Efficient algorithms for clone items detection

Raoul Medina, Caroline Noyer and Olivier Raynaud

LIMOS - Université Blaise Pascal,
Campus universitaire des Cézeaux, Clermont-Ferrand, France
{medina, raynaud}@isima.fr

Abstract. This paper presents efficient algorithms for clone items detection in a binary relation. Best implementation of these algorithms has $\mathcal{O}(|J|.||\mathcal{M}||)$ time complexity, with J corresponding to the set of items of the relation and $||\mathcal{M}||$ corresponding to the size of the relation. This result improves the previous algorithm given in [3] which is $\mathcal{O}(|J|^2.||\mathcal{M}||)$. Clone items have been introduced by Medina and Nourine to explain why, sometimes, the number of rules of a minimum cover of a relation is exponential with the number of items of the relation.

1 Introduction

The clone items notion has been introduced by Medina and Nourine in [3]. Aim of their paper was to understand the combinatorial explosion that might arise in a minimum basis of a relation. Most famous example of such exponential minimum basis is given by Manilla and Rähiä in [2]. Medina and Nourine noticed that in this example some items play symetrical role in the basis. Indeed, for each rule containing a given item a in the antecedent there is a symetrical rule where item a is replaced by item b. Such symetrical items are said to be clone items. The clone notion is an equivalence relation and thus classes of clone items can be found in a binary relation. In [3], the authors show how to compute such clone classes and, from those classes, how to reduce the binary relation in order to obtain a minimum basis with no clone items. The detection algorithm as well as the reduction algorithm have both polynomial time complexities. The obtained minimum basis is smaller than the original one (in the case of the Mannila and Rähiä example it reduces to a single rule) and this later can be reconstructed from the clone free minimum basis and the clone classes information in polynomial time.

Recently, Gely et al. (in [1]) extended the notion of clone items (which are defined on closed sets) to the notion of P-clone items (defined on pseudo-closed sets) and to the notion of A-clone items (defined on what they call an "atomized" context). Their approach could be generalized to any generation problem where, from a given relation, one wants to compute a (potentially exponential) collection of sets verifying a property (e.g. the sets must be closed, or the sets must be ideals, or the sets are pseudo-closed sets, etc...). The idea is then to reduce the combinatorial explosion of the wanted collection by representing it by a clone free collection and the classes of clone items of this collection. The problem is

Radim Bělohlávek, Václav Snášel (Eds.): CLA 2005, pp. 70–81, ISBN 80–248–0863–3.

then to be able to compute the clone classes of the (potentially exponential) collection without generating its sets. To solve this problem in polynomial time, the general method could be as follows:

- Let $\mathcal{M}$ be the (potentially exponential) collection of sets verifying a property over a given binary relation R. Compute in polynomial time a collection $\mathcal{M}'$ such that:
  1. the size of $\mathcal{M}'$ is polynomial in the size of R,
  2. items a and b are clone in $\mathcal{M}$ if and only if they are clone items in $\mathcal{M}'$.
- Detect the clone classes in $\mathcal{M}'$.

Our paper focuses on the detection phase of this approach. The clone classes computation algorithm given in [3] has an $\mathcal{O}(|J|^2.||\mathcal{M}||)$ time complexity, where J is the set of items and $||\mathcal{M}||$ is the size of the input collection. In other words, $||\mathcal{M}|| = \sum_{m\in\mathcal{M}} |m|$. In this paper, we present different algorithms to solve the clone classes computation. The best complexity we obtain is in $\mathcal{O}(|J|.||\mathcal{M}||)$.

This paper is organized as follows: section 2 formally defines the problem in terms of collection of sets and introduces the corresponding Abstract Data Type which will be used by our algorithms. Section 3 describes three computation strategies and the corresponding time complexities are studied. Section 4 shows how to take advantage of those algorithms in order to compute the clone items classes as defined in [3].

2 General context and definitions

In this section, we first formally define the studied problem. Then we present the Abstract Data Structure called *Map* used in our algorithms and discuss on its possible implementations.

2.1 Clone items in a Sets Collection

Let J be a set of items $\{x_1, ..., x_{|J|}\}$ and $\mathcal{M}$ a sets collection on J. We denote by $\varphi_{a,b} : 2^J \to 2^J$ the mapping which associates to any subset of J its image by swapping items a and b. More formally :

$$\varphi_{a,b}(m) = \begin{cases} (m \setminus \{a\}) \cup \{b\} & if\ b \notin m\ and\ a \in m \\ (m \setminus \{b\}) \cup \{a\} & if\ a \notin m\ and\ b \in m \\ m & otherwise \end{cases}$$

Definition 1. *Let $\mathcal{M}$ be a collection of sets defined on J. We say that items a and b are clone items in $\mathcal{M}$* ***if and only if*** *$\forall m \in \mathcal{M}$, $\varphi_{a,b}(m) \in \mathcal{M}$.*

The clone items concept is a binary one. To the question "are a and b clone items ?", only the answer "true" or "false" is possible. It could be interesting to have more precisions when the negative response is given. Are a and b very far from being clone items or why are not they clone ? For this purpose, we introduce a measure to qualify the clone property. This measure will represent

a distance between two items a and b, based on the definition of the clone items property. This distance is exactly the number of elements m of $\mathcal{M}$ which do not have an image in $\mathcal{M}$ when applying the swapping function $\varphi_{a,b}(m)$. When this distance is zero, a and b are clone items. The greater the distance is, the farther a and b are to be clone. More formally:

Definition 2. *Let $\mathcal{M}$ be a sets collection on J and let (a, b) in J^2. We call distance between a and b, denoted by $d_{\mathcal{M}}(a, b)$, the mapping :*

$$\begin{aligned} J^2 &\rightarrow \mathbb{N} \\ d_{\mathcal{M}}(a,b) &\rightarrow \left|\{m \in \mathcal{M} \mid \varphi_{a,b}(m) \notin \mathcal{M}\}\right| \end{aligned}$$

Thanks to definition 2, clone items could be charaterized as follows :

Proposition 1. *Let $\mathcal{M}$ be a sets collection on J and (a, b) in J^2, a and b are clone items* ***if and only if*** *$d_{\mathcal{M}}(a, b) = 0$.*

The problem we study in this paper is the computation of *distances* between all pair of items of J:

Problem 1 (Distance).
Data : a sets collection $\mathcal{M}$ on J;
Result : the matrix $d_{\mathcal{M}}$.

Here, we present the main property on which rely our algorithms. It characterizes a couple (m, m') of the sets collection $\mathcal{M}$ such that $m = \varphi_{a,b}(m')$.

Proposition 2. *Let $\mathcal{M}$ be a sets collection defined over J, m and m' two distinct sets of $\mathcal{M}$ and (a, b) two items of J such that $a \in m$ and $b \in m'$. Then the following assertions are equivalent:*

1. $m = \varphi_{a,b}(m')$
2. $m' = \varphi_{a,b}(m)$
3. $|m| = |m'|$ *and* $m \setminus m' = \{a\}$ *and* $m' \setminus m = \{b\}$
4. $m \setminus \{a\} = m' \setminus \{b\}$

This property states that two sets m and m' are their respective images by the swapping function φ **if and only if** they have same size t and share $t - 1$ items. This property follows directly from the definition of the φ mapping.

2.2 Abstract Data Type : Key Mapping

Interface. We use a Map abstract data type similar to the Map interface of Java language. This data structure maps keys to values. In our case, the keys are the sets of the collection. The values mapped by the keys depend on the algorithm.

This abstract data type supplies the following operators:

- `new()` operator: creates a *Map* object and returns an empty map.

- `get(e)` operator: returns the value associated to the key `e` if this key maps a value, or `Nil` otherwise.
- `put(e,value)` operator: inserts set `e` in the map and associates `value` to it.

Time complexities of those operators deeply rely on the data structure used for the implementation of the Map data type. We propose an implementation which takes advantage of the type of the keys, i.e. sets. To implement the Map type we propose a lexicographic tree: itemsets are represented by branches of the tree.

Implementation: lexicographic tree. We first give a formal definition of a lexicographic tree corresponding to a sets collection.

Definition 3. *Let $\mathcal{M}$ be a sets collection defined over J, with a total order on J denoted by $<_J$. A unique lexicographic tree is associated to $\mathcal{M}$ such that:*

- *Each edge of the tree is labeled with an element of J;*
- *To each marked node of the tree corresponds a set of $\mathcal{M}$;*
- *To each set m of $\mathcal{M}$ correspond a unique path in the tree (starting from root and ending with a marked node) such that the union of labels in this path corresponds exactly to the set m.*
- *For any path from the root to any node, the order of the successive labels respects the order defined by $<_J$;*
- *The order of edges leaving a node respects the order defined by $<_J$.*

Figure 1 gives an example of collection and its associated lexicographic order.

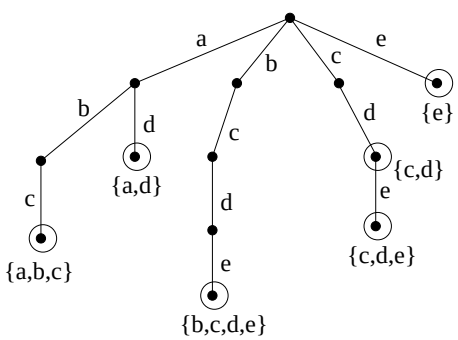

Fig. 1. The lexicographic tree corresponding to the collection $\{abc, ad, bcde, cd, cde, e\}$, with $J = \{a, b, c, d, e\}$ and $<_J$ being the alphabetical order of the items. Circled nodes are the marked nodes corresponding to sets of the collection.

A boolean can be associated to a node of the tree in order to indicate if the node is marked (i.e. corresponds to a set of the collection and thus to a key) or

not. Any field type can be associated to the node for storing the value associated to the key (on marked nodes only). There are two ways of implementing the list of children of a node in the lexicographic tree:

- Using a `List` structure, each entry of the list containing the label of the associated edge and a reference to the child;
- Using an `Array` structure being indexed by the labels and the entries containing either `NIL` or reference to the child.

Complexities of the `put` and `get` operators rely on the chosen implementation.

Proposition 3. *If the lexicographic tree is implemented with `Lists` then:*

- *The `put(m,value)` operator as an $\mathcal{O}(|J|)$ time complexity;*
- *The `get(m)` operator as an $\mathcal{O}(|J|)$ time complexity;*

Access complexity is due to the fact that an item appears only once in a set. Cost of a node creation is done in constant time.

Proposition 4. *If the lexicographic tree is implemented with `Arrays` then:*

- *The `put(m,value)` operator as an $\mathcal{O}(|m| \times |J|)$ time complexity;*
- *The `get(m)` operator as an $\mathcal{O}(|m|)$ time complexity;*

Acces complexity is due to the fact that we have direct access to a child labeled by a given item. Creation of a node of the tree costs $|J|$ since we need to create and initialize a $|J|$ sized array.

3 Strategies and algorithms

In this section, we study three different strategies for solving the *Distance* problem. All strategies rely on the same basic idea: first the distance matrix is initialized with the maximum possible values for each distance $d_{\mathcal{M}}(a, b)$. Then, each time the algorithm finds m and m' such that $m = \varphi_{a,b}(m')$, the distance $d_{\mathcal{M}}(a, b)$ is decreased by one. Main difference between the three strategies is the way they detect that $m = \varphi_{a,b}(m')$.

The first strategy is the one given in [3]. We call it *set existence checking*. Main idea of the algorithm is, given $m \in \mathcal{M}$, to compute all possible sets $\varphi_{a,b}(m)$ and then check if these sets belong to the collection. Second strategy is called $\varphi_{a,b}$ *relation checking*. Its principle is, for each pair (m, m'), to check if there exist items a and b such that $m = \varphi_{a,b}(m')$. This test is done using Property 2. Third strategy is called *classes computation*. Based on Property 2, it computes classes C of itemsets such that for any pair (m, m') of C, there exist items a and b such that $m = \varphi_{a,b}(m')$.

We first discuss about the initialization of the distance matrix since this is a common ground for all strategies.

3.1 Discussion on the distance matrix

Since the notion of distance $d_{\mathcal{M}}(a,b)$ is a symetrical one, the distance matrix is also symmetrical. We thus choose to represent it by a triangular matrix such that:

$$\begin{cases} d_{\mathcal{M}}(a,a) = 0 \\ d_{\mathcal{M}}(a,b) = d_{\mathcal{M}}(b,a) \end{cases}$$

In this section we discuss the different strategies for the initialization and the update of the distance matrix, as well as their respective costs.

Let $\mathcal{M} = \mathcal{M}_{a\overline{b}} + \mathcal{M}_{\overline{a}b} + \mathcal{M}_{ab} + \mathcal{M}_{\overline{a}\overline{b}}$, with:

- $\mathcal{M}_{a\overline{b}}$: the sets m of $\mathcal{M}$ such that $a \in m$ and $b \notin m$
- $\mathcal{M}_{\overline{a}b}$: the sets m of $\mathcal{M}$ such that $a \notin m$ and $b \in m$
- $\mathcal{M}_{ab}$: the sets m of $\mathcal{M}$ such that $a \in m$ and $b \in m$
- $\mathcal{M}_{\overline{a}\overline{b}}$: the sets m of $\mathcal{M}$ such that $a \notin m$ and $b \notin m$.

Incrementation or decrementation strategy ? There are two ways of computing the distance matrix:

- Either by first initializing all distances to 0 and then by increasing by 1 the distance $d_{\mathcal{M}}(a,b)$ each time we find $m \in \mathcal{M}$ such that $\varphi_{a,b}(m) \notin \mathcal{M}$,
- or by first initializing the distances by a maximal value and then decreasing by 1 the distance $d_{\mathcal{M}}(a,b)$ each time we find m and $m' \in \mathcal{M}$ such that $m = \varphi_{a,b}(m')$.

What is the maximal value that can be taken by $d_{\mathcal{M}}(a,b)$? Clearly the answer is $|\mathcal{M}_{a\overline{b}}| + |\mathcal{M}_{\overline{a}b}|$. Indeed, for any item $m \in \mathcal{M}_{ab} \cup \mathcal{M}_{\overline{a}\overline{b}}$ we have $m = \varphi_{a,b}(m)$ and thus m cannot increase the distance between a and b. This represents the maximum number of basic operations (either incrementation or decrementation) needed to compute the distance matrix in the worst case. Thus, whatever strategy we choose, the number of basic operations will be the same in the worst case.

What is the best time complexity for both strategies ? We consider that the basic operations can be done in $\mathcal{O}(1)$. Thus the overall complexity will be:

$$\mathcal{O}(\sum_{a\in J}\sum_{b\in J} |\mathcal{M}_{a\overline{b}}| + |\mathcal{M}_{\overline{a}b}|).$$

Now, consider $m \in \mathcal{M}$. In the worst case, m will be taken into account in at most $|m| \times |J \setminus m|$ distances $d_{\mathcal{M}}(a,b)$. Indeed, for m to be taken into account in $d_{\mathcal{M}}(a,b)$ either a or b belongs to m but not both. Thus, we have:

$$\mathcal{O}(\sum_{a\in J}\sum_{b\in J} |\mathcal{M}_{a\overline{b}}| + |\mathcal{M}_{\overline{a}b}|) = \mathcal{O}(\sum_{m\in\mathcal{M}} |m| \times |J \setminus m|).$$

This can be rewritten as follows:

$$\mathcal{O}(\sum_{m\in\mathcal{M}} |m| \times (|J| - |m|)) = \mathcal{O}((\sum_{m\in\mathcal{M}} |m| \times |J|) - (\sum_{m\in\mathcal{M}} |m| \times |m|)).$$

And since $|m| \times |m|$ is lesser or equal than $|m| \times |J|$, we obtain the worst case complexity:

$$\mathcal{O}(\sum_{m \in \mathcal{M}} |m| \times |J|) = \mathcal{O}(|J| \times ||\mathcal{M}||).$$

Thus, whatever update strategy is chosen, time complexity cannot be less than $\mathcal{O}(|J| \times ||\mathcal{M}||)$ (upon the hypothesis that the basic operations increment or decrement by 1).

In this paper we choose to adopt the decrementation strategy since our algorithms are based on Property 2. We now discuss the complexity of the initialization of the distance matrix.

Initializing the distance matrix. We initialize each distance $d_{\mathcal{M}}(a, b)$ with the maximal value possible, i.e. with $|\mathcal{M}_{a\overline{b}}| + |\mathcal{M}_{\overline{a}b}|$. Initially, the distances are equal to 0. This can be done in $\mathcal{O}(|J| \times |J|)$.

Then, for each $m \in \mathcal{M}$ we increment by 1 the distances $d_{\mathcal{M}}(a, b)$, with $a \in m$ and $b \notin m$. As shown in previous subsection, this can be done in $\mathcal{O}(|J| \times ||\mathcal{M}||)$.

Algorithm 1: $InitDistance(\mathcal{M})$

```
Data   : A sets collection M defined over J.
Result : The distance matrix d_M such that for all a and b in J^2 we have
         d_M(a,b) = |M_{a b̄}| + |M_{ā b}|.
begin
    foreach a ∈ J do
        foreach b ∈ J do
            d_M(a,b) = 0;
    foreach m ∈ M do
        foreach a ∈ m do
            foreach b ∉ m do
                d_M(a,b)++;
    return d_M;
end
```

3.2 Set Existence Checking Strategy

For each set m of $\mathcal{M}$ and for all pair (a, b) of J^2, we check if $\varphi_{a,b}(m)$ belongs to $\mathcal{M}$. In order to check the existence of $\varphi_{a,b}(m)$, we choose to store the collection $\mathcal{M}$ in the Map structure presented before. The sets of $\mathcal{M}$ are the keys while their associated value is `"present"`.

Beware that since a distance $d_{\mathcal{M}}(a, b)$ is initialized with $|\mathcal{M}_{a\overline{b}}| + |\mathcal{M}_{\overline{a}b}|$, this distance should be decremented only when $m \neq \varphi_{a,b}(m)$. Indeed, if $m = \varphi_{a,b}(m)$ then $m \notin \mathcal{M}_{a\overline{b}} \cup \mathcal{M}_{\overline{a}b}$.

Algorithm 2: *ComputeDistance*($\mathcal{M}$) : Set Existence Checking Strategy

```
   Data   : A sets collection M defined over J.
   Result : The distance matrix d_M.
   begin
       d_M = InitDistance(M);
       T = new Map();
       foreach m ∈ M do
           T.put(m,"present");
1      foreach m ∈ M do
2          foreach couple(a, b) ∈ J^2 do
               if T.get(m) ≠ NIL and m ≠ φ_{a,b}(m) then
                   d_M(a, b) − −;
       return d_M;
   end
```

Proposition 5. *The Set Existence Checking Strategy has an* $\mathcal{O}(|J|^2 \times ||\mathcal{M}||)$ *time complexity.*
Proof. Initialization of the matrix is done in $\mathcal{O}(|J| \times ||\mathcal{M}||)$*. We suppose that the* $\mathcal{T}$ *Map structure is implemented using* `Arrays`*. Thus, the initialization of* $\mathcal{T}$ *is done in* $\mathcal{O}(\sum_{m\in\mathcal{M}} |m| \times |J|) = \mathcal{O}(|J| \times ||\mathcal{M}||)$*. Loop on line* 1 *does* $|\mathcal{M}|$ *iterations while loop of line* 2 *does* $|J|^2$ *iterations. The* $\varphi_{a,b}(m)$ *operation as well as the test* $m \neq \varphi_{a,b}(m)$ *and the* `get(m)` *operation can be done in* $\mathcal{O}(|m|)$ *time complexity. Overall complexity is thus,* $\mathcal{O}(\sum_{m\in\mathcal{M}} |J|^2 \times |m|) = \mathcal{O}(|J|^2 \times ||\mathcal{M}||)$*.* □

This strategy is a slighter improvement of the one presented in [3]. Note that it can be improved a little more by choosing a in m and b in $J \setminus m$.

3.3 $\varphi_{a,b}$ Relation Checking Strategy

For any pair of elements (m, m') of $\mathcal{M}$, we check if there exist a and b such that $m = \varphi_{a,b}(m')$. According to Property 2, items a and b exist **if and only if** m and m' have same size and share $|m| - 1$ items. In this case, $d_{\mathcal{M}}(a, b)$ is decremented by 1.

Proposition 6. *The* $\varphi_{a,b}$ *relation checking strategy has an* $\mathcal{O}((|J| + |\mathcal{M}|) \times ||\mathcal{M}||)$ *time complexity.*
Proof. Initialisation of the matrix is done in $\mathcal{O}(|J| \times ||\mathcal{M}||)$*. External loop (line* 1*) does* $|\mathcal{M}|^2$ *iterations. All tests of line* 2 *can be done in* $\mathcal{O}(|m|)$ *provided that the sets* m *and* m' *are stored sorted according to* $<_J$*. The complexity of the loop is in* $\mathcal{O}(|\mathcal{M}| \times ||\mathcal{M}||)$*. Overall complexity is thus in* $\mathcal{O}((|J| + |\mathcal{M}|) \times ||\mathcal{M}||)$*.* □

3.4 Classes Computation Strategy

This strategy also relies on Property 2. Let consider m, m' and m'' be sets of $\mathcal{M}$ such that $m = \varphi_{a,b}(m')$ and $m = \varphi_{a,c}(m'')$. Then, according to Property

Algorithm 3: *ComputeDistance*($\mathcal{M}$): $\varphi_{a,b}$ Relation Checking Strategy

```
   Data   : A set collections M defined over J.
   Result : The distance matrix d_M.
   begin
       d_M = InitDistance(M);
1      foreach (m, m') ∈ M^2 do
2          if |m| = |m'| and |m \ m'| = 1 and |m' \ m| = 1 then
               d_M(m \ m', m' \ m) − −;
       return d_M;
   end
```

2 we have $m \setminus \{a\} = m' \setminus \{b\} = m'' \setminus \{c\}$. And thus, $m' = \varphi_{b,c}(m'')$. Idea of the algorithm is to compute classes of sets m_i of $\mathcal{M}$ having $|m_i| - 1$ common items. Thus, a class C can be represented by the set of common items and we memorize in a set `Union` all the extra items x_i which are not common. In the Map structure we use, the set C will be the key while the set `Union` will be the value associated to the key.

Then, for any $m \in C$ and for any $(a, b) \in$ `Union`, we know that according to Property 2 we have $\varphi_{a,b}(m) \in \mathcal{M}$ and $m \neq \varphi_{a,b}(m)$. And thus, $d_{\mathcal{M}}(a, b)$ has to be decremented. Note that a set m can belong to at most $|m|$ classes.

The algorithm is quite straightforward using the Map structure. For all sets m of $\mathcal{M}$ we insert each of its $|m|$ subsets of size $|m| - 1$ in the Map structure. If the key was already present, we just append the extra item of m to the set `Union` and update all the necessary entries in the distance matrix. Otherwise, a new key $m \setminus \{x\}$ is present in the Map structure and its associated `Union` value is initialized with $\{x\}$.

Proposition 7. *Classes computation strategy has* $\mathcal{O}(|J| \times ||\mathcal{M}||)$ *time complexity.*

Proof. Initialization of the distance matrix is done in $\mathcal{O}(|J| \times ||\mathcal{M}||)$. *We suppose that the Map structure is implemented using a lexicographic tree with* `Lists`. *The line* 1 *loop does* $|\mathcal{M}|$ *iterations. Loop in line* 2 *does* $|m|$ *iterations. In line* 3*, the retrieval is done in* $\mathcal{O}(|J|)$. *The loop in line* 4 *does at most* $|J|$ *iterations and the update of the matrix takes constant time. The insertion of line* 5 *is done in* $\mathcal{O}(|J|)$. *Thus, the overall complexity is in* $\mathcal{O}(\sum_{m \in \mathcal{M}} |m| \times |J|) = \mathcal{O}(|J| \times ||\mathcal{M}||)$. □

Let us illustrate Algorithm 4 with an example. The considered collection is $\mathcal{M} = \{m_1 = \{a, e, f, h\}, m_2 = \{b, e, f, h\}, m_3 = \{b, d, f, h\}\}$. The resulting lexicographic tree obtained with Algorithm 4 is shown in Figure 2.

From this tree, we conclude that m_2 and m_3 share the items $\{b, f, h\}$ and that $m_2 = \varphi_{d,e}(m_3)$. Thus, $d_{\mathcal{M}}(d, e)$ should be decremented. For the same reason, the distance $d_{\mathcal{M}}(a, b)$ should also be decremented since $m_1 = \varphi_{a,b}(m_2)$ (they share the items $\{e, f, h\}$).

Algorithm 4: *ComputeDistance*($\mathcal{M}$): Classes Computation Strategy

```
   Data  : A set collections M defined over J.
   Result : The distance matrix d_M.
   begin
       d_M = InitDistance(M);
       T = new Map();
1      foreach m ∈ M do
2          foreach x ∈ m do
               C = m \ {x}
3              Union = T.get(C)
               if Union ≠ Nil then
4                  foreach y ∈Union do
                       d_M(x, y) − −;
                   Union = Union ∪{x};
                   T.put(C, Union);
               else
5                  T.put(C, {x})
       return d_M;
   end
```

3.5 Memory usage

In this section we discuss the space complexity required by each strategy and show how to reduce the memory usage when possible.

First, it is obvious that the distance matrix should be present in memory. It requires $\mathcal{O}(|J|^2)$ memory space.

Concerning the Set Existence Checking Strategy, a lexicographic tree implemented with arrays is used. The number of nodes is clearly bounded by $||\mathcal{M}||$. And since each node requires a $|J|$ sized array, this strategy uses $\mathcal{O}(|J| \times ||\mathcal{M}||)$ memory space.

For the $\varphi_{a,b}$ Relation Checking Strategy, no lexicographic tree is used. However, the collection $\mathcal{M}$ needs to be present in memory. Thus, this strategy requires $\mathcal{O}(||\mathcal{M}||)$ memory space.

Now, for the Classes Computation Strategy, the used lexicographic tree is implemented using lists. Such implementation requires $\mathcal{O}(||\mathcal{M}||)$ memory space. But $\mathcal{M}$ is not the collection stored in the tree. Indeed, for each $m \in \mathcal{M}$, we store its $|m|$ subsets of size $|m|-1$. And since $|m|$ is bounded by $|J|$ we conclude that this strategy requires $\mathcal{O}(|J| \times ||\mathcal{M}||)$ memory space.

The conclusion seems to be that whatever strategy is used, at least $\mathcal{O}(||\mathcal{M}||)$ memory space will be needed. But one can notice that, according to Property 2, not all sets in $\mathcal{M}$ need to be present in memory **at the same time**. Indeed, if $m = \varphi_{a,b}(m')$, then m and m' have same size. The idea is then to do a partitionning of $\mathcal{M}$ according to the size of the sets. Thus, only sets of same size need to be present in memory at the same time. Computations done for

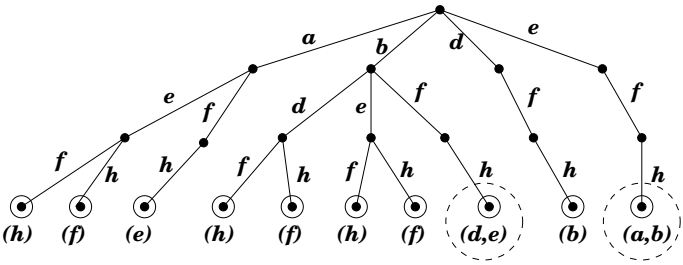

Fig. 2. The lexicographic tree corresponding to the collection $\mathcal{M} = \{\{a, e, f, h\}, \{b, e, f, h\}, \{b, d, f, h\}\}$. Circled nodes correspond to sets of $\mathcal{M}$. The `Union` value associated to these node is presented in parenthesis.

those sets is totally independent of computations done for sets with a different size. Note that the partitionning of $\mathcal{M}$ can be done in $\mathcal{O}(||\mathcal{M}||)$ time complexity, with a single scan of the collection. Thus, the partitionning does not change the overall complexity of the different strategies.

Another remark is that since computations by size are totally independent, one could easily implement a distributed version of the strategies. This could eventually speed up the computation of the distance matrix. But note that in this case, the distance matrix should also be distributed. Best solution should be to initialize local distance matrices with the local partition. After local computation (for sets with same size), all the distance matrices should be merged together (by adding all the obtained values) in order to obtain the global distance matrix.

4 Clone classes computation

The problem stated in [1] and in [3] is the computation of clone classes. We thus give the algorithm which computes clone classes from a distance matrix.

First, let us recall that two items a and b are clone **if and only if** $d_{\mathcal{M}}(a, b) = 0$. Since, the clone relation is an equivalence relation, it defines a partition of the set J.

Principle of our algorithm 5 is the following. Let $a \in J$ be an item which has still not been assigned to a class. We then search all remaining items b which distance with a is null. All those items will form a clone class with a and thus are removed from the list of items which are not assigned to a class. The class of a is then stored in a list $\mathcal{L}$. Total complexity of the clone classes computation is in $\mathcal{O}(|J| \times ||\mathcal{M}||)$ time and space complexity.

5 Conclusion

We have seen that if the problem is the computation of the distance matrix and only basic incrementation or decrementation by 1 are allowed, then the minimal

Algorithm 5: *ComputeCloneClasses*($\mathcal{M}$)

```
    Data   : A sets collection M defined over J.
    Result : The list L of clone classes.
    begin
        d_M = ComputeDistance(M);
        L = ∅; temp = J;
        while temp ≠ ∅ do
1           foreach a ∈ temp do
                l_a = newList(); l_a = l_a ∪ {a}; temp = temp \ {a};
2               foreach b ∈ temp do
3                   if d_M(a, b) = 0 then
4                       l_a = l_a ∪ {b};
5                       temp = temp \ {b};
                L = L + l_a;
        return L;
    end
```

time complexity for the update of the matrix is in $\mathcal{O}(|J| \times ||\mathcal{M}||)$. Thus, under this hypothesis, our algorithm is optimal. Figure 3 gives the different time and space complexities obtained with our algorithms.

Algorithm	Time complexity	Space complexity
Set Existence Checking	$\|J\|^2 \times \|\|\mathcal{M}\|\|$	$\|J\| \times \|\|\mathcal{M}\|\|$
$\varphi_{a,b}$ Relation Checking	$(\|J\| + \|\mathcal{M}\|) \times \|\|\mathcal{M}\|\|$	$\|\|\mathcal{M}\|\|$
Classes Computation	$\|J\| \times \|\|\mathcal{M}\|\|$	$\|J\| \times \|\|\mathcal{M}\|\|$
Optimal matrix computation	$\|J\| \times \|\|\mathcal{M}\|\|$	?

Fig. 3. Complexities of algorithms presented in the paper.

An open question is to know whether or not the distance matrix could be incremented (or decremented) by more than 1 at each step. This could be the only way of improving our algorithm. Another open question is to know if there is a more efficient algorithm to compute the clone classes (for instance without computing the distance matrix). Those are two questions we are investigating.

References

1. A. Gely, R. Medina, L. Nourine, and Y. Renaud. Uncovering and reducing hidden combinatorics in guigues-duquenne covers. In *ICFCA'05*, 2005.
2. H. Manilla and K.J. Rähiä. On the complexity of inferring functionnal dependencies. *Discret Applied Mathematics*, 40(2):237–243, 1992.
3. R. Medina and L. Nourine. Clone items: a pre-processing information for knowledge discovery. *submitted*.