to perform each sum or multiplication, in milliseconds. We also show the average space required, in KB, to represent the source matrices from each set.

The `k²-tree` space usage is almost exactly as expected from dv^2 points distributed uniformly on a $v \times v$ matrix, $2 \log_2(1/d)$ bits per 1 in the matrix.[6] The `kc-tree` uses 12%-18% more space than the `k²-tree`, which is below the 25% we would expect from using 5-bit signatures instead of 4-bit ones. This is a consequence of using a relatively large rmM-tree leaf size $b = 1024$,[7] yielding an extra space under 1% on the `kc-tree`, compared to the 6% extra space of the *rank*/*select* data structures used by the `k²-tree`.

On the other hand, the `kc-tree` is considerably faster than the `k²-tree` in almost all cases. In sums, the difference is larger as the matrices are sparser, because as the density decreases there are more opportunities for copying whole submatrices, which as explained the `kc-tree` can do more efficiently. For ex-

[6] The matrix is essentially full up to level $\log_{k^2}(dv^2)$ and, from there to the last level, $\log_{k^2} v^2$, each 1 basically induces a distinct path. Thus, each 1 induces $\log_{k^2}(1/d) = \frac{1}{2}\log_k(1/d)$ k^2-bit signatures, which is the dominant term in the space.

[7] To tune b, we tried values $b \in \{256, 512, 1024, 2048, 4096\}$ in the synthetic scenario: $b = 1024$ offered the best space/time tradeoff in practice.

		cnr-2000	enwiki-2023	eu-2005	uk-2014-host
	nodes (v)	325,557	6,625,370	862,664	4,769,354
	edges (#1s)	3,216,152	165,206,104	19,235,140	50,829,923
k²-tree	Space $\|M\|$	1.45	360.89	10.04	59.74
	Time $M + M^T$	56.94	17,082.80	402.18	2,849.48
	Time $M \times M$	6.13	48,406.20	115.20	5,811.81
	Time $M \times M^T$	300.67	–	2,524.89	–
kc-tree	Space $\|M\|$	1.71	426.84	11.87	70.66
	Time $M + M^T$	49.89	13,416.60	356.57	2,132.31
	Time $M \times M$	4.14	48,219.10	89.58	5,077.20
	Time $M \times M^T$	102.69	–	1,118.66	–
Ratio	Space $\|M\|$	0.85	0.85	0.85	0.85
	Time $M + M^T$	1.14	1.27	1.13	1.34
	Time $M \times M$	1.48	1.00	1.29	1.14
	Time $M \times M^T$	2.93	–	2.26	–

Table 2. Experiments on real scenario. Times for sum and multiplication are shown in msec and sec respectively; the space of the source matrices is in MB. Ratio indicates the value `k²-tree`/`kc-tree`. The product $M \times M^T$ did not finish for `enwiki-2023` and `uk-2014-host` for lack of main memory space.

ample, on the largest matrices, the `kc-tree` is from 3% to 60% faster than the `k²-tree` (i.e., `k²-tree`/`kc-tree` = 1.03 to 1.60) as the density decreases.

Multiplications are orders of magnitude more expensive than sums, taking hours on the largest matrices. On the denser matrices, which are the most expensive to multiply, the `kc-tree` is around 60% faster than the `k²-tree`. This speedup decreases with the density, reaching a point where the `kc-tree` is just around 8% faster. The decrease can be attributed to our need, when skipping over a subtree that is multiplied with a zero submatrix, to scan subtrees in order to find where they finish, which is not an issue in the `k²-tree` representation.

Real scenario. Table 2 shows results on the real-life matrices. Those are very large and have very low density, from $d \approx 2 \cdot 10^{-6}$ to $3 \cdot 10^{-5}$. The space usage of the `k²-tree` is much lower than $2\log_2(1/d)$, which was an excellent predictor on uniformly distributed matrices; this shows that the k^2-tree representation is very efficient in exploiting clustering of the matrix. The space of our `kc-tree` is always around 18% over that of the `k²-tree`.

While the advantage of the `kc-tree` over the `k²-tree` seems to decrease on non-uniform matrices, it is still significant: the `kc-tree` is 13%–34% faster than the `k²-tree` for sums, and 0%–48% faster on multiplications of the form $M \times M$. Interestingly, the difference becomes much larger on multiplications of the form $M \times M^T$, where the `kc-tree` is 2 to 3 times faster than the `k²-tree`. This suggests that `kc-tree` outperforms the `k²-tree` more sharply when the output is larger: $M \times M^T$ tends to produce bigger outputs than $M \times M$; indeed, for lack of main memory space our experiments ran only on the two smaller matrices.

The speedups are largely attributable to the reduced amount of cache misses: `perf` shows that the `k²-tree` generated 31%–42% and 7%–225% more cache-misses than the `kc-tree` on sums and multiplications, respectively. The only

exception is the sum in the smallest dataset (which fits in less than 2MB) where the `k2-tree` produces 40% less cache misses than the `kc-tree`.

7 Conclusions and Future Work

We have introduced *kache-trees*, a new representation of k^2-trees that, instead of the original level-wise deployment of nodes, traverses the nodes in depth-first order. This poses various new challenges, like efficiently distinguishing the last-level k^2-tree nodes, which we manage by slightly increasing the space — by less than 20% in practice. Navigating the tree also requires more operations, because we must build on parentheses sequences instead of just *rank*/*select* on bitvectors. In reward, the depth-first order yields the important advantage of making navigation operations more cache-friendly. We demonstrate this aspect by experimentally comparing our new representation with standard k^2-trees to represent sparse Boolean matrices, and obtaining speedups of up to 60%, even when our traversal operations require more operations.

Not only k^2-trees have many more applications than representing Boolean matrices, and our new kache-trees are likely to outperform them in all those, but our technique can also be used to represent general cardinal trees (of low arity) in cache-friendly form. In this case, leaves may need to be explicitly represented with k zeros instead of extending the signatures of internal nodes by one bit.

There are several lines of future work. One is to complete the implementation of the algebra. This includes, most prominently, to implement the transitive closure operation, where we expect kache-trees to excel because the resulting matrices tend to be considerably denser than the original one.

In the longer term, we aim to produce a fully cache-oblivious implementation of Boolean matrices. This is currently not true because the rmM-tree is deployed in heap format, which (even if it is much smaller than the sequence of signatures) makes subtrees not represented compactly in memory. We plan to change its representation to a depth-first order. This will imply that, whenever a submatrix has $O(B)$ points, it will be represented within a contiguous memory area of $O(B)$ rmM-tree nodes and another one of $O(B)$ signatures. Therefore, summing two matrices of n and m 1s stored in external memory with a(n unknown) block size of B, will require $O((n+m)/B)$ I/Os, whereas multiplying them will require $O((n+m+r)/B)$ I/Os, where r is the number of 1s in the product. This matches current results on plain representations of sparse matrices [6, 9].

Finally, we aim to explore the possibility of stopping the recursive decomposition into submatrices at sizes below $r \times r$, for some moderate r, and represent those using Huffman codes. This reported important space reductions on various k^2-tree applications [4], but poses some challenges when implementing a matrix algebra, where new matrices are generated on the fly. It may also be less cache-friendly for not so small r, due to the Huffman tables. Encodings that are inferior to Huffman, but are local and easily computed on the fly, like differentially encoding the positions of the 1s, can be a better alternative. Special procedures to operate pairs of encoded submatrices must also be developed.

References

1. D. Arroyuelo, A. Gómez-Brandón, and G. Navarro. Evaluating regular path queries on compressed adjacency matrices. In *Proc. 30th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 35–48, 2023.
2. D. Arroyuelo, A. Gómez-Brandón, and G. Navarro. Evaluating regular path queries on compressed adjacency matrices. *The Very Large Databases Journal*, 34:article 2, 2025.
3. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
4. N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of Web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.
5. D. R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.
6. M. Dusefante and R. Jacob. Cache oblivious sparse matrix multiplication. In *Proc. 13th Latin American Theoretical Informatics Symposium (LATIN)*, pages 437–447, 2018.
7. A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for declarative large-scale machine learning. *Communications of the ACM*, 62(524):83–91, 2019.
8. Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.
9. N. Gleinig, M. Besta, and T. Hoefler. I/O-optimal cache-oblivious sparse matrix-sparse matrix multiplication. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 36–46, 2022.
10. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
11. J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
12. G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.
13. G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.