Set Operations over Compressed Binary Relations[☆]

Carlos Quijada-Fuentes[a,*], Miguel R. Penabad[b], Susana Ladra[b], Gilberto Gutiérrez[a]

[a] *Universidad del Bío-Bío, Facultad de Ciencias Empresariales, Chillán, Chile*
[b] *Universidade da Coruña, Facultade de Informática, A Coruña, Spain*

Abstract

Binary relations are commonly used to represent relationships between real-world objects. Classical representations for binary relations can be very space-consuming when the set of elements is large. In these cases, compressed representations, such as the k^2-tree, have proven to be a competitive solution, as they are efficient in time while consuming very little space. Moreover, k^2-trees can successfully represent both sparse and dense binary relations, using different variants of the technique.

In this paper, we propose and evaluate algorithms to efficiently perform set operations over binary relations represented using k^2-trees. More specifically, we present algorithms for computing the union, intersection, difference, symmetric difference, and complement of binary relations. Thus, this work extends the functionality of the different variants of the k^2-tree representation for binary relations.

Our algorithms are computed directly over the compressed representation, without requiring previous decompression, and generate the result in compressed form. The experimental evaluation shows that they are efficient in terms of space and time, compared with different baselines where the binary relations are represented in plain form or require a previous decompression to perform the set operation.

Keywords: Compact data structures, Compressed binary relations, set operations, k^2-trees

1. Introduction

A binary relation between two sets X and Y is formally defined as a subset $R \subseteq A \times B$. In case that $(x, y) \in R$, or xRy, we say that element $x \in X$ is related with element $y \in Y$. Binary relations have been used as a common abstraction for representing relations between objects of the same collection or among objects of two different collections of different nature. Graphs, trees, strings, among others, can also be seen as binary relations. There are many examples of data structures that can be regarded as binary relations, both low-level and complex ones. For instance, we can define an inverted index (Baeza-Yates and Ribeiro-Neto, 1999; Zobel and Moffat, 2006) as a

[☆] A preliminary partial version of this paper appeared in Proc. DCC 2015, pp. 373–382.

[*] Corresponding author: Quijada-Fuentes Carlos. e-mail: caquijad@egresados.ubiobio.cl Address: Avenida Andrés Bello 720, Chillán, Chile.

Email addresses: caquijad@egresados.ubiobio.cl (Carlos Quijada-Fuentes), penabad@udc.es (Miguel R. Penabad), sladra@udc.es (Susana Ladra), ggutierr@ubiobio.cl (Gilberto Gutiérrez)

binary relation between the vocabulary of terms and the documents where they appear, and Web graphs can be represented as the relation between web pages in the Web.

Any representation of a binary relation R must answer basic queries, such as determining whether two elements $x \in X$ and $y \in Y$ are related, that is, if xRy; counting or listing all elements in Y related with a given element $x \in X$, that is, all $y \in Y \mid xRy$; or counting or listing all elements in X related with an element $y \in Y$, that is, all $x \in X \mid xRy$. Taking as an example an inverted index, we must support queries such as determining whether a word appears in a document or not, counting/listing all documents where a word appears, or counting/listing all words included in a document.

Moreover, any representation of binary relations must also support other operations, such as the well-known set operations: union, intersection, difference, or complement of binary relations. For instance, if we represent two snapshots of the same Web graph at two different instants of time as two binary relations, we may want to know which pages were connected in any instant of time, which pages were connected in the first instant of time but not in the second one, which pages were connected at both instants of time, etc.

Set operations are also of great interest when using binary relations to represent relationships among objects in Geographic Information Systems (GIS). Standards ISO 19107:2003 (Geographic information – Spatial schema) and ISO/TS 19103:2015 (Geographic information – Conceptual schema language) include definitions for these set operations (`ST_Union`, `ST_Intersection`, `ST_Difference`, and `ST_SymDifference` operations). Consistent with these specifications, OpenGIS standard OGC 06-104r4 (OGC, 2010) (Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option) requires that these operations shall be available for all types of geometry, and be implemented under set-theoretic semantics. Some implementations of OGC standards, such as PostGIS (http://www.postgis.net), already implement these operations. We are not comparing the efficiency of PostGIS with that of our proposal in this work because PostGIS uses a vectorial data model, while we are using what would be considered a raster approach to represent geographic information.

The best-known representation for a binary relation R over X and Y is the relation matrix $\{r_{ij}\}$ of size $|X|\times|Y|$. It is a binary matrix where a cell r_{xy} contains a 1 iff xRy, that is, if $x \in X$ and $y \in Y$ are related, and a 0 in other case. All the previous queries and set operations can be solved easily using this representation, some of them being very efficient, such as determining if two elements are related or not, as it only requires accessing to one cell of the matrix. However, in case that X or Y are large but the binary relation is sparse[1], this representation becomes impractical in terms of space requirements. Thus, it cannot be used for real large datasets. Instead of having a matrix, we can also represent a binary relation using lists representation, where, for each $x \in X$, we store the list of all $y \in Y \mid xRy$. A list representation is more suitable for sparse binary relations, but in general, a plain list representation is still a high space-demanding representation for large binary relations.

To overcome the large space requirements needed by traditional representations, different compressed representations and compact data structures have been proposed for representing binary relations. One of the most complete representations for binary relations was proposed by Barbay et al. (2013), who studied the representation of binary relations using succinct data structures, while supporting a wide set of operators efficiently. They also considered data structures

[1]We say that a binary relation R over X and Y is *sparse* if the number of related elements $(x, y) \in R$ is low compared to the number of possible related elements, that is, $|X \times Y|$. We say that a binary relation is *dense* in case that the number of related elements is high.

for binary relations and adaptive algorithms on these data structures (Barbay et al., 2007).

Other works exploited the compressibility of the relation matrix to compactly represent binary relations while supporting operations in an efficient way. Originally designed as an ad-hoc compressed representation for Web graphs, Brisaboa et al. (2014) proposed the k^2-tree, based on a compact tree structure that takes advantage of large empty areas of the adjacency matrix of the Web graph. The k^2-tree supports basic queries, such as determining if two Web pages are connected or obtaining the list of pages that point to or are pointed by a specific Web page, as well as more sophisticated queries, such as obtaining all the links among two specific domains. However, there are operations that are not supported by the original work, such as set operations of k^2-trees. k^2-trees have been used in other scenarios, such as geographical and RDF data, or images (de Bernardo et al., 2013; Álvarez-García et al., 2017). The viability of the usage of k^2-tree in these scenarios depends on the operations and queries that it supports. Hence, operations such as the union, intersection, difference, or complement should be also supported by the k^2-tree to be considered as a fully functional representation of binary relations.

Thus, in this work, we propose efficient algorithms to extend the functionality of k^2-trees for binary relations. More concretely, we present algorithms that perform set operations directly over the compressed representation of the binary relations and generate the result as a k^2-tree. Furthermore, we ran an intensive set of experiments that evaluate the performance of our algorithms, considering real datasets and several baselines. The paper is organized as follows. In Section 2, we revise the k^2-tree structure, describing its two variants: one designed for sparse binary relations and other designed for dense binary relations. Section 3 presents the algorithms for performing set operations over the first variant of the k^2-trees and Section 4 for the second variant. Section 5 includes the experimental evaluation of the proposed algorithms. We conclude in Section 6 with some discussion of the results and some perspectives on future work.

2. Previous work: k^2-trees

In this section, we will describe the k^2-tree representation for binary relations, which was originally designed for compressing Web graphs (Brisaboa et al., 2014), and has been extensively used in recent years to represent binary relations such as Web graphs and social networks (Claude and Ladra, 2011), raster data (de Bernardo et al., 2013), or RDF datasets (Álvarez-García et al., 2017). It basically represents a binary matrix following a quadtree strategy (Samet, 2006) and using succinct representation for trees (Jacobson, 1989) combined with compact representation of integers (Brisaboa et al., 2013). We first describe the original k^2-tree representation, which is better applied for sparse binary relations, and then describe one variation of the method, denoted by k^2-tree1, which is more suitable for dense binary relations.

2.1. k^2-trees for sparse binary relations

The k^2-tree method represents a binary matrix using a non-balanced k^2-ary tree. Given k and a squared matrix of size n, with n a power of k,[2] it first subdivides the matrix into k^2 submatrices of the same size, that is, k rows and k columns of submatrices of size n^2/k^2. Each of the resulting k^2 submatrices is then included as a child of the root node and its value is 1 iff there is at least one

[2]In case of general binary relations, where the relation matrix has size $n_X \times n_Y$, with $n_X = |X|$ and $n_Y = |Y|$ not necessary power of k, the matrix is expanded to the size $n' \times n'$, where n' is the smallest power of k greater than n_X and n_Y. It can be proven that this expansion of the matrix requires negligible extra space (Brisaboa et al., 2014).

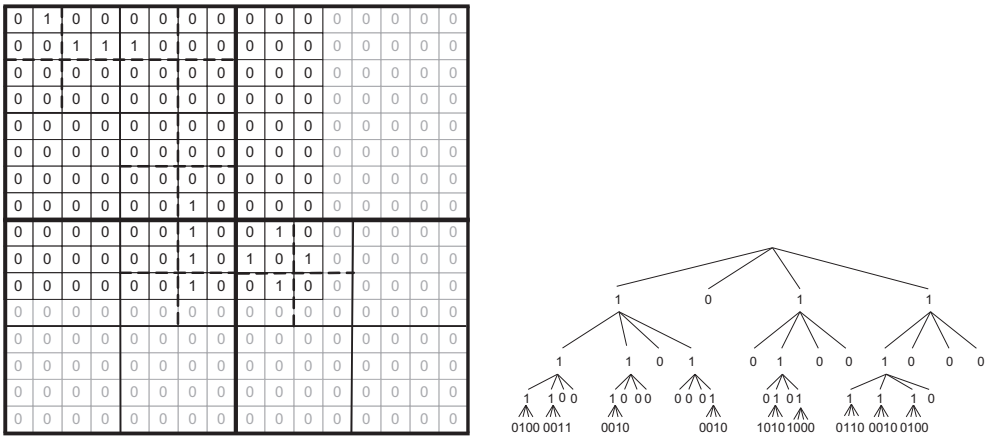


Figure 1: A k^2-tree representation (right) for an example of binary matrix (left).

1 in the cells of the submatrix. A 0 child means that the submatrix has all 0s and hence the tree decomposition ends there. Once the first level of the tree, with the children of the root, has been built, the method proceeds recursively for each child with value 1, until it reaches submatrices full of 0s or the cells corresponding to the original matrix. Fig. 1 shows an example of this subdivision (left) and the resulting k^2-ary tree (right).

The previous description of the data structure is just conceptual. The k^2-ary tree is not really stored as illustrated, but compactly represented using two bit arrays: T (tree) and L (leaves). T stores all the bits of the k^2-tree except those in the last level following a level-wise traversal. Bitmap L stores the last level of the tree. Additionally, an auxiliary structure is created over T that allows the navigation through the compact representation of the tree, that is, to travel down from the position of a node to the start position of its children. More concretely, this auxiliary structure supports efficient computation of *rank* queries (González et al., 2005), that is, counting the number of bits set up to a certain position, which is the basis of the navigation in the k^2-tree.

One of the most important features of the k^2-tree representation is that it is a compressed representation of a binary relation that not only supports efficiently the most basic primitives, being extremely fast when determining if two elements are related or not, or retrieving the related elements of one given element. It also supports efficient range queries inside the matrix, which allows the restriction of the relation to subsets of the original sets.

However, there are other operations that are common for binary relations and have not been described for k^2-trees. Thus, in Section 3 we will propose efficient algorithms to perform set operations for binary relations that are represented using k^2-trees.

2.2. k^2-tree1s for dense binary relations

Due to the decomposition strategy of the original k^2-tree, it performs very well when the matrix it represents has a relative small number of ones, and they are clustered. It takes full advantage of compression by representing large submatrices of zeros by a simple 0. However, if the number of ones grows, k^2-tree's behavior worsens, because it needs more bits to represent a simple one. In order to overcome this problem, k^2-trees with compression of ones were developed

(de Bernardo et al., 2013). The basic idea is to represent uniform areas (either *black* zones of ones, or *white* zones of zeros) by a 0, and areas with mixed ones and zeros (*gray* areas) by a 1, adding a way to distinguish black and white areas.

More specifically, in addition to bitmaps L and T, we add a bitmap T' to distinguish black and white zones. In this way, we implement an alphabet of size 3 instead of 2. A 1 in T represents a gray area, while a 0 represents a black or white area, and T' will have a bit for each 0 in T with a value of either 0 (white zone: all zeros) or 1 (black zone: all ones). L remains without changes: each value represents directly the bit (0/1) of the original matrix.

Let us name k^2-tree1 this implementation of k^2-tree with compression of ones. Figure 2 shows a conceptual k^2-tree1. Note how uniform areas (both quadrants on the right) are represented by just one node of the conceptual tree.

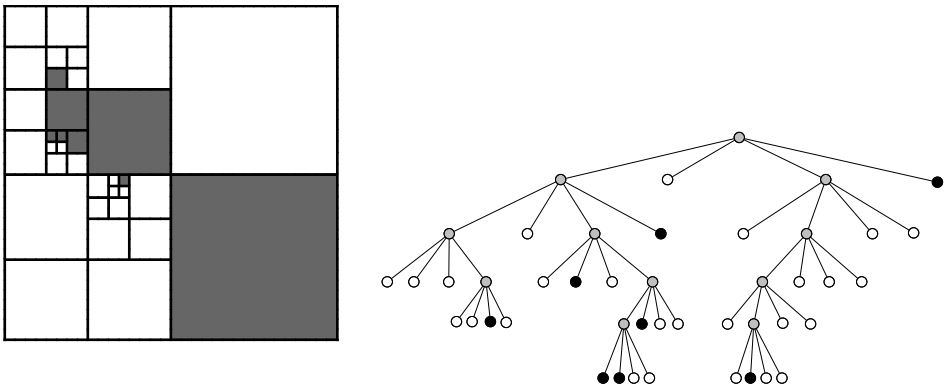


Figure 2: A conceptual k^2-tree1, where zones of ones are also compressed.

As k^2-tree1 can be regarded as a compact and efficient representation of a quadtree (Samet, 2006), some existing works studied this problem before for different implementations of quadtrees (Shaffer and Samet, 1990; Lin, 1997). The algorithms we propose are particular for the implementation of k^2-tree1, which use succinct data structures.

3. Set operations over k^2-trees

We first present the algorithms for computing the union, intersection, difference, symmetric difference, and complement of binary relations using the original k^2-tree representation. These algorithms receive k^2-trees as input, and generate a new k^2-tree which is returned as result of the procedure. Thus, this differs from the original functionality of the k^2-tree, where queries return a boolean value or a list of elements.

Before describing each algorithm in detail, we include here some common variables and assumptions that all of the procedures share. The algorithms receive as input the bitmaps of the k^2-trees that represent the input binary relations. Denoting R_A and R_B these input binary relations between two sets X and Y, then we will use bitmaps A and B, which correspond to the concatenation of the bitmaps from their k^2-tree compact representation, that is, $A = T_A||L_A$ and $B = T_B||L_B$. We denote C the output bitmap of the resulting k^2-tree. Auxiliary *rank* structures

Algorithm 3.1 Union over k^2-trees

```
1:  Q.Insert(⟨0, 1, 1⟩)
2:  pA ← 0, pB ← 0
3:  while not Q.Empty() do
4:      ⟨l, rA, rB⟩ ← Q.Delete()
5:      for i ← 0 . . . k² − 1 do
6:          bA ← 0, bB ← 0
7:          if rA = 1 then
8:              bA ← A[pA]
9:              pA ← pA + 1
10:         end if
11:         if rB = 1 then
12:             bB ← B[pB]
13:             pB ← pB + 1
14:         end if
15:         C ← C‖(bA ∨ bB)
16:         if (l < H) ∧ (bA ∨ bB = 1) then
17:             Q.Insert(⟨l + 1, bA, bB⟩)
18:         end if
19:     end for
20: end while
```

over this bitmap will be built afterwards and are not considered as part of the algorithms. We consider bitmaps A, B, and C as global variables in the algorithms. In most of the algorithms we will use pointers that reference the next bit to read in each bitmap, namely pA and pB. In case of unary set operations, the algorithm requires just bitmap A and pointer pA. Some algorithms also use a temporary bitmap t with size k^2 bits, in order to store partial results. In addition we use l to denote the current level of the visited nodes at each k^2-tree, n to denote the input matrix size[3]; and $H = \lceil \log_k n \rceil$, is the height for the input k^2-trees.

The construction of the result C in the case of the union operation differs from the rest of the binary operations. When computing the union of two k^2-trees, C is generated as just one bitmap corresponding to the k^2-tree of the result of $R_A \cup R_B$. However, in the case of the intersection, difference, symmetric difference, and complement operations, C is built following a decomposition strategy by levels: we used an array containing H bitmaps $C[l]$, where each $C[l]$ $(0 < l \leq H)$ stores the bitmap corresponding to level l of the resulting k^2-tree. In addition, instead of two pointers pA and pB, we will use two arrays of pointers of size H where $pA[l]$ and $pB[l]$ reference to the next bit to read from bitmap A and B at level l. Those arrays of pointers avoid an abuse of *rank* functions to access child nodes. The k^2-tree representation of the result thus requires a concatenation of the H bitmaps $C[l]$, which is not included in the pseudocodes.

3.1. Union

Union algorithm traverses bitmaps A and B in a synchronized, breadth-first way. Algorithm 3.1 shows the pseudocode.

For the traversal of the bitmaps we maintain a queue, denoted by Q, where we store tuples $\langle l, rA, rB \rangle$. Values rA and rB will indicate whether the processed internal nodes from A and B, respectively, have children or not. The algorithm begins by inserting $\langle 0, 1, 1 \rangle$ tuple at Q, meaning that the root nodes of trees corresponding to A and B, which are at level 0, have children. Then, the algorithm processes the queue as follows until there is no tuple to process.

[3]We assume that the input matrix is a square matrix of size $n \times n$, being $n = max(|X|, |Y|)$, adding $n - min(|X|, |Y|)$ extra columns or rows with zeros if necessary.

Algorithm 3.2 Intersection(l, pA, pB) over k^2-trees

```
1: writesomething ← 0
2: for i ← 0 . . . k^2 − 1 do
3:     if l < H then
4:         if (A[pA[l]] ∧ B[pB[l]]) then
5:             t[i] ← Intersection(l + 1, A, B, pA, pB)
6:         else
7:             t[i] ← 0
8:             SkipNodes(A, pA, l + 1, A[pA[l]])
9:             SkipNodes(B, pB, l + 1, B[pB[l]])
10:        end if
11:    else
12:        t[i] ← A[pA[l]] ∧ B[pB[l]]
13:    end if
14:    writesomething ← writesomething ∨ t[i]
15:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
16: end for
17: if writesomething = 1 then
18:    C[l] ← C[l] || t
19: end if
20: return writesomething
```

Algorithm 3.3 SkipNodes(X, pX, l, s) for k^2-trees

```
newpos ← pX[l] + s ∗ k^2 − 1
nOnes ← rank(X, newpos) − rank(X, pX[l] − 1)
pX[l] ← newpos + 1
if (nOnes > 0) ∧ (l < H) then
    SkipNodes(X, pX, l + 1, nOnes)
end if
```

Every tuple corresponds to just one of four cases depending on rA and rB values: *Case 1:* $rA = 1$ *and* $rB = 1$. If both internal nodes have children, the algorithm adds a 1 at the end of bitmap C and the indexes pA and pB are incremented. *Case 2:* $rA = 0$ *and* $rB = 0$. If none of the internal nodes have children, the algorithm adds one 0 at the end of bitmap C. In this case, indexes pA and pB are not incremented. *Case 3 and Case 4: Just one* k^2*-tree has children.* Suppose that node from A does not have children ($rA = 0, rB = 1$), in this case, the algorithm copies the k^2 bits from B pointed by pB, adding the result to bitmap C. Note that in this case, just pB index is incremented. When $rA = 1, rB = 0$ we proceed analogously with exchanged roles between the nodes from A and B. For all cases, if the OR operation for a pair of children outputs a 1 and it is not the last level of the corresponding k^2-trees, a new tuple is inserted in the queue for further processing the children.

3.2. Intersection

Intersection algorithm performs a depth-first traversal by the different levels of the tree using bitmaps A and B to intersect, in a synchronized way, the corresponding subtrees of each pair of nodes, where the subtrees are processed from left to right at each level. Algorithm 3.2 shows the pseudocode. The first call to the algorithm is **Intersection**$(1, pA, pB)$, where all values of pA and pB are set to the initial node at each level.

Intersection algorithm compares the k^2 nodes at each step and considers one of three cases. *Case 1:* If the input nodes belong to the last level H, then we can compute the value of the intersection directly using an AND operator. *Case 2:* If one of the input values from A or B at level l is 0, then the result is 0, and we concatenate a 0 value at the temporary bitmap $t[i]$, $0 < i \leq k^2$. In this case, we need to update pointers $pA[l + 1], ..., pA[H]$ or $pB[l + 1], ..., pB[H]$,

Algorithm 3.4 Difference(l, pA, pB) over k^2-trees

```
1: writesomething ← 0
2: for i ← 0 . . . k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] ∧ B[pB[l]]) then
5:         if (l < H) then
6:             t[i] ← Difference(l + 1, pA, pB)) {Internal nodes}
7:         else
8:             t[i] ← A[pB[l]]∧ ∼ B[pB[l]]  {Last level}
9:         end if
10:    else if (A[pA[l]]) ∧ (∼ B[pB[l]]) then
11:        if (l < H) then
12:            t[i] = Copy(A, l + 1, pA, A[pA[l]]){copy subtree}
13:        else
14:            t[i] ← 1
15:        end if
16:    else
17:        skipNodes(B, pB, l + 1, B[pB[l]])
18:    end if
19:    writesomething ← writesomething ∨ t[i]
20:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
21: end for
22: if writesomething = 1 then
23:    C[l] ← C[l]||t
24: end if
25: return writesomething
```

Algorithm 3.5 Copy(X, pX, l, s) for k^2-trees

```
end ← pX[l] + s ∗ k^2 − 1
nOnes ← rank(X, end) − rank(X, pX[l] − 1)
C[l] ← C[l]||X[pX[l]..end]
pX[l] ← end + 1
if (nOnes > 0) ∧ (l < H) then
    Copy(X, pX, l + 1, nOnes)
end if
```

to omit the whole non-empty subtree. For this, we propose a function **SkipNodes**, which is described at Algorithm 3.3. *Case 3:* If both values are 1, then we have a "candidate" to take value 1, but we need to check it by a deep traversal. Here we apply a recursive call with the next level as input parameter. As the algorithm returns if it added any bit to the bitmap $C[j]$ for a certain level $l < j \leq H$, this recursive call will determine if the intersection of the subtrees of both nodes is empty (the recursive call returns 0) or not (it returns 1), determining the value of the intersection for the original input values.

After examining the k^2 pairs of bits, the algorithm determines the need to add the values of the temporary bitmap t to the result at $C[l]$, being l the current level, and decides the return value of the function. When at least one bit of t is 1, t is added to $C[l]$ and the function returns 1; otherwise t is discarded and the function returns 0.

3.3. Difference

The computation of the resulting k^2-tree of the difference of two binary relations R_A, R_B (that is, $R_A - R_B$) can be performed in an analogous way as the intersection operation. When comparing two bits, the cases here are the following. *Case 1:* If the input value in the node corresponding to A is 0, then the result is 0 and this value is added to the temporary bitmap t. In case the value from B is 1, we need to update the values for pB. *Case 2:* If the input value from A is 1 and the input value from B is 0, we **Copy** the subtree from the current position to the last level H of

Algorithm 3.6 Complement(l, pA) over k^2-trees

```
1: writesomething ← 0
2: for i ← 0 . . . k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] = 1) then
5:         if l < H then
6:             t[i] ← Complement(l + 1, A, pA) {internal nodes}
7:         else
8:             t[i] ← ∼ A[pA[l]]
9:         end if
10:    else if (l < H) then
11:        FillIn(l + 1) {Fill in the subtree with 1 values}
12:    end if
13:    writesomething ← writesomething ∨ t[i]
14:    pA[l] ← pA[l] + 1
15: end for
16: if (writesomething = 1) then
17:    C[l] ← C[l]||t
18: end if
19: return writesomething
```

A concatenating the bits level by level in $C[j]$, with $l < j \leq H$ (see Algorithm 3.5). *Case 3:* If value from A is 1, and the value from B is 1, then we have a "candidate" to take value 0 but we need to check it by a deep traversal, computing with a recursive call if this difference is 0 or 1 for the subtrees. For the last level we compute directly the difference of the bits. Algorithm 3.4 shows the pseudocode. The first call to the algorithm is **Difference**$(1, pA, pB)$, where all values of pA and pB are properly initialized.

Like the intersection algorithm, after processing the k^2 entries, the temporary bitmap t is added to $C[l]$ only when t has at least one 1 (and the recursive call returns 1), otherwise t is discarded and the recursive call returns 0.

3.4. Complement

The complement of a k^2-tree A often requires to apply depth-first traversal from the actual input node to the nodes in the last level (see Algorithm 3.6); because of this, we used the decomposition strategy by levels. The process finalizes when all of the branches of bitmap A corresponding to the k^2-tree of the binary relation have been visited. Complement algorithm processes each node represented in A as follows. *Case 1:* If the input value is 0, then we used a function named *FillIn()* to turn the subtree completely full of 1's from the current node to the last level H. *Case 2:* If the input value is 1, then we have a "candidate" to take value 0 but we need to check it by a deep traversal. There is a possibility that some values change from 0 to 1 at last level H, and in this case upper levels do not change. In this case we apply a recursive call to compute the complement of the subtree. For the last level, we assign directly the negated value ($\sim$) of the input.

The inclusion of the temporary bitmap t in the result, as well as the return value of the recursive call, is managed similarly to the previous algorithms.

3.5. Symmetric Difference

Algorithm 3.7 specifies the operation of symmetric difference over two k^2-trees. Like the **Difference** and **Intersection** algorithms, it is performed in a depth-first fashion. Symmetric difference is equivalent to an exclusive OR of the two relation matrices rA and rB corresponding to the binary relations R_A and R_B, that is, the relation matrix rC of the binary relation R_C has a 1

Algorithm 3.7 SymmetricDifference(l, pA, pB) over k^2-trees

```
1: writesomething ← 0
2: for i ← 0 . . . k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] ∧ B[pB[l]]) then
5:         if l < H then
6:             t[i] ← SymmetricDifference(l + 1, A, B, pA, pB)) {Internal nodes}
7:         end if
8:     else if (A[pA[l]]) then
9:         if l < H then
10:            Copy(A, pA, l + 1, A[pA[l]])
11:        end if
12:        t[i] ← 1
13:    else if (B[pB[l]]) then
14:        if l < H then
15:            Copy(B, pB, l + 1, B[pB[l]])
16:        end if
17:        t[i] ← 1
18:    end if
19:    writesomething ← writesomething ∨ t[i]
20:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
21: end for
22: if (writesomething = 1) then
23:    C[l] ← C[l]||t
24: end if
25: return writesomething
```

Algorithm 4.1 Complement(A) over k^2-tree1s

```
1: C.T ← A.T
2: C.T′ ←~ A.T′
3: C.L ←~ A.L
```

in cell rC_{ij} if the corresponding cells rA_{ij} and rB_{ij} have different values, and 0 if they have the same value.

When the algorithm compares two bits of non-leaf nodes of the conceptual k^2-trees, we have the following cases: *Case 1:* when both bits are 1, the next level must be checked recursively to obtain the current output value. *Case 2:* when exactly one of the bits is a 1, the corresponding subtree is copied in $C[j]$ using the **Copy** function. *Case 3:* when both bits are 0, the output is 0 (in this case, there are no subtrees, so no recursion is needed).

Again, the partial result t and the return value of the recursive call are managed like in the previous algorithms.

4. Set operations over k^2-trees1s

We now describe the algorithms for computing the complement, union, intersection, difference, and symmetric difference for binary relations represented using the variant of k^2-trees with compression of ones, called k^2-tree1, which is suitable for dense binary relations.

4.1. Complement

The complement of a k^2-tree1 is straightforward: we just have to turn black areas into white ones (and vice versa), and to complement the bits in the leaves of the tree. All this information is stored in T' and L (T remains unchanged). Algorithm 4.1 shows the pseudocode.

4.2. Union, Intersection, Difference, and Symmetric Difference

Intersection, difference and symmetric difference in Section 3 were recursive, because the value of a result bit depended on the operation applied to the children of the current bit on the operand k^2-tree. In the case of k^2-tree1s, this same dependence applies to the union: if two (sub)trees A and B are gray areas, their union can still be a black zone (for example, if A is the complement of B). Thus, we will use a recursive algorithm to compute the union, intersection, difference, and symmetric difference of k^2-tree1s. In fact, the traversal of the trees is so similar for all of them, that we have combined the four set operations in the same algorithm (Algorithm 4.2). A table is included to indicate the next steps to execute, which depend on the operation to perform.

Input bitmaps A and B, and output array of bitmaps C will be used here for simplicity, like in Section 3 (omitting T, T' and L in the pseudocode). Notice that these bitmaps (including the vector t to store partial results) use now an alphabet of size 3, with elements: 0 (white area), 1 (gray area), and 2 (black area), being the value of 2 valid only in the T part of the bitmap (leaves can only have 0 or 1). Reading (updating) real bitmaps T and T' just requires reading (updating) bitmap T and, if its bit is a 0, also T'. Pointers pA and pB are also composed of two values, one for T and one for T'.

Algorithm 4.2 traverses all bits of the root nodes of A and B. Depending on the operation and the values of the bits, it produces an output bit and takes some other action (like copying a subtree or skipping one). In the case of gray areas (when both bits are 1), the operation continues recursively to the next level. Step 5 on Algorithm 4.2 shows how the operation must proceed depending on the input bits. All 9 possibilities for the combinations of two 3-valued bits are considered.

Let us focus first on the union operation. If both bits are 0, the result is 0, and if both are 2 (black areas) the result is also 2. Slighter complicated cases occur when gray areas appear (since union is commutative, some cases are symmetric, so we will not cover all of them). If the bit in A is 1 and the bit in B is 0, the result will be the tree whose root is the bit in A, so we copy the subtree in A to the result C (several levels can be copied, until either leaves or 0 bits are found). If A has a 2 (black zone) we know the output is 2, regardless of what B has. But if B has a 1, it has a subtree that must be skipped (it will produce no output). If both bits are 1, the result is computed recursively (child nodes are processed). During a recursive call, if t becomes full of ones, it is not added to the result. Instead, the function returns 2. The base case for the union is the output from two leaves, which is computed as the logical OR of their bits.

The intersection of R_A and R_B uses the same algorithm, but the base case and the action for each combination of two input bits is different from the union. The intersection of two leaves is the logical AND. In upper levels, the intersection of A and B behaves as follows: if one of them is a black area, the intersection is the other one; if one of them is a white area, the intersection is a white area and we must skip the bits of the other one; if both are gray areas, we must check recursively the value of the intersection.

For the difference $R_A - R_B$ the base case at the leaf level implies the difference between the bits. In upper levels, we highlight the following cases: if A is 0, the output is 0 and B, if present, must be skipped; if B is 0, the output is A; if B is a black zone, the output is 0, and A, if present, must be skipped; if A is a black zone and B is gray, then the output is 1, and we copy the complement of B in the output (this is put this way again for clarity, but we do not need to compute the complement of the whole tree). If both A and B are 1, we must check recursively the difference.

Finally, the symmetric difference uses the exclusive OR between the bits at the leaf level. In upper levels, as the name of the operation suggests, it behaves symmetrically. If both bits represent a uniform area (either white or black areas), the output is 0. If one of the bits is a 1 (gray area) and the other is a white area, the subtree corresponding to the 1 is copied on the output. If one of the bits is again a 1, but the other is a black area, the complement of the gray area is copied to the output. Only when both bits are 1, representing gray areas, the algorithm recursively checks for the symmetric difference output.

Algorithm 4.2 BinOp($\odot$,l) for union, intersection, difference, and symmetric difference over k^2-tree1s

1: $writesomething \leftarrow 0$
2: $full \leftarrow 1$
3: **for** $i \leftarrow 0 \ldots k^2 - 1$ **do**
4: **if** $l < H - 1$ **then**
5: Execute the following step depending on values $a = A[pA[l]]$, $b = B[pB[l]]$ and the binary operation $\odot$

a	b	$\odot$			
		$\cap$	$\cup$	$-$	$\triangle$
0	0	$t[i] \leftarrow 0$	$t[i] \leftarrow 0$	$t[i] \leftarrow 0$	$t[i] \leftarrow 0$
0	1	$t[i] \leftarrow 0$ **SkipNodes**$(B, pB, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**$(B, pB, l+1, 1)$	$t[i] \leftarrow 0$ **SkipNodes**$(B, pB, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**$(B, pB, l+1, 1)$
0	2	$t[i] \leftarrow 0$	$t[i] \leftarrow 2$	$t[i] \leftarrow 0$	$t[i] \leftarrow 2$
1	0	$t[i] \leftarrow 0$ **SkipNodes**$(A, pA, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**$(A, pA, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**$(A, pA, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**$(A, pA, l+1, 1)$
1	1	$t[i] \leftarrow$ **BinOp**$(\odot, l+1)$	$t[i] \leftarrow$ **BinOp**$(\odot, l+1)$	$t[i] \leftarrow$ **BinOp**$(\odot, l+1)$	$t[i] \leftarrow$ **BinOp**$(\odot, l+1)$
1	2	$t[i] \leftarrow 1$ **Copy**$(A, pA, l+1, 1)$	$t[i] \leftarrow 2$ **SkipNodes**$(A, pA, l+1, 1)$	$t[i] \leftarrow 0$ **SkipNodes**$(A, pA, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**(**Complem**$(A), pA, l+1, 1)$
2	0	$t[i] \leftarrow 0$	$t[i] \leftarrow 2$	$t[i] \leftarrow 2$	$t[i] \leftarrow 2$
2	1	$t[i] \leftarrow 1$ **Copy**$(B, pB, l+1, 1)$	$t[i] \leftarrow 2$ **SkipNodes**$(B, pB, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**(**Complem**$(B), pB, l+1, 1)$	$t[i] \leftarrow 1$ **Copy**(**Complem**$(B), pB, l+1, 1)$
2	2	$t[i] \leftarrow 2$	$t[i] \leftarrow 2$	$t[i] \leftarrow 0$	$t[i] \leftarrow 0$

6: **else**
7: $t[i] \leftarrow A[pA[l]]\{\wedge | \vee | \wedge \sim | \oplus\} B[pB[l]]$
8: **end if**
9: $writesomething \leftarrow writesomething \vee t[i]$
10: $full \leftarrow full \wedge (t[i] = 2)$
11: $pA[l] \leftarrow pA[l] + 1, pB[l] \leftarrow pB[l] + 1$
12: **end for**
13: **if** $writesomething = 1$ **then**
14: **if** $full = 1$ **then**
15: **return** 2
16: **else**
17: $C[l] \leftarrow C[l] \parallel t$
18: **end if**
19: **end if**
20: **return** $writesomething$

5. Experiments

In this section, we describe the experimental evaluation we have performed to analyze the behavior of the proposed algorithms. We have run our experiments over real data of different nature and compare our algorithms to different baselines.

We first describe the implementation details of our proposed algorithms and those used for the baselines we compare with, and describe the datasets we used. Next, we include a discussion on the results obtained for our algorithms when varying different properties of the data. Finally, we also analyze the scalability of our proposal and compare our techniques with other compressed representations that can be also used to represent binary relations. This last section is just to illustrate the efficiency of our approach in terms of the space and time used, but it is not a real comparison, as these compressed representations do not allow for the same functionality as k^2-trees offer when representing binary relations.

5.1. Implementation details of our algorithms

We use the original variant of the k^2-tree, which does not compress the last-level submatrices (Brisaboa et al., 2014), and configure the same k value for all the levels of the tree, setting $k = 2$. We do not use the hybrid approach nor the vocabulary-based approach to better analyze the behavior of each algorithm over the k^2-tree data structure without the influence of other parameters. We denote `kt` this variant.

To navigate the tree, we use an implementation for supporting *rank* operations that requires 5% of extra space on top of the bit sequence T and provides fast queries (González et al., 2005).

We use the same configuration for the k^2-tree1, which also compresses regions of ones. We denote `ktones` this variant.

5.2. Baselines

In the first set of experiments we evaluate our contribution in regards of extending the functionality of k^2-tree data structure when used for representing binary relations. In this way, we measure the performance of the proposed algorithms and compare them with two different baselines: *i)* the time required by the processes of decompressing the two binary relations, compute the result using plain list representation, and compressing the result, and *ii)* the time required when we use just lists for representing the input and output for the operation.

5.2.1. `kt-list-kt` / `ktones-list-ktones`

We denote as `kt-list-kt` the first baseline, where the binary relations are compressed as k^2-trees, and we want to obtain the result of the operation in the same format. Instead of using the proposed algorithms, we proceed in the following way: we obtain the lists of related elements for the input k^2-trees, we compute the operation over these lists, obtaining the result also as a list, and finally we generate the k^2-tree for the output.

This baseline is the most significant regarding the comparison of the obtained time results, as it allows us to determine if the proposed algorithms are more or less efficient than the process required in case we do not have the implementation of these set operations over the compressed representations. Notice that this procedure can be unaffordable in terms of space, at it requires maintaining the uncompressed binary relations in main memory.

We also consider the baseline `ktones-list-ktones`, which denotes the analogous procedure, but using k^2-tree1s instead of k^2-trees.

5.2.2. `lists`

For this baseline, we assume that the binary relation is represented using lists of related elements. More concretely, we consider the binary relation as its corresponding directed graph, where each pair of related elements represents an edge, such that the binary relation can be represented by the adjacency list of this graph (using 4-byte integers to represent each edge). The result of the set operation is also represented using a plain list.

It should be noted that this baseline has not been included for a direct comparison with the proposed algorithms, but as reference of the space required and the time consumed by this process within the `kt-list-kt` baseline.

We do not include here as baseline the representation of the binary relation using its matrix representation, as it is very space-demanding, especially for large sparse datasets[4].

5.2.3. *Compressed adjacency lists:* `qmx` *and* `rice-runs`

For the second set of experiments, where we study the scalability of our approach and its performance compared with that of other possible alternative approaches of representing binary relations in small space, we use as baselines two variants of compressed adjacency lists.

We use to different techniques for the compressed adjacency lists: i) `qmx` uses QMX technique (Trotman, 2014), which employs SIMD-instructions, coupled with an intersection algorithm that also benefits from SIMD-instructions (Lemire et al., 2016); and ii) `rice-runs`, which combines Rice coding (Witten et al., 1999) with run-length compression (Culpepper and Moffat, 2010; Transier and Sanders, 2010). These two alternatives present different spatio-temporal properties and are both competitive according to Claude et al. (2016). We use their own code, which is publicly available[5].

On one hand, `qmx` is extremely fast for long adjacency lists, as it can exploit SIMD instructions. On the other hand, `rice-runs` obtains very competitive results when there are large sequences of 1s consecutives in the input relation matrix, thanks to the use of run-length compression, boosting both compression and intersection speed. In this last case, intersections are performed merge-wise, and taking advantage of the fact that a run of 1 values can be skipped/decoded in a unique operation.

5.3. *Datasets*

We used two different real datasets for the experiments. We denote them as `uk-snaps` and `land-use`, and detail their properties in the following subsections.

5.3.1. `uk-snaps`

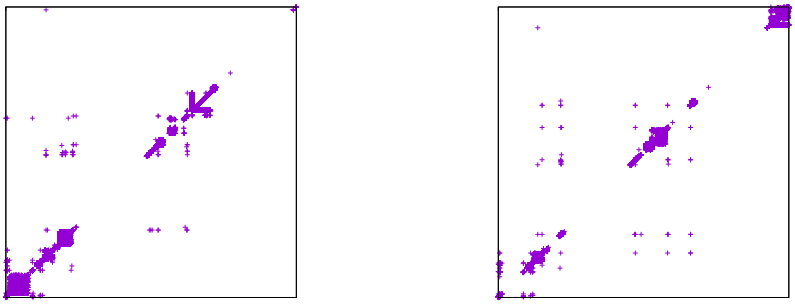
This first dataset (actually, a collection of datasets) corresponds to a series of twelve monthly snapshots of a Web graph, namely the .uk domain, collected by the Laboratory for Web Algorithmics[6] (Boldi and Vigna, 2004; Boldi et al., 2011, 2008).

The original graph, `uk-union-2006-06-2007-05`, was generated by combining 12 snapshots of the .uk graph, from June 2006 to May 2007. It contains 133.6 millions of nodes and 5.5

[4]The matrix representation for the larger matrixes of `uk-snap` dataset described in Section 5.3.1 would require more than 1 Petabyte.

[5]`https://github.com/migumar2/uiHRDC`

[6]`http://law.di.unimi.it`



(a) `snap00` (b) `snap11`

Figure 3: Preview of two of the `uk-snaps` snapshots.

billions of arcs, and occupies more than 22 Gigabytes when represented as a plain adjacency list (using 4-byte integers).

Each snapshot i can be regarded as a binary relation R_i over the set N, being N the nodes of the Web graph, that is, the Web pages contained in the .uk domain, where page p is related with page q, that is, pR_iq, if page p contains a hyperlink to page q. In this context, we will compute the union, intersection, difference, and symmetric difference for two different binary relations R_i and R_j, so that we can compute all the hyperlinks existing at snapshot i or snapshot j (union), hyperlinks existing at snapshot i and also at snapshot j (intersection), hyperlinks existing at snapshot i but not at snapshot j (difference), or those hyperlinks existing at snapshot i or at snapshot j but not at both snapshots (symmetric difference).

Note that each binary relation R_i related to snapshot i corresponds to a graph, where each Web page is a node and each hyperlink is an edge. It can be represented using an adjacency matrix, containing 1 values at those positions (p, q) where page p contains a hyperlink pointing to page q. Thus, the number of related elements, which we will denoted as m, corresponds to the number of edges of the graph, that is, the number of hyperlinks among all pages.

Figure 3 shows a sample of two of the snapshots, namely `snap00` (left) and `snap11` (right). Web pages are sorted lexicographically, thus, related elements are usually grouped together, as there exist more hyperlinks within page domains.

We did not use the complete snapshots for our first set of experiments, mainly in order to prevent main memory limitations when using the classical adjacency list representations to perform set operations (Section 5.5). However, we wanted to take into account different graph sizes in our second set of experiments (Section 5.6). For this reason, we built a collection of 6 different 12-matrix datasets. Each one was built by extracting 12 snapshots from the combined graph and restricting these snapshots to the first $1,000$; $10,000$; $100,000$; $1,000,000$; $10,000,000$; and $100,000,000$ nodes. We denote these datasets `1k`, `10k`, `100k`, `1m`, `10m`, and `100m`, respectively.

Table 1 shows the properties for each snapshot of the dataset with 1 million nodes (`1m`), which is the one used for the first set of experiments. We include the number of related elements in the binary relation (m), the percentage of related elements among all the possible ones ($m/n^2 \times 100$), and the sizes of each representation when using the classical adjacency list representation (`list`), the original k^2-tree representation (`kt`), and the k^2-tree representation that compresses also zones of ones (`ktones`). We omit from this table the plain matrix or bitmap representation of the

dataset, which would show the same size of 116 GB for all cases, since this representation is not affected by the density. It would not fit into main memory, so the times required to perform set operations with these matrices as input would be orders of magnitude larger.

Table 1: Properties of the `uk-snaps` dataset that corresponds to matrices of size $1,000,000 \times 1,000,000$. For each snapshot, we show its name, the number of links (m), the binary relation density, computed as the percentage of links of the total possible relations ($m/n^2 \times 100$), and sizes for each representation: adjacency lists (denoted by `list`), k^2-trees (denoted by `kt`), and k^2-trees compressing ones (denoted by `ktones`).

	m	density $m/n^2 \times 100$	`list` (bytes)	`kt` (bytes)	`ktones` (bytes)
`snap00`	2,404,620	0.00024%	13,618,492	1,462,364	1,824,072
`snap01`	1,360,019	0.00014%	9,440,088	855,900	1,072,996
`snap02`	3,200,913	0.00032%	16,803,664	2,129,416	2,662,548
`snap03`	3,395,695	0.00034%	17,582,792	2,559,828	3,285,056
`snap04`	2,350,289	0.00024%	13,401,168	1,721,128	2,201,560
`snap05`	2,526,547	0.00025%	14,106,200	1,701,508	2,157,840
`snap06`	1,507,033	0.00015%	10,028,144	889,680	1,095,900
`snap07`	1,950,981	0.00020%	11,803,936	1,089,980	1,315,704
`snap08`	1,772,026	0.00018%	11,088,116	1,027,872	1,257,796
`snap09`	1,767,230	0.00018%	11,068,932	968,788	1,169,936
`snap10`	2,665,612	0.00027%	14,662,460	1,504,804	1,847,972
`snap11`	1,989,568	0.00020%	11,958,284	1,117,508	1,365,672

Table 2 shows the properties for the collections used in the second set of experiments, where we analyze the scalability properties of our approach. For each dataset, we show the average values of their 12 snapshots.

Table 2: Average properties of `uk-snaps` datasets.

	n	m	density $m/n^2 \times 100$	`list` (bytes)	`kt` (bytes)	`ktones` (bytes)
`1k`	10^3	6,860	0.68603%	31,453	3,347	3,886
`10k`	10^4	85,128	0.08513%	380,524	39,667	46,468
`100k`	10^5	273,579	0.00279%	1,515,002	144,310	172,200
`1m`	10^6	2,240,878	0.00022%	12,963,523	1,419,065	1,771,421
`10m`	10^7	39,567,654	0.00004%	198,270,627	28,530,261	36,708,879
`100m`	10^8	143,421,531	0.00001%	3,802,858,042	456,395,798	609,148,714

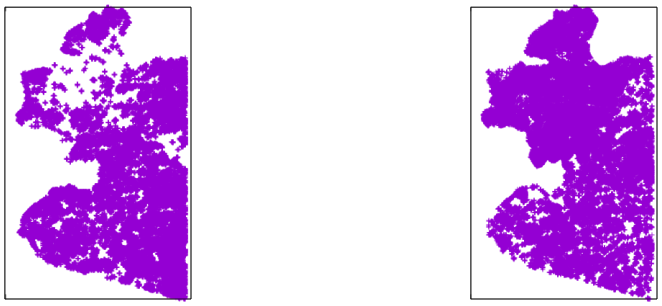
As we can see in Tables 1 and 2, this collection of datasets contains binary relations with a low number of related elements (m), as Web graphs contain few hyperlinks compared to the number of possible related Web pages. This low density of related elements causes that the binary matrix is sparse and that `kt` obtains better compression than `ktones`, and much better than the one required by the classical adjacency list. The average filesize for `kt` ranges from 9.74% to 14.43% of the space used by the binary file used by `list`, while `ktones` ranges from 11.67% to 18.55% of the same space.

5.3.2. land-use

This second dataset corresponds to a series of ten raster matrices of size $7,665 \times 4,085$, which represent information about the land usage of a region of Chile, more concretely, from the Comuna de Carahue, Region of Araucanía, using a 5-meter spatial resolution. These raster matrices indicate the type of usage: agricultural land (`ag`), native forest (`nf`), scrubland (`sc`), forest plantations of pines, eucalyptus, or others (`fp`), or bare ground with no vegetation (`bg`) in two different years (1986 and 2001).

Each raster matrix denoted by `tt_yyyy` can be regarded as a binary relation R over the set of possible geographical positions, where xRy if the use of the portion of land at position (x, y) was of type `tt` in year `yyyy`. Note that this binary relation can be represented using a binary matrix, where m here is the number of positions that were used for the given type of land use in the specific year.

In this context, we will compute the complement, union, intersection, difference, and symmetric difference for different binary relations, so that we can answer queries such as: which regions where not used for a given use type (complement), which regions were used for a given use type at any year (union), which regions were used for different use types (union), which regions were used for the same given use type in the two different years (intersection), etc.



(a) `ag_1986` (b) `fp_2001`

Figure 4: Preview of two of the `land-use` raster matrices.

Figure 4 shows a sample of two of the raster matrices containing the land use information, namely `ag_1986` (left) and `fp_2001` (right). In this case, related elements are distributed among the geographical space represented in the raster matrix.

Table 3 shows the properties for each raster matrix of this dataset. Again, we include the number of related elements in the binary relation (m), the percentage of related elements among all the possible ones ($m/n^2 \times 100$), and the sizes of each representation when using the classical list representation (`list`), the original k^2-tree representation (`kt`), and the k^2-tree representation that compresses also zones of ones (`ktones`).

In this scenario, the density of the binary relation is high, due to the nature of the dataset: each point of the region must be classified in one of the five land uses. Thus, the number of 1's in the binary matrix representing each raster matrix is high compared to the number of 0's, making `ktones` a better strategy to represent this dataset. Both outperform significantly the space required by the classical list representation, as `kt` requires less than 5% and `ktones` less than 2% of the space required by `list`.

Table 3: Properties of `land-use` dataset. All raster matrices are of size $7,665 \times 4,085$. For each raster matrix, we show its name, the number of marked points (m), the binary relation density, computed as the percentage of marked points out of the possible positions ($m/n^2 \times 100$), and sizes for each representation: plain lists of of marked points (denoted by `list`), k^2-trees (denoted by `kt`), and k^2-trees compressing ones (denoted by `ktones`).

	m	density $m/n^2 \times 100$	`list` (bytes)	`kt` (bytes)	`ktones` (bytes)
`ag_1986`	4,423,600	14.13%	17,725,072	818,544	187,276
`ag_2001`	3,437,825	10.98%	13,781,972	626,596	121,656
`nf_1986`	5,985,825	19.12%	23,973,972	1,073,284	166,164
`nf_2001`	4,026,725	12.86%	16,137,572	762,280	212,368
`sc_1986`	2,640,775	8.43%	10,593,772	510,160	162,548
`sc_2001`	2,861,100	9.14%	11,475,072	553,516	178,376
`fp_1986`	2,602,625	8.31%	10,441,172	491,984	134,268
`fp_2001`	5,683,000	18.15%	22,762,672	1,044,488	222,732
`bg_1986`	1,168,925	3.73%	4,706,372	226,680	73,172
`bg_2001`	813,100	2.60%	3,283,072	159,712	55,620

5.4. Experimental framework

All algorithms and baselines were implemented using C language and compiled with gcc version 6.3.0. The experiments were run on an isolated Intel® Xeon® ES2470@2.30GHz Processor with 20 MB of cache, and 64GB of RAM. It runs Debian 9.4 (stretch) with kernel 4.9.82 (64 bits).

We measured time to execute each set operation, without including any I/O time from/to secondary memory, that is, we do not include the time required to read/write each representation from/to the files where they were stored.

For each execution, time measuring was initiated immediately after loading in main memory the data structures to process (plain lists or compressed representations using k^2-trees), and before performing any operation. Time was stopped just after obtaining the result in the corresponding format. Reported results correspond to average times for 20 repetitions of the same execution.

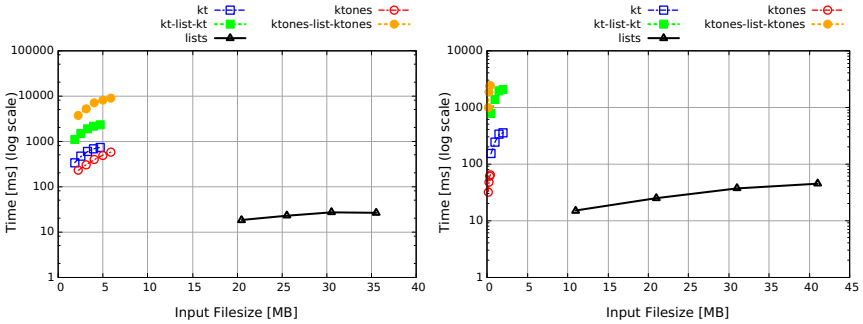
5.5. Analyzing the behavior of the proposed algorithms

In this section, we present the results obtained according to four different criteria: Input Size, Input Density, Output Density, and Input Compressibility. We describe each criterion in its own subsection. In these experiments the baselines described in Sections 5.2.1–5.2.2 and the datasets described in Tables 1 and 3.

5.5.1. Input Size

This criterion considers the size of the files that store the data structures of the input. The total size of the input corresponds to the sum of the sizes of the data structures that represent the input binary relations, either using plain lists (`list`) or k^2-trees in one of their two variants (`kt` or `ktones`).

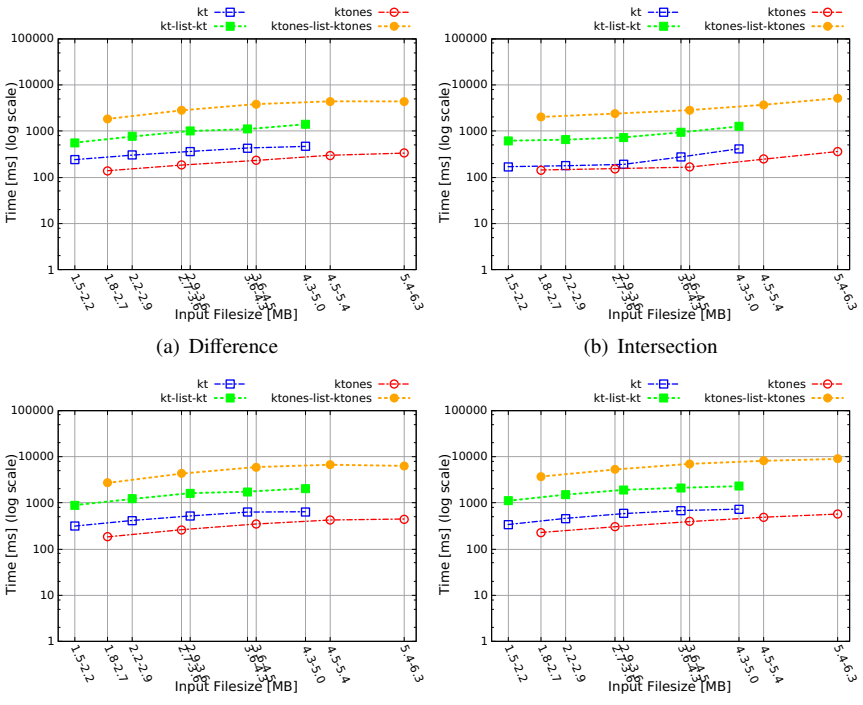
Using this criterion, values for lists are omitted on the graphs, as their size is much larger that the size of `kt` and `ktones`. This can be seen in Figure 5, where we show the results obtained



(a) Union in `uk-snaps` dataset (b) Union in `land-use` dataset

Figure 5: Time results obtained by `kt`, `ktones`, and the two baselines when computing union operations over `uk-snaps` and `land-use` datasets. The Input Size criterion is used for x-axis. `list` shows impractical space requirements.

.



(a) Difference (b) Intersection

(c) Symmetric Difference (d) Union

Figure 6: Times obtained for different set operations over `uk-snaps` dataset. We use the Input Size criterion for x-axis.

for union operations over the two datasets. We can see here that baseline `list` is not practical in real scenarios, with large datasets, but we include it in this empirical evaluation as reference.

Figures 6 and 7 show that when performing difference, intersection, symmetric difference, complement, and union operations, the proposed algorithms outperform those approaches that

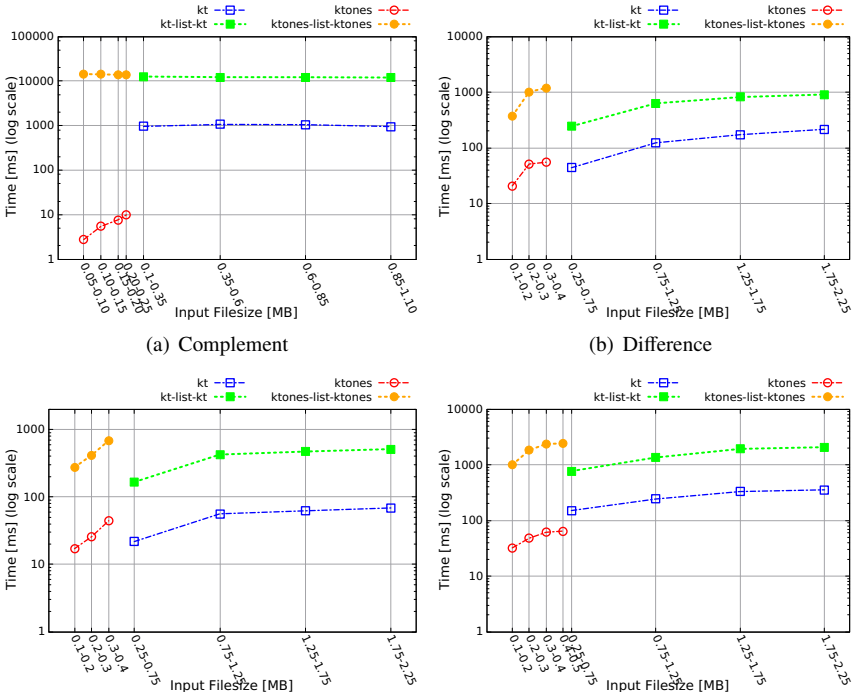


Figure 7: Times obtained for different set operations over `land-use` dataset. We use the Input Size criterion for x-axis.

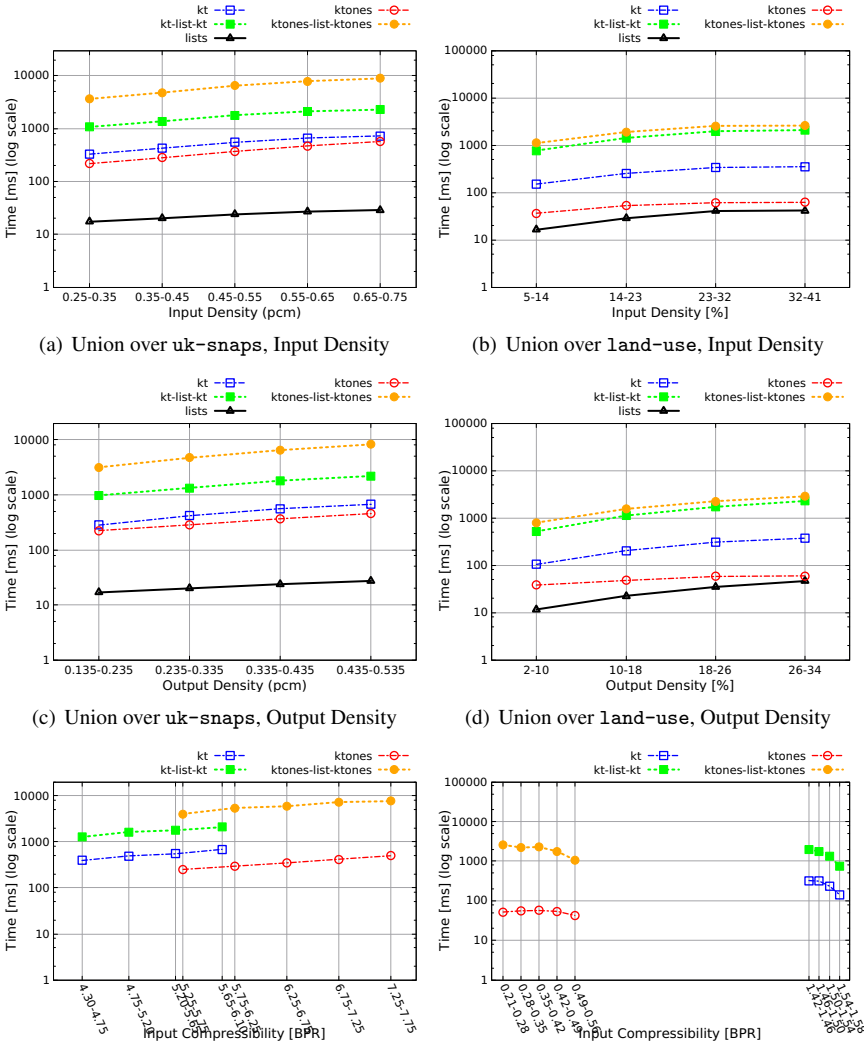
require decompressing the representation first (`kt-list-kt` and `ktones-list-ktones`). Our algorithms obtain time results that are approximately one order of magnitude faster.

Comparing `kt` and `ktones`, we can see that they have a similar behavior on all operations, when the considered dataset is `uk-snaps` (see Figure 6), being `ktones` always faster. Notice that the intersection operation has a slightly higher slope for larger input sizes. Considering `land-use` dataset, we can see in Figure 7 that `ktones` is also faster than `kt`, especially in the complement operation, where `ktones` can be up to 3 orders of magnitude faster. For the remaining operations, the difference is not so high. We can see that for this dataset the slope of the intersection is much higher than for `uk-snaps`. The time consumed by `kt-list-kt` and `ktones-list-ktones` is consistently higher than the time consumed by `kt` and `ktones`, and the shape of the curves in the graphic shows a similar pattern in their behavior. Due to this, for the following criteria we will only illustrate the results obtained for the union operation for both datasets (Figure 8) and the complement operation for `land-use` (Figure 9).

5.5.2. Input Density

This criterion corresponds to the percentage of related elements in the input binary relations. For a binary operation $R_A \odot R_B$ between two binary relations R_A and R_B over the sets X and Y, we define Input Density as the number of related elements in R_A plus the number of related elements in R_B divided by the number of total possible related elements ($|X \times Y|$).

Figure 8(a) shows that, in the case of `uk-snaps` dataset, time results increase when increas-



(a) Union over `uk-snaps`, Input Density

(b) Union over `land-use`, Input Density

(c) Union over `uk-snaps`, Output Density

(d) Union over `land-use`, Output Density

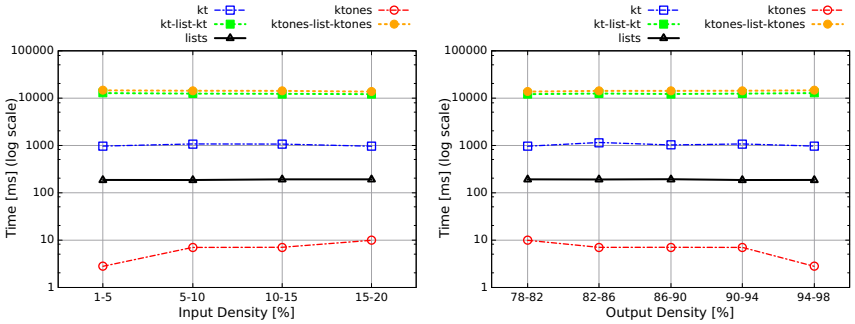
(e) Union over `uk-snaps`, Input Compressibility

(f) Union over `land-use`, Input Compressibility

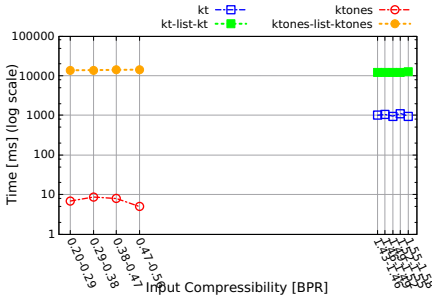
Figure 8: Times obtained for union operation over datasets `uk-snaps` and `land-use` using different criteria for x-axis.

ing the input related elements. We can also see that the behavior of `kt` and `ktones` is very similar. In this graphic we include the time taken by plain lists to perform the operation, which can be taken as a reference for the `kt-list-kt` and `ktones-list-ktones`. It shows that the compression and decompression time is higher for `ktones-list-ktones`, taking much larger times than `kt-list-kt`, which is expected as `kt` obtains better compression than `ktones` and this impacts the compression and decompression times.

In the case of `land-use` dataset, we can see in Figure 8(b) that times also increase when increasing the input related elements for the union operation. For difference and intersection operation the behavior is analogous. We show the complement operation in Figure 8(a), which



(a) Complement over `land-use`, Input Density (b) Complement over `land-use`, Output Density



(c) Complement over `land-use`, Input Compressibility

Figure 9: Times obtained for complement operation over dataset `land-use` using different criteria for x-axis.

obtains more constant times. The time taken by `ktones` for the complement is also very noticeable, showing an excellent performance compared to the remaining data structures, being more than one order of magnitude faster than the plain list representation.

5.5.3. Output Density

In this case, the Output Density criterion considers the number of related elements of the output of the operation. By measuring the output instead of the input, we analyze the results regarding the intrinsic difficulty of the problem.

Figure 8(c) shows an increase on the time consistent with the increment of the number of related elements of the output for union operation over the `snap-uk` dataset. Times taken by `kt` and `ktones` are very close for all cases. Including decompression and compression, we can see that again `ktones-list-ktones` is slower than `kt-list-kt`.

We can see in Figure 8(d) a clear advantage of `ktones` over `kt` for union operations over the `land-use` dataset. In this case, `ktones-lits-ktones` and `kt-list-kt` take similar times, although the latter is slightly faster.

From Figure 9(b), it draws our attention that times obtained for the complement operation are independent of the output density, except for `ktones`, which shows a strong decrement after an output density of 94%. This can be explained by the appearance of more "black areas" that are represented using very few bits and were complemented by just switching a bit in the bitmap of the `ktones`, while the remaining data structures must go through many more bits or elements

of a list. This ability of `ktones` to complement big "white areas" of zeros or big "black areas" of ones justifies the good behavior of this method. We could not perform the analysis of the complement operation for the `snaps-uk` dataset, as the snapshots are very sparse, and computing their complement requires unaffordable space for such a large dataset.

5.5.4. Input Compressibility

In this case, we use BPR (*Bits Per Related element*) as a criterion. Denoting as A_{fs} the file size of the matrix A, and m_A the number of related elements in A, and analogously for matrix B, BPR is computed as $\frac{A_{fs}+B_{fs}}{m_A+m_B}$. Notice that higher BPR values mean worse compression, as we need more bits for each related element. Thus, the Input Compressibility criterion allows us to evaluate the performance of the algorithms based on the compression level they achieve.

Figure 8(e) shows that, as BPR increases, so increases the processing time for union operation over `snaps-uk` dataset. Figure 8(f), on the contrary, shows a decrement on the times taken for the union operation over `land-use` dataset as the BPR increases. However, in Figure 9(c), we can see that the complement shows this decrement only for `ktones`, while the rest of the techniques keep almost constant times.

These results are consistent with those obtained for the Input Density criterion. Notice that the compressibility obtained by any of the variants of the k^2-tree is highly dependent on the clustering of the related elements, and it does not directly depend on the number of related elements (Brisaboa et al., 2014). For the datasets used in this experimental evaluation, Figure 10 shows the relationship between the number of related elements and the compression obtained. We can see that `ktones` obtains better compression than `kt` for the `land-use` dataset, obtaining its best result for the raster matrix with the larger number of related elements, and obtaining its worst result for the raster matrix with the lower number of related elements. On the contrary, `kt` obtains better compression than `ktones` for the `snaps-uk` dataset, obtaining its best result for one of the snapshots with lowest number of related elements, and obtaining its worst result for the snapshot with the largest number of related elements. This causes the behavior shown in the figures previously commented.

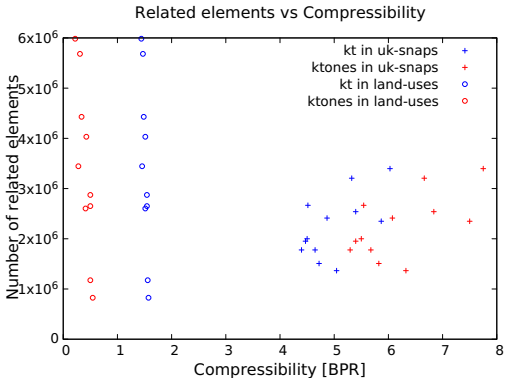
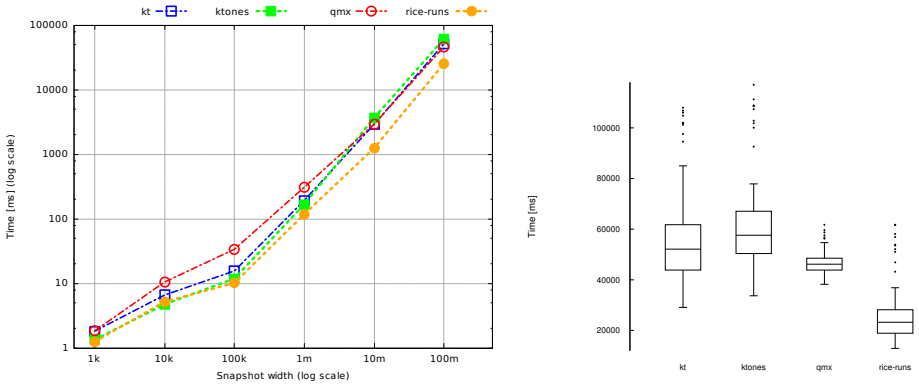


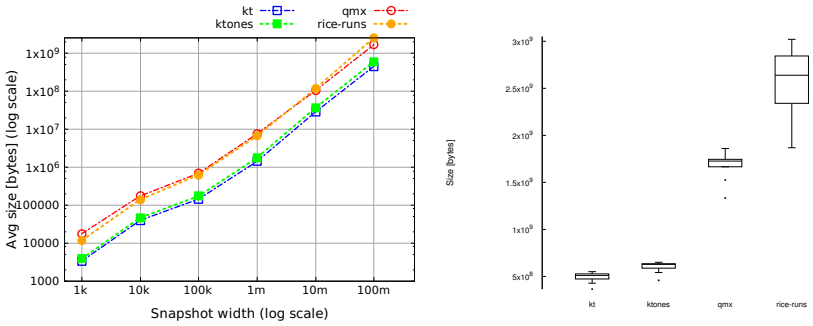
Figure 10: Number of related elements and compressibility of the binary relations for both datasets, `uk-snaps` and `land-use`.



(a) Times (log scale) varying input size

(b) Box plot for `100m` dataset

Figure 11: Times taken by compressed adjacency lists and our data structures when performing intersections: (a) for different dataset sizes, and (b) box plot for the dataset with 100 million nodes (`100m`).



(a) Sizes (log scale) varying input size

(b) Box plot for `100m` dataset

Figure 12: Sizes used by compressed adjacency lists and our data structures: (a) for different dataset sizes, and (b) box plot for the dataset with 100 million nodes (`100m`).

5.6. Testing scalability and comparing with compressed adjacency lists

In this section we analyze the scalability of our approach. However, the baselines used in the previous section are not longer valid. Uncompressed adjacency list are generally much faster, but, as the size of the input binary relations grows, they become not practical due to their memory requirements. In these scenarios, compressed adjacency lists are commonly use as they require much less memory (Yan et al., 2009; Claude et al., 2016). The drawback of these solutions is that they decrease their performance in terms of time, as accessing compressed data is slower than processing them in plain form. Moreover, depending on the strategies or encodings used to compress the adjacency lists, it may be require decompressing each list before performing the set operations, and then compressing back the result.

Thus, in this section we compare the time efficiency and memory usage of our data structures with those obtained by two different baselines using compressed adjacency lists. More specifically, we analyze space and times results when performing intersection operations, varying also the size of the input binary relations. We use as baselines those described in Section 5.2.3, that

is, qmx and rice-runs, and datasets described in Table 2.

Figure 11(a) shows the time taken for the four representations to perform the intersection. We expected the compressed lists to behave better than our approach (at the expense of being more memory demanding), but only rice-runs outperforms us independently of the input size. qxm shows timings similar to ours, being slightly faster only for the largest dataset, without being these differences statistically significant. QMX is an extremely fast technique for long adjacency lists, but this is not generally the case of our dataset, as many of the nodes of Web graphs tend to have short adjacency lists. Only when the size of the dataset grows, these lists become to be larger and QMX starts to be competitive. Rice coding combined with run-length compression does obtain good results, as the adjacency lists contain consecutive elements, which represent pages pointing to several pages of the same domain. Figure 11(b) shows the box plot for the times obtained by the different binary relation from the largest dataset (100m).

Figure 12(a) shows the size of each representation, where we can see that both implementations of compressed adjacency lists always demand more memory than k^2-trees (both variants), independently to the input size. For better appreciate these difference, we show in Figure 12(b) the box plot with the distribution of sizes for the dataset snaps-uk that contains 100 million nodes (100m). We can see that there is a significant difference of almost one order of magnitude.

Regarding scalability, we can see that our approach scales similarly to uncompressed adjacency lists. The more noticeable fact is that ktones is more competitive for smaller datasets than for larger datasets, particularly in terms of time. This is not actually due to the size of the dataset, but to the input density, as our smaller datasets contain a much higher density than the largest ones (see Table 2). As we have used real Web graphs, if we consider small portions of their binary adjacency matrix, they will contain few domains, showing a higher density than larger matrices, which contain many domains and a larger percentage of non-related page.

It must be noticed that this comparison is just a proof of concept that our approach is competitive in terms of spatio-temporal efficiency with other approaches that can be used to also represent binary matrices in compact space. However, compressed adjacency list do not offer the same extended functionality as k^2-trees. Compressed adjacency lists are fast when retrieving the whole adjacency list of an element, or to sequentially process all the elements of the relation, which is the way in which the set operations are solved. Thus, the operation performed in our experiments is the best case scenario for compressed adjacency lists. However, there are other needs, such as determining if two elements are related or not, or range queries over binary relations that are not efficiently implemented, and require the retrieval of one or several complete adjacency lists. Moreover, given an element y, obtaining all elements x such as xRy (which can be considered as obtaining the predecessors of a node, in the case of a graph), requires processing all the adjacency lists, which is very time-consuming. k^2-trees solve these kinds of queries in a very efficient way, as they were designed specifically to represent binary relations.

6. Conclusions and future work

In this work, we have extended the original functionality of the k^2-tree by proposing efficient algorithms to compute the union, intersection, difference, symmetric difference, and complement of k^2-trees. Thus, the k^2-tree supports now different set operations to operate with binary relations that are compactly represented using k^2-trees. Moreover, we have proposed these algorithms for different variants of k^2-trees, which were designed to represent both sparse and dense binary relations.

According to the bibliography review, and to the best of our knowledge, these algorithms are the first for the set operations of binary relations stored in a k^2-tree compact data structure, which is of interest to multiple domains where k^2-tree have proven to be applicable, such as Web and social graphs, raster data, RDF datasets, multidimensional data, etc.

The proposed algorithms are able to compute the result directly over the k^2-trees and generate a new k^2-tree, without requiring a previous decompression or transformation into classical representations such as their corresponding relation matrices or lists. This guarantees that the space requirements for computing the result is close to the one required by the k^2-trees involved in the operation. Furthermore, for all the algorithms shown, it can be proven that no bit is processed twice. Thus, the resulting k^2-tree is efficiently obtained in time linear to the size of the input k^2-trees.

All algorithms were implemented over the two variants of the k^2-tree and evaluated considering real datasets. Two baselines were used: `list`, a naive approach where the binary relations are directly represented and operated using a plain list of related elements, which is not feasible in practice when operating with real large datasets; and `kt-list-kt/ktones-list-ktones`, where the input binary relations represented using k^2-trees were first decompressed into lists, operated, and the result was then compressed. We analyzed the results considering four different criterion: input size, input density, output density, and input compressibility. Results show that the proposed algorithms are more than one order of magnitude faster than the `kt-list-kt/ktones-list-ktones` baseline. Moreover, some scenarios show that we obtain up to three orders of magnitude faster than this baseline, and even obtain better results than baseline `list`. In addition, all baselines require much more space, unfeasible for large datasets.

We also analyzed the scalability properties of our approach, using two different compressed adjacency list representations as baselines. Our algorithms require much less memory and are competitive in terms of time with these approaches. This comparison was included only as a proof of concept, as these representations are not totally suitable for binary relations, as they are not prepared to efficiently answer all typical queries that can be performed over binary relations.

The article focuses on the static version of k^2-trees. There exists a dynamic version (Brisaboa et al., 2017), which allows updating the binary relation or extending it, by loading more data into the data structure. This dynamic version uses an analogous strategy, but using dynamic bitmaps. Thus, the algorithms proposed in this paper are directly applicable to the dynamic version.

Future lines of research include extending the set of binary operations supported by the k^2-tree, such as restriction or composition of relations, and also algorithms that check different properties over the binary relations, such as reflexivity, symmetry, antisymmetry, or transitivity.

7. Acknowledgments

Gilberto Gutiérrez was supported by the research group ALgoritmos y BAses de Datos (ALBA) 160119 GI/EF and the research project 171319 4/R, both funded by Universidad del Bío-Bío (Chile).

Susana Ladra was supported by Ministerio de Economía y Competitividad (PGE and FEDER) [grant number TIN2016-77158-C4-3-R]; Xunta de Galicia (co-founded with European Regional Development Fund - ERDF) [grant number ED431C 2017/58]; and support from Xunta de Galicia (Centro singular de investigación de Galicia accreditation 2016-2019) and the European Union (European Regional Development Fund - ERDF).

Miguel R. Penabad was supported by Ministerio de Economía y Competitividad (PGE and FEDER) [grant number TIN2016-78011-C4-1-R], Xunta de Galicia (co-founded with European

Regional Development Fund - ERDF) [grant number ED431C 2017/58]; and support from Xunta de Galicia (Centro singular de investigación de Galicia accreditation 2016-2019) and the European Union (European Regional Development Fund - ERDF). .

Carlos Quijada-Fuentes was supported by CONICYT-PCHA/MagisterNacional/2015-22150776 and by the research group ALgoritmos y BAses de Datos (ALBA) 160119 GI/EF funded by Universidad del Bío-Bío (Chile).

References

Álvarez-García, S., de Bernardo, G., Brisaboa, N. R., Navarro, G., 2017. A succinct data structure for self-indexing ternary relations. Journal of Discrete Algorithms 43, 38–53.

Baeza-Yates, R. A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Barbay, J., Claude, F., Navarro, G., 2013. Compact binary relation representations with rich functionality. Infomation and Computation 232, 19–37.

Barbay, J., Golynski, A., Munro, J. I., Rao, S. S., 2007. Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theoretical Computer Science 387, 284–297.

Boldi, P., Rosa, M., Santini, M., Vigna, S., 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M. P., Bertino, E., Kumar, R. (Eds.), Proceedings of the 20th international conference on World Wide Web. ACM Press, pp. 587–596.

Boldi, P., Santini, M., Vigna, S., 2008. A large time-aware graph. SIGIR Forum 42 (2), 33–38.

Boldi, P., Vigna, S., 2004. The WebGraph framework I: Compression techniques. In: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004). ACM Press, Manhattan, USA, pp. 595–601.

Brisaboa, N., Cerdeira-Pena, A., de Bernardo, G., Navarro, G., 2017. Compressed representation of dynamic binary relations with applications. Information Systems 69, 106–123.

Brisaboa, N. R., Ladra, S., Navarro, G., 2013. Dacs: Bringing direct access to variable-length codes. Journal of Discrete Algorithms (In press) 49 (1), 392–404.

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

Claude, F., Fariña, A., Martínez-Prieto, M. A., Navarro, G., 2016. Universal indexes for highly repetitive document collections. Information Systems 61, 1–23.

Claude, F., Ladra, S., 2011. Practical representations for web and social graphs. In: Proc. of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011). ACM Press, pp. 1185–1190.

Culpepper, J., Moffat, A., 2010. Efficient set intersection for inverted indexing. ACM Transactions on Information Systems) 29 (1), art. 1.

de Bernardo, G., Álvarez-García, S., Brisaboa, N. R., Navarro, G., Pedreira, O., 2013. Compact querieable representations of raster data. In: Procs. of SPIRE. pp. 96–108.

González, R., Grabowski, S., Mäkinen, V., Navarro, G., 2005. Practical implementation of rank and select queries. In: Poster Procs. of WEA. pp. 27–38.

Jacobson, G., 1989. Space-efficient static trees and graphs. In: Procs. of FOCS. pp. 549–554.

Lemire, D., Boytsov, L., Kurz, N., 2016. SIMD compression and the intersection of sorted integers. Software: Practice and Experience 46(6), 723–749.

Lin, T. W., 1997. Set operations on constant bit-length linear quadtrees. Pattern Recognition 30(7), 1239–1249.

OGC, 2010. OpenGIS implementation standard for geographic information - simple feature access - part 2: SQL option. http://portal.opengeospatial.org/files/?artifact_id=25354, open Geospatial Consortium Inc.

Samet, H., 2006. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc.

Shaffer, C. A., Samet, H., 1990. Set operations for unaligned linear quadtrees. Computer Vision, Graphics, and Image Processing 50(1), 29–49.

Transier, F., Sanders, P., 2010. Engineering basic algorithms of an in-memory text search engine. ACM Transactions on Information Systems) 29 (1), art. 2.

Trotman, A., 2014. Compression, SIMD, and postings lists. In: Proc. 19th Australasian Document Computing Symposium (ADCS). pp. 50–57.

Witten, I., Moffat, A., Bell, T., 1999. Managing Gigabytes. Morgan Kaufmann, 2nd edition.

Yan, H., Ding, S., Suel, T., 2009. Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th International Conference on World Wide Web. WWW '09. ACM, New York, NY, USA, pp. 401–410.
URL http://doi.acm.org/10.1145/1526709.1526764

Zobel, J., Moffat, A., Jul. 2006. Inverted files for text search engines. ACM Comput. Surv. 38 (2).
URL http://doi.acm.org/10.1145/1132956.1132959