Efficient set operations over k^2-trees

Nieves R. Brisaboa[⋆], Guillermo de Bernardo[⋆], Gilberto Gutiérrez[*],
Susana Ladra[⋆], Miguel R. Penabad[⋆] and Brunny A. Troncoso[*]

[⋆]Universidade da Coruña
Facultade de Informática
A Coruña, 15071, Spain
brisaboa,gdebernardo,
sladra,penabad@udc.es

[*]Universidad del Bío-Bío
Facultad de Ciencias Empresariales
Concepción, Chile
ggutierr,btroncos@ubiobio.cl

Abstract

Abstract: k^2-trees have been proved successful to represent in a very compact way different kinds of binary relations, such as web graphs, RDFs or raster data. In order to be a fully functional succinct representation for these domains, the k^2-tree must support all the required operations for binary relations. In their original description, the authors include how to answer some of the most relevant queries over the k^2-tree. In this paper, we extend this functionality and detail the algorithms to efficiently compute the k^2-tree resulting from the union, intersection, difference or complement of binary relations represented using k^2-trees.

1 Introduction

Binary relations can be an abstraction to represent the relation between the objects of two collections of different nature: graphs, trees, strings, among others. They can be used in several low-level structures within a more complex information retrieval system, or even replace one of the most used one: an inverted index can be regarded as a binary relation between the vocabulary of terms and the documents where they appear. Moreover, it can represent the relation between terms and web pages, or even social networks or the connection between the web pages in the Web, which is normally represented as a web graph.

Given a binary relation R between two sets X and Y where elements $x \in X$ are related with elements $y \in Y$, which is denoted $(x, y) \in R$ or xRy, the basic operations that we want to perform over R are: given two elements $x \in X$ and $y \in Y$, determine whether xRy; given an element $x \in X$, count/list all $y \in Y|xRy$; or given an element $y \in Y$, we want to count/list all $x \in X|xRy$. In addition, other common operations over binary relations are the well-known union, intersection, difference or complement.

Binary relations can be represented by a matrix $\{r_{ij}\}$ called relation matrix. If xRy then the cell r_{xy} of the relation matrix is 1, and if these elements are not related, then the cell value is 0. This matrix representation has been exploited in several works to compactly represent binary relations while supporting operations in an efficient way.

This work was supported in part by MECESUP UBB0704 project (GG), the group "Bases de Datos" 132019 GI/EF of the Universidad del Bío-Bío (GG, BT), TIN2013-46238-C4-3-R (NB, MP), TIN2013-46801-C4-3-R (SL), and GRC2013/053 (NB, GB, SL, MP).

1068-0314/15 $31.00 © 2015 IEEE
DOI 10.1109/DCC.2015.9

IEEE computer society

One of the most complete representations for binary relations was proposed by Barbay et al. [2, 1], who study the representation of binary relations using succinct data structures, while supporting a wide set of operators efficiently. They also consider data structures for binary relations and adaptive algorithms on these data structures [2].

Brisaboa et al. [4] present a complete study of the k^2-tree, a new Web graph representation based on a compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph. A k^2-tree is able to efficiently answer if two Web pages are connected or the list of pages that point to or are pointed by a specific Web page. However, there are operations that are not supported by the original work. For instance, given two snapshots of the Web graph in two different moments of time, it would be interesting to obtain Web pages that were connected in the first snapshot but not in the second, Web pages that were connected at both snapshots, or Web pages that were connected at any temporal moment. k^2-trees have been used in other scenarios, such as geographical and RDF data, or images. The viability of the usage of k^2-tree in these scenarios depend on the set of operations and queries that it supports. Hence, operations such as the union, intersection, difference or complement should also be supported by the k^2-tree to be considered as a fully functional representation of binary relations.

In this paper, we want to extend the functionality of k^2-tree for binary relations. We want to verify whether the implementation of binary operations directly on this structure is possible and efficient. Thus, we present algorithms that perform those binary operations directly over the compressed representation and generate the result as a k^2-tree. We first review the k^2-tree structure and then propose the algorithms for different variants of the k^2-tree.

2 Previous work: k^2-tree

The k^2-tree was originally designed for compressing Web graphs [4], and has been extensively used in recent years to represent binary relations such as web graphs and social networks [4], raster data [5], and RDF datasets [3]. It represents a binary matrix by a special kind of tree inspired in quadtrees [9] and using succinct representation for trees [7].

More specifically, the k^2-tree method represents the adjacency matrix of a graph, or more generally the binary matrix of any binary relation, using a non-balanced k^2-ary tree. The method consists in subdividing the binary matrix into k rows and k columns, thus producing k^2 submatrices of size n/k by n/k. Each of these submatrices will be a child of the root node and its value will be 1 iff there is at least one 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 level 1, with the children of the root, has been built, the method proceeds recursively for each child with value 1, until we reach submatrices full of 0s, or we reach the cells of the original matrix. Fig. 1 shows an example of this subdivision (left) and the resulting k^2-ary tree (right).

The k^2-ary tree described is not really stored as illustrated. Instead, the binary matrix is represented in a compact way with just two bit arrays: T (tree) and L

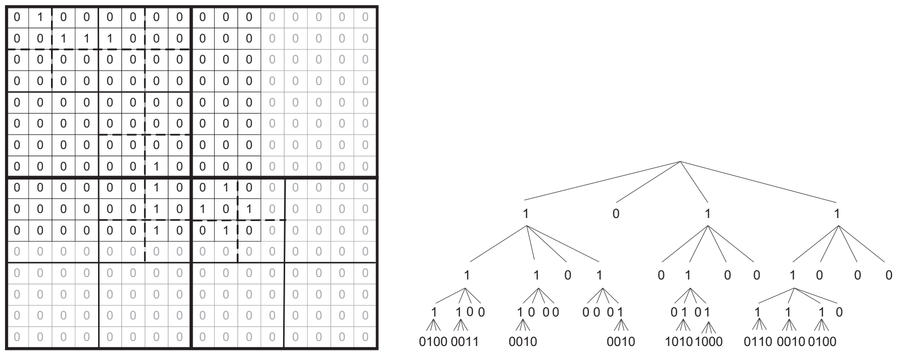
Figure 1: A k^2-tree.

(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 [6], 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.

While alternative compressed graph representations are limited to retrieving direct, and sometimes reverse, neighbors of a given node, the k^2-tree allows for these, but also for more sophisticated forms of retrieval, such as range searches or checking whether two Web pages are connected or not without extracting the complete adjacency lists of the pages involved in the query.

This extended functionality makes the k^2-tree a perfect representation for binary relations, since common operations over binary relations involve determining if an element x is related to another element y or restricting the relation to subsets of the original sets. There are more queries, properties and operations that are common for binary relations and have not been described for the k^2-trees. In the next section, we will describe efficient algorithms to perform set operations for binary relations that are represented using k^2-trees.

3 Operations for binary relations over k^2-trees

In this section we present the algorithms for computing the Union, Intersection, Difference and Complement of binary relations using k^2-trees. Thus, unlike for other queries where the answer is a boolean value or a list of elements, here we need to generate a new k^2-tree that represents the result of the operation.

All the algorithms described in this section share some common variables, which we define as follows. As input of the algorithms we consider the bitmaps representing the k^2-trees of the input binary relations. Let be the input binary relations R_A and

R_B between two sets X and Y, then we will use bitmaps A and B corresponding 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 over this bitmap will be built afterwards and are not considered in the algorithms. In addition, pA and pB are pointers that reference the next bit to read from each bitmap; l is the current level of the visited nodes from k^2-trees A and B; n is the input matrix size[1]; and $H = \lceil \log_k n \rceil$, is the height for the input k^2-trees.

For computing the union of two k^2-trees, C will be generated as just one bitmap that represents as a k^2-tree the result of $R_A \cup R_B$. However, for efficiently computing the intersection, difference and complement of k^2-trees, we use a decomposition strategy by levels: we used an array of bitmaps C with size H, where each $C[i]$ $(0 < i \leq H)$ stores the bitmap corresponding to level i of the resulting k^2-tree. Instead of two pointers pA and pB, we will use two array of pointers of size H where $pA[i]$ and $pB[i]$ reference to the next bit to read from bitmap A and B at level i. Those arrays of pointers avoid an abuse of *rank* functions to access children nodes. The final k^2-tree representation requires a concatenation of the H bitmaps $C[i]$. In addition, we will also use a temporal bitmap t with size k^2 bits, in order to store partial results.

3.1 Union algorithm

Union algorithm traverses bitmaps A and B in a synchronized, breadth-first way. Algorithm 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 A and B trees, which are at level 0, have children. Then, the algorithm processes the queue as follows until there is no tuple to process.

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 executes an OR operation among bits of their children pointed by pA and pB respectively, and the result is added to bitmap C. *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 symmetrically 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.

[1]We can 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 zeroes if necessary.

Algorithm 1 Union

```
$Union(A,B)$
$Q.Insert(\langle 0,1,1\rangle)$
$pA \leftarrow 0, pB \leftarrow 0$
while $not\ Q.Empty()$ do
    $\langle l, rA, rB\rangle \leftarrow Q.Delete()$
    for $i \leftarrow 0 \ldots k^2-1$ do
        $bA \leftarrow 0, bB \leftarrow 0$
        if $rA = 1$ then
            $bA \leftarrow A[pA]$, $pA \leftarrow pA+1$
        end if
        if $rB = 1$ then
            $bB \leftarrow B[pB]$, $pB \leftarrow pB+1$
        end if
        $C \leftarrow C||(bA \vee bB)$
        if $(l < H) \wedge (bA \vee bB = 1)$ then
            $Q.Insert(\langle l+1, bA, bB\rangle)$
        end if
    end for
end while
return $C$
```

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 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, then we can compute the value of the intersection directly using an AND operator. *Case 2:* If one of the input values from k^2-tree A and B at level i is 0, then the result is 0, and we concatenate a 0 value at $C[i]$. In this case, we need to update pointers $pA[i+1], ..., pA[H]$ or $pB[i+1], ..., pB[H]$, to omit the whole non-empty subtree. For this, we propose a function $skipNodes$, which is described at Algorithm 5. *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 j, 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.

Algorithm 2 Intersection

```
boolean $Intersection(l, pA, pB)$
$writesomething \leftarrow 0$
for $i \leftarrow 0 \ldots k^2-1$ do
    if $l < H$ then
        if $(A[pA[l]] \wedge B[pB[l]])$ then
            $t[i] \leftarrow Intersection(l+1, pA, pB)$
        else
            $t[i] \leftarrow 0$
            $skipNodes(A, pA, l+1, A[pA[l]])$
            $skipNodes(B, pB, l+1, B[pB[l]])$
        end if
    else
        $t[i] \leftarrow A[pA[l]] \wedge B[pB[l]]$
    end if
    $writesomething \leftarrow writesomething \vee t[i]$
    $pA[l] \leftarrow pA[l]+1, pB[l] \leftarrow pB[l]+1$
end for
if $writesomething = 1$ then
    $C[l] \leftarrow C[l]\ ||\ t$
end if
return $writesomething$
```

Algorithm 5 *SkipNodes* (left) and *Copy* (right) functions

```
1: $SkipNodes(X, pX, l, s)$
2: $newpos \leftarrow pX[l] + s * k^2 - 1$
3: $nOnes \leftarrow rank(X, newpos) - rank(X, pX[l] - 1)$
4: $pX[l] \leftarrow newpos + 1$
5: if $(nOnes > 0) \wedge (l < H)$ then
6:     $SkipNodes(X, pX, l + 1, nOnes)$
7: end if
```

```
1: $Copy(X, pX, l, s)$
2: $end \leftarrow pX[l] + s * k^2 - 1$
3: $nOnes \leftarrow rank(X, end) - rank(X, pX[l] - 1)$
4: $C[l] \leftarrow C[l] || X[pX[l]..end]$
5: $pX[l] \leftarrow end + 1$
6: if $(nOnes > 0) \wedge (l < H)$ then
7:     $Copy(X, pX, l + 1, nOnes)$
8: end if
```

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 from k^2-tree A is 0, then the result is 0 and this value is added to $C[i]$, where i means the associated level. 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 of A concatenating the bits level by level in $C[i]$ (see Algorithm 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 6 shows the pseudocode. The first call to the algorithm is $Difference(1, pA, pB)$, where all values of pA and pB are properly initialized.

Algorithm 6 Difference

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

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 of A (see Algorithm 7); because of this, we used the decomposition strategy by levels. The process finalizes when all of the branches of the k^2-tree A have been visited. Complement algorithm processes each node of the k^2-tree 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. *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, 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.

Algorithm 7 Complement

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

4 Set operations for k^2-trees with compression of ones (k^2-tree1)

A k^2-tree 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 zeroes by a simple 0. However, if the number of ones grows, k^2-trees 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 [5]. The basic idea is to represent uniform areas (either *black* zones of ones, or *white* zones of zeroes) by a 0, and areas with mixed ones and zeroes (*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, a 1 in a bitmap T means it is a gray area, while a 0 means it is a black or white area. T' will have a bit for each 0 in T, and its value is either 0 (white zone: all zeroes) 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. The pseudocode and explanation of the Complement, Union, Intersection and Difference operations follows. As k^2-tree1 can be regarded as a compact and efficient representation of a quadtree [9], some existing works studied this problem before for different implementations of quadtrees [10, 8]. The algorithms we propose are particular for the implementation of k^2-tree1, which use succinct data structures.

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 8 shows the pseudocode.

Algorithm 8 Complement Algorithm for k^2-tree1

```
1: $Complement(A)$
2: $C.T \leftarrow A.T$
3: $C.T' \leftarrow \sim A.T'$
4: $C.L \leftarrow \sim A.L$
5: **return** $C$
```

4.2 Union, Intersection and Difference

Intersection and 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 and difference of k^2-tree1s. In fact, traversal of the trees is so similar on all of them, that we have combined the three operations in the same algorithm (Algorithm 9). A table is included to indicate the next steps to execute that 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). However, these bitmaps (including the vector t to store partial results) contain now "3-valued bits": 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 have only 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, pB are also composed of two values, one for T and one for T'.

Algorithm 9 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 6 on Algorithm 9 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 symmetrical, 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 (children 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.

Algorithm 9 Recursive Binary Operation Algorithm (k^2-tree1)

```
int $BinOp(\odot,l)$
$writesomething \leftarrow 0$
$full \leftarrow 1$
for $i \leftarrow 0 \dots k^2-1$ do
    if $l < H-1$ then
        Execute the following step depending on values $a = A[pA[l]]$, $b = B[pB[l]]$ and the binary operation $\odot$
```

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

```
    else
        $t[i] \leftarrow A[pA[l]]\{\wedge|\vee|\wedge\sim\}B[pB[l]]$
    end if
    $writesomething \leftarrow writesomething \vee t[i]$
    $full \leftarrow full \wedge (t[i] = 2)$
    $pA[l] \leftarrow pA[l]+1, pB[l] \leftarrow pB[l]+1$
end for
if $writesomething = 1$ then
    if $full = 1$ then
        return 2
    else
        $C[l] \leftarrow C[l] \mid\mid t$
    end if
end if
return $writesomething$
```

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 leaves 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.

5 Discussion and Future Work

The k^2-tree has been extensively used in recent years to represent binary relations: web graphs, social networks, raster data, RDF datasets, etc. In addition to its data structures, several navigation algorithms have been described, such as retrieving the original binary relation, determining if one element x is related to another element y or all the elements related with a specific one. In this paper we go further in the proposal of functionality for this data structure, detailing the algorithms to compute the union, intersection, complement and difference of k^2-trees, including different variants of their representation. 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.

We plan to continue this work extending the set of binary operations supported by the k^2-tree, including other operations such as restriction or composition of relations, and also algorithms that check different properties over the binary relations, such as reflexivity, symmetry, antisymmetry, transitivity, etc. We also plan to include an extensive experimental evaluation of the practical performance of the algorithms, compared with trivial baselines, applying set operations directly over raw data, and also with other representations such as [1]. We also plan to extend this work to support operations for n-ary relations.

References

[1] J. Barbay, F. Claude, and G. Navarro, "Compact binary relation representations with rich functionality," *Infomation and Computation*, vol. 232, pp. 19–37, 2013.

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

[3] N. R. Brisaboa, G. de Bernardo, and G. Navarro, "Compressed dynamic binary relations," in *Procs. of DCC*, 2012, pp. 52–61.

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

[5] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira, "Compact querieable representations of raster data," in *Procs. of SPIRE*, 2013, pp. 96–108.

[6] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Poster Procs. of WEA*, 2005, pp. 27–38.

[7] G. Jacobson, "Space-efficient static trees and graphs," in *Procs. of FOCS*, 1989, pp. 549–554.

[8] T. W. Lin, "Set operations on constant bit-length linear quadtrees," *Pattern Recognition*, vol. 30(7), pp. 1239–1249, 1997.

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

[10] C. A. Shaffer and H. Samet, "Set operations for unaligned linear quadtrees," *Computer Vision, Graphics, and Image Processing*, vol. 50(1), pp. 29–49, 1990.