Fast Discovery of Minimal Sets of Attributes Functionally Determining a Decision Attribute*

Marzena Kryszkiewicz and Piotr Lasek

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
{mkr,p.lasek}@ii.pw.edu.pl

Abstract. In our paper, we offer an efficient *Fun* algorithm for discovering minimal sets of conditional attributes functionally determining a given dependent attribute, and in particular, for discovering Rough Sets certain, generalized decision, and membership distribution reducts. *Fun* can operate either on partitions or alternatively on stripped partitions that do not store singleton groups. It is capable of using functional dependencies occurring among conditional attributes for pruning candidate dependencies. The experimental results show that all variants of *Fun* have similar performance. They also prove that *Fun* is much faster than the Rosetta toolkit's algorithms computing all reducts and faster than *TANE*, which is one of the most efficient algorithms computing all minimal functional dependencies.

1 Introduction

The determination of minimal functional dependencies is a standard task in the area of relational databases. TANE [5] or Dep-Miner [11] are example efficient algorithms for discovering minimal functional dependencies from relational databases. A variant of the task, which consists in discovering minimal sets of conditional attributes that functionally or approximately determine a given decision attribute, is one of the topics of Artificial Intelligence and Data Mining. Such sets of conditional attributes can be used, for instance, for building classifiers. In the terms of Rough Sets, such minimal conditional attributes are called reducts [13]. One can distinguish a number of types of reducts. Generalized decision reducts (or equivalently, possible/approximate reducts [7]), membership distribution reducts (or equivalently, membership reducts [7]), and certain decision reducts belong to most popular Rough Sets reducts. In general, these types of reducts do not determine the decision attribute functionally. However, it was shown in [8] that these types of reducts are minimal sets of conditional attributes functionally determining appropriate modifications of the decision attribute. Thus, the task of searching such reducts is equivalent to looking for minimal sets of attributes functionally determining a given attribute. In this

* Research has been supported by grant No 3 T11C 002 29 received from Polish Ministry of Education and Science.

paper, we focus on finding all such minimal sets of attributes. To this end, one might consider applying either methods for discovering Rough Sets reducts, or discovering all minimal functional dependencies and then selecting such that determine a requested attribute.

A number of methods for discovering reducts have already been proposed in the literature. e.g. [3-4],[6],[9-10],[12-20]. The most popular methods are based on discernibility matrices [15]. Unfortunately, the existing methods for discovering all reducts are not scalable. The recently offered algorithms for finding all minimal functional dependencies are definitely faster. In this paper, we focus on direct discovery of all minimal functional dependencies with a given dependent attribute, and expect this process to be faster than the discovery of all minimal functional dependencies. Here, we offer an efficient *Fun* algorithm for discovering minimal functional dependencies with a given dependent attribute, and, in particular, for discovering three above mentioned types of reducts. *Fun* can operate either on partitions or alternatively on stripped partitions that do not store singleton groups. It is capable of using functional dependencies occurring among conditional attributes, which are found as a sideeffect, for pruning candidate dependencies.

The layout of the paper is as follows: Basic notions of information systems, functional dependencies, decision tables and reducts are recalled in Section 2. In Section 3, we offer the *Fun* algorithm. The experimental results are reported in Section 4. We conclude our results in Section 5.

2 Basic Notions

2.1 Information Systems

An *information system* is a pair $S = (O, AT)$, where O is a non-empty finite set of *objects* and AT is a non-empty finite set of *attributes* of these objects. In the sequel, $a(x)$, $a \in AT$ and $x \in O$, denotes the value of attribute a for object x, and V_a denotes the *domain* of a. Each subset of attributes $A \subseteq AT$ determines a binary A-*indiscernibility* relation $IND(A)$ consisting of pairs of objects indiscernible wrt. attributes A; that is, $IND(A) = \{(x, y) \in O{\times}O | \forall_{a \in A}\, a(x) = a(y)\}$. $IND(A)$ is an equivalence relation and determines a partition of O, which is denoted by π_A. The set of objects indiscernible with an object x with respect to A in S is denoted by $I_A(x)$ and is called A-*indiscernibility class*; that is, $I_A(x) = \{y \in O | (x, y) \in IND(A)\}$. Clearly, $\pi_A = \{I_A(x) | x \in O\}$.

2.2 Functional Dependencies

Functional dependencies are of high importance in designing relational databases. We recall this notion after [2]. Let $S = (O, AT)$ and $A, B \subseteq AT$. $A \rightarrow B$ is defined a *functional dependency* (or A is defined to *determine* B *functionally*), if $\forall_{x \in O}\ I_A(x) \subseteq I_B(x)$. A functional dependency $A \rightarrow B$ is called *minimal*, if $\forall_{C \in A}\ C \rightarrow B$ is not functional.

Table 1. Sample DT extended with d^{N}_{AT}, ∂_{AT}, μ^{AT}_{d}

oid	a	b	c	e	f	d	d^{N}_{AT}	∂_{AT}	$\mu^{AT}_{d} :< \mu^{AT}_{1}, \mu^{AT}_{2}, \mu^{AT}_{3} >$
1	1	0	0	1	1	1	1	$\{1\}$	$< 1, 0, 0 >$
2	1	1	1	1	2	1	1	$\{1\}$	$< 1, 0, 0 >$
3	0	1	1	0	3	1	N	$\{1, 2\}$	$< 1/2, 1/2, 0 >$
4	0	1	1	0	3	2	N	$\{1, 2\}$	$< 1/2, 1/2, 0 >$
5	0	1	1	2	2	2	2	$\{2\}$	$< 0, 1, 0 >$
6	1	1	0	2	2	2	N	$\{2, 3\}$	$< 0, 1/3, 2/3 >$
7	1	1	0	2	2	3	N	$\{2, 3\}$	$< 0, 1/3, 2/3 >$
8	1	1	0	2	2	3	N	$\{2, 3\}$	$< 0, 1/3, 2/3 >$
9	1	1	0	3	2	3	3	$\{3\}$	$< 0, 0, 1 >$
10	1	0	0	3	2	3	3	$\{3\}$	$< 0, 0, 1 >$

Example 2.2.1. Let us consider the information system in Table 1. $\{ce\} \rightarrow \{a\}$ is a functional dependency, nevertheless, $\{c\} \rightarrow \{a\}$, $\{e\} \rightarrow \{a\}$, and $\emptyset \rightarrow \{a\}$ are not. Hence, $\{ce\} \rightarrow \{a\}$ is a minimal functional dependency. □

Property 2.2.1. Let $A, B, C \subseteq AT$.

a) If $A \rightarrow B$ is a functional dependency, then $\forall_{C \supset A}\ C \rightarrow B$ is functional.
b) If $A \rightarrow B$ is not a functional dependency, then $\forall_{C \subset A}\ C \rightarrow B$ is not a functional dependency.
c) If $A \rightarrow B$ is a functional dependency, then $\forall_{C \supset A}\ C \rightarrow B$ is not a minimal functional dependency.
d) If $A \rightarrow B$ and $B \rightarrow C$ are functional dependencies, then $A \rightarrow C$ is functional.
e) If $A \subset B$, $A \rightarrow B$ is a functional dependency, and $B \cap C = \emptyset$, then $B \rightarrow C$ is not a minimal functional dependency.

Functional dependencies can be calculated by means of partitions [5] as follows:

Property 2.2.2. Let $A, B \subseteq AT$. $A \rightarrow B$ is a functional dependency iff $\pi_A = \pi_{AB}$ iff $|\pi_A| = |\pi_{AB}|$.

Example 2.2.2. Let us consider the information system in Table 1. We observe that $\pi_{\{ce\}} = \pi_{\{cea\}} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6, 7, 8\}, \{9, 10\}\}$. The equality of $\pi_{\{ce\}}$ and $\pi_{\{cea\}}$ (or their cardinalities) is sufficient to conclude that $\{ce\} \rightarrow \{a\}$ is a functional dependency. □

The next property recalls a method of calculating a partition with respect to an attribute set C by intersecting partitions with respect to subsets of C. Let $A, B \subseteq AT$. The *product of partitions* π_A and π_B, denoted by $\pi_A \cap \pi_B$, is defined as $\pi_A \cap \pi_B = \{Y \cap Z | Y \in \pi_A \text{ and } Z \in \pi_B\}$.

Property 2.2.3. Let $A, B, C \subseteq AT$ and $C = A \cup B$. Then, $\pi_C = \pi_A \cap \pi_B$.

2.3 Decision Tables, Reducts and Functional Dependencies

A *decision table* is an information system $DT = (O, AT \cup \{d\})$, where $d \notin AT$ is a distinguished attribute called the *decision*, and the elements of AT are called *conditions*. A *decision class* is defined as the set of all objects with the same decision value. By X_{d_i} we will denote the decision class consisting of objects the decision value of which equals d_i, where $d_i \in V_d$. Clearly, for any object x in O,

$I_d(x)$ is a decision class. It is often of interest to find minimal subsets of AT (or *strict reducts*) that functionally determine d. It may happen, nevertheless, that such minimal sets of conditional attributes do not exist.

Example 2.3.1. Table 1 describes a sample decision table $DT = (O, AT \cup \{d\})$, where $AT = \{a, b, c, e, f\}$. Partition $\pi_{AT} = \{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6, 7, 8\}, \{9\}, \{10\}\}$ contains all AT-indiscernibility classes, whereas $\pi_{\{d\}} = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9, 10\}\}$ contains all decision classes. There is no functional dependency between AT and d, since there is no decision class in $\pi_{\{d\}}$ containing AT-indiscernibility class $\{3, 4\}$ (or $\{6, 7, 8\}$). As $AT \to d$ is not functional, then $C \to d$, where $C \subseteq AT$, is not functional either. □

Rough Sets theory deals with the problem of non-existence of strict reducts by means of other types of reducts, which always exist, irrespectively if $AT \to d$ is a functional dependency, or not. We will now recall such three types of reducts, namely certain decision reducts, generalized decision reducts, and membership distribution reducts.

Certain decision reducts. Certain decision reducts are defined based on the notion of a *positive region* of DT, thus we start with introducing this notion. A *positive region* of DT, denoted as POS, is the set-theoretical union of all AT-indiscernibility classes, each of which is contained in a decision class of DT; that is, $POS = \bigcup\{X \in \pi_{AT} | X \subseteq Y, Y \in \pi_d\} = \{x \in O | I_{AT}(x) \subseteq I_d(x)\}$. A set of attributes $A \subseteq AT$ is called a *certain decision reduct* of DT, if A is a minimal set, such that $\forall_{x \in POS}\ I_A(x) \subseteq I_d(x)$ [13]. Now, we will introduce a *derivable decision attribute* for an object $x \in O$ as a modification of the decision attribute d, which we will denote by $d^{\mathrm{N}}_{AT}(x)$ and define as follows: $d^{\mathrm{N}}_{AT}(x) = d(x)$ if $x \in POS$, and $d^{\mathrm{N}}_{AT}(x) = \mathrm{N}$, otherwise (see Table 1 for illustration). Clearly, all objects with values of d^{N}_{AT} that are different from N belong to POS.

Property 2.3.1 [8]. Let $A \subseteq AT$. A is a certain decision reduct iff $A \to \{d^{\mathrm{N}}_{AT}\}$ is a minimal functional dependency.

Generalized decision reducts. Generalized decision reducts are defined based on a *generalized decision*. Let us thus start with introducing this notion. An A-*generalized decision* for object x in DT (denoted by $\partial_A(x)$), $A \subseteq AT$, is defined as the set of all decision values of all objects indiscernible with x wrt. A; i.e., $\partial_A(x) = \{d(y) | y \in I_A(x)\}$ [15]. For $A = AT$, an A-*generalized decision* is also called a *generalized decision* (see Table 1 for illustration). $A \subseteq AT$ is defined a *generalized decision reduct* of DT, if A is a minimal set such that $\forall_{x \in O}\ \partial_A(x) = \partial_{AT}(x)$.

Property 2.3.2 [8]. Let $A \subseteq AT$. Attribute set A is a generalized decision reduct iff $A \to \{\partial_{AT}\}$ is a minimal functional dependency.

μ-Decision Reducts. The generalized decision informs on decision classes to which an object may belong, but does not inform on the degree of the membership to these classes, which could be also of interest. A *membership distribution function*) $\mu^A_d : O \to [0, 1]^n, A \subseteq AT, n = |V_d|$, is defined as follows [7],[16-17]:

$$\mu^A_d(x) = (\mu^A_{d_1}(x), \ldots, \mu^A_{d_n}(x)), \text{ where}$$

$$\{d_1, \ldots, d_n\} = V_d \text{ and } \mu^A_{d_i}(x) = \frac{|I_A(x) \cap X_{d_i}|}{|I_A(x)|}.$$

Please, see Table 1 for illustration of μ_d^{AT}. $A \subseteq AT$ is a called a *μ-decision reduct* (or *membership distribution reduct*) of DT, if A is a minimal set such that $\forall_{x \in O}\ \mu_d^A(x) = \mu_d^{AT}(x)$.

Property 2.3.3 [8]. Let $A \subseteq AT$. A is a μ-decision reduct iff $A \rightarrow \{\mu_d^{AT}\}$ is a minimal functional dependency.

3 Computing Minimal Sets of Attributes Functionally Determining Given Dependent Attribute with Fun

In this section, we offer the *Fun* algorithm for computing all minimal subsets of conditional attributes AT that functionally determine a given dependent attribute ∂. Clearly, *Fun* shall return certain decision reducts for $\partial = \partial_{AT}$, generalized decision for $\partial = d^{\mathrm{N}}_{AT}$, and μ-decision reducts for $\partial = \mu_d^{AT}$. For brevity, a minimal subset of AT that functionally determines a given dependent attribute ∂ will be called a *∂-reduct*.

3.1 Main Algorithm

The *Fun* algorithm takes two arguments: a set of conditional attributes AT and a functionally dependent attribute ∂. As a result, it returns all ∂-reducts. *Fun* starts with creating singleton candidates $\mathcal{C}_1$ for ∂-reducts from each attribute in AT. Then, the partitions (π) and their cardinalities (*groupNo*) wrt. ∂ and all attributes in $\mathcal{C}_1$ are determined.

Notation for *Fun*	
• $\mathcal{C}_k$	candidate k attribute sets (potential ∂-reducts);
• $\mathcal{R}_k$	k attribute ∂-reducts;
• $C.\pi$	the representation of the partition π_C of the candidate attribute set C; it is stored as the list of groups of objects identifiers (*oids*);
• $C.groupNo$	the number of groups in the partion of the candidate attribute set C; that is, $\|\pi_C\|$;
• $\partial.T$	an array representation of π_∂;

```
Algorithm $Fun$(attribute set $AT$, dependent attribute $\partial$);
$\mathcal{C}_1 = \{\{a\}|a \in AT\}$;           // create singleton candidates from conditional attributes in $AT$
forall $C$ in $\mathcal{C}_1 \cup \{\partial\}$ do begin
  $C.\pi = \pi_C$;
  $C.groupNo = |\pi_C|$
endfor;
/* calculate an array representation of $\pi_\partial$ for later multiple use in the $Holds$ function */
$\partial.T = PartitionArrayRepresentation(\partial)$;
for ($k = 1$; $\mathcal{C}_k \neq \emptyset$; $k++$) do begin                                    // Main loop
  $\mathcal{R}_k = \{\}$;
  forall candidates $C \in \mathcal{C}_k$ do begin
    if $Holds(C \rightarrow \{\partial\})$ then                  // Is $C \rightarrow \{\partial\}$ a functional dependency?
      remove $C$ from $\mathcal{C}_k$ to $\mathcal{R}_k$;                   // store $C$ as a $k$ attribute $\partial$-reduct
    endif
  endfor;
  /* create $(k+1)$ attribute candidates for $\partial$-reducts from $k$ attribute non-$\partial$-reducts */
  $\mathcal{C}_{k+1} = FunGen(\mathcal{C}_k)$;
endfor;
return $\bigcup_k \mathcal{R}_k$;
```

Next, the *PartitionArrayRepresentation* function (see Section 3.3) is called to create an array representation of π_∂. This representation shall be used multiple times in the *Holds* function, called later in the algorithm, for efficient checking whether candidate attribute sets determine ∂ functionally. Now, the main loop starts. In each k-th iteration, the following is performed:

– The *Holds* function (see Section 3.3) is called to check if k attribute candidates $\mathcal{C}_k$ determine ∂ functionally. The candidates that do are removed from the set of k attribute candidates to the set of ∂-reducts R_k.
– The *FunGen* function (see Section 3.2) is called to create $(k+1)$ attribute candidates $\mathcal{C}_{k+1}$ from the k attribute candidates that remained in $\mathcal{C}_k$.

The algorithm stops when the set of candidates becomes empty.

3.2 Generating Candidates for ∂-Reducts

The *FunGen* function creates $(k+1)$ attribute candidates $\mathcal{C}_{k+1}$ by merging k attribute candidates $\mathcal{C}_k$, which are not ∂-reducts. The algorithm adopts the manner of creating and pruning of candidates introduced in [1] (here: candidate sets of attributes instead of candidates for frequent itemsets). There are merged only those pairs of k attribute candidates $\mathcal{C}_k$ that differ merely on their last attributes (see [1] for justification that this method is lossless and non-redundant). For each new candidate C, π_C is calculated as the product of the partitions wrt. the merged k attribute sets (see Section 3.3 for the *Product* function). The cardinality (*groupNo*) of π_C is also calculated. Now, it is checked for each new $(k+1)$ attribute candidate C, if there is its k attribute subset A not present in $\mathcal{C}_k$. If

```
function $FunGen(\mathcal{C}_k)$;
/* Merging */
forall $A, B \in \mathcal{C}_k$ do
  if $A[1] = B[1] \wedge \ldots \wedge A[k-1] = A[k-1] \wedge A[k] < B[k]$ then begin
    $C = A[1] \cdot A[2] \cdot \ldots \cdot A[k] \cdot B[k]$;
    /* compute partition $C.\pi$ as a product of $A.\pi$ and $B.\pi$, and the number of groups in $C.\pi$ */
    $C.groupNo = Product(A.\pi, B.\pi, C.\pi)$;
    add $C$ to $\mathcal{C}_{k+1}$
  endif;
endfor;
/* Pruning */
forall $C \in \mathcal{C}_{k+1}$ do
  forall $k$ attribute set $A$, such that $A \subset C$ do
    if $A \notin \mathcal{C}_k$ then
      /* $A \subset C$ and $\exists B \subseteq A$ such that $B \rightarrow \{\partial\}$ holds, so $C \rightarrow \partial$ holds, but is not minimal */
      begin delete $C$ from $\mathcal{C}_{k+1}$; break
      end
    elseif $A.groupNo = C.groupNo$ then                    // optional pruning step
      /* $A \rightarrow C$ holds, so $C \rightarrow \{\partial\}$ is not a minimal functional dependency */
      begin delete $C$ from $\mathcal{C}_{k+1}$; break
      end
    endif
  endfor
endfor;
return $\mathcal{C}_{k+1}$;
```

∂-reduct, and hence C is deleted from the set $\mathcal{C}_{k+1}$. Optionally, for each tested k attribute subset A that is present in $\mathcal{C}_k$, it is checked, if $|\pi_A|$ equals $|\pi_C|$. If so, then $A \rightarrow C$ holds (by Property 2.2.2). Hence, $A \rightarrow \{\partial\}$ is not a minimal functional dependency (by Property 2.2.1e), and thus C is deleted from $\mathcal{C}_{k+1}$.

3.3 Using Partitions in Fun

Computing Array Representation of Partition. The *PartitionArrayRepresentation* function returns an array T of the length equal to the number of objects O in DT. For a given attribute C, each element of T is assigned the index of the group in $C.\pi$ to which the index of the element belongs. As a result, j-th element of T informs to which group in $C.\pi$ j-th object in DT belongs, $j = 1..|O|$.

```
function PartitionArrayRepresentation(attribute set $C$);
/* assert: $T$ is an array[$1 \ldots |O|$] */
$i = 1$;
for $i$-th group $G$ in partition $C.\pi$ do begin
   for each $oid$ $G$ do $T[oid] = i$ endfor;
   $i = i + 1$
endfor
return $T$;
```

Verifying Candidate Dependency. The *Holds* function checks, if there is a functional dependency between the set of attributes C and an attribute ∂. It is checked for successive groups G in $C.\pi$, if there is an *oid* in G that belongs to a group in $\partial.\pi$ different from the group in $\partial.\pi$ to which the first *oid* in G belongs (for the purpose of efficiency, the pre-calculated $\partial.T$ representation of the partition for ∂ is applied instead of $\partial.\pi$). If so, this means that G is not contained in one group of $\partial.\pi$ and thus $C \rightarrow \{\partial\}$ is not a functional dependency. In such a case, the function stops returning **false** as a result. Otherwise, if no such group G is found, the function returns **true**, which means that $C \rightarrow \{\partial\}$ is a functional dependency.

```
function Holds($C \rightarrow \{\partial\}$);
/* assert: $\partial.T$ is an array representation of $\partial.\pi$ */
for each group $G$ in partition $C.\pi$ do begin
   $oid$ = first element in group $G$;
   $\partial$-$firstGroup = \partial.T[oid]$;          // the identifier of the group in $\partial.\pi$ to which $oid$ belongs
   for each next element $oid \in G$ do begin
      $\partial$-$nextGroup = \partial.T[oid]$;
      if $\partial$-$firstGroup \neq \partial$-$nextGroup$ then
         /* there are $oids$ in $G$ that identify objects indiscernible wrt. $C$, but discernible wrt. $\partial$ */
         return false                                   // hence, $C \rightarrow \{\partial\}$ does not hold
      endif
   endfor;
endfor;
return true;                                            // $C \rightarrow \{\partial\}$ holds
```

Computing Product of Partitions. The *Product* function computes the partition wrt. the attribute set C and its cardinality from the partitions wrt. the attribute sets A and B. The function examines successive groups wrt. the partition for B. The objects in a given group G in $B.\pi$ are split into maximal subgroups in such a way that the objects in each resultant subgroup are contained in a same group in $A.\pi$. The obtained set of subgroups equals $\{G \cap Y | Y \in A.\pi\}$. Product $C.\pi$ is calculated as the set of all subgroups obtained from all groups in $B.\pi$; i.e., $C.\pi = \bigcup_{G \in B.\pi}\{G \cap Y | Y \in A.\pi\} = \{G \cap Y | Y \in A.\pi \text{ and } G \in B.\pi\} = B.\pi \cap A.\pi$.

In order to calculate the product of the partitions efficiently (with time complexity linear wrt. the number of objects in DT), we follow the idea presented in [5], and use two static arrays T and S: T is used to store an array representation of the partition wrt. A; S is used to store subgroups obtained from a given group G in $B.\pi$.

```
function $Product(A.\pi,\ B.\pi;$ var $C.\pi)$;
/* assert: $T[1..|O|]$ is a static array */
/* assert: $S[1..|O|]$ is a static array with all elements initially equal to $\emptyset$ */
$C.\pi = \{\}$; $groupNo = 0$;
/* calculate an array representation of $A.\pi$ for later multiple use in the $Product$ function */
$T = PartitionArrayRepresentation(A)$; $i = 1$;
for $i$-th group $G$ in partition $B.\pi$ do begin
   $A\text{-}GroupIds = \emptyset$;
   for each element $oid \in G$ do begin
      $j = T[oid]$;                    // the identifier of the group in $A.\pi$ to which $oid$ belongs
      insert $oid$ into $S[j]$; insert $j$ into $A\text{-}GroupIds$
   endfor;
   for each $j \in A\text{-}GroupIds$ do begin
      insert $S[j]$ into $C.\pi$;
      $groupNo = groupNo + 1$; $S[j] = \emptyset$
   endfor;
   $i = i + 1$
endfor;
return $groupNo$;
```

3.4 Using Stripped Partitions in Fun

The representation of partitions that requires storing objects identifiers (*oids*) of all objects in DT may be too memory consuming. In order to alleviate this problem, it was proposed in [5] to store *oids* only for objects belonging to non-singleton groups in a partition representation. Such a representation of a partition is called a *stripped* one. Clearly, the stripped representation is lossless.

```
function $StrippedHolds(C \rightarrow \{\partial\})$;
$i = 1$;
for $i$-th group $G$ in partition $C.\pi$ do begin
   $oid$ = first element in group $G$;
   $\partial\text{-}firstGroup = \partial.T[oid]$;          // the identifier of the group in $\partial.\pi$ to which $oid$ belongs
   if $\partial\text{-}firstGroup$ = null then return false endif;
   /* $\partial.T[oid]$ = null indicates that $oid$ constitutes a singleton group in the partition for $\partial$. */
   /* Hence, no next object in $G$ belongs to this group in $\partial.\pi$ , so $C \rightarrow \{\partial\}$ does not hold.   */
   for each next element $oid \in G$ do begin
      $\partial\text{-}nextGroup = \partial.T[oid]$;
      if $\partial\text{-}firstGroup \neq \partial\text{-}nextGroup$ then
         /* there are $oids$ in $G$ that identify objects indiscernible wrt. $C$, but discernible wrt. $\partial$ */
         return false                                        // hence, $C \rightarrow \{\partial\}$ does not hold
      endif
   endfor;
   $i = i + 1$
endfor;
return true;                                                   // $C \rightarrow \{\partial\}$ holds
```

Example 3.4.1. In Table 1, the partition wrt. $\{ce\}$ equals $\{\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6, 7, 8\}, \{9, 10\}\}$, whereas the stripped partition wrt. $\{ce\}$ equals $\{\{3, 4\}, \{6, 7, 8\}, \{9, 10\}\}$. □

When applying stripped partitions in our *Fun* algorithm instead of usual partitions, one should call the *StrippedHolds* function instead of *Holds*, and the

StrippedProduct function instead of *Product*. The modified parts of the functions have been shadowed in the code below. We note, however, that the *groupNo* field still stores the number of groups in an unstripped partition (singleton groups are not stored, but are counted!).

```
function $StrippedProduct(A.\pi, B.\pi;$ var $C.\pi)$;
$C.\pi = \{\}$; $groupNo = B.groupNo$;
$T = PartitionArrayRepresentation(A)$; $i = 1$;
for $i$-th group $G$ in partition $B.\pi$ do begin
   $A - GroupIds = \emptyset$;
   for each element $oid \in G$ do begin
      $j = T[oid]$;                       // the identifier of the group in $A.\pi$ to which $oid$ belongs
      if $j =$ null then $groupNo = groupNo + 1$;                 // respect singleton subgroups
      else begin insert $oid$ into $S[j]$; insert $j$ into $A$-$GroupIds$ endif
   endfor;
   for each $j \in A - GroupIds$ do begin
      if $|S[j]| > 1$ then
         insert $S[j]$ into $C.\pi$                                  // store only non-singleton groups
      endif ;
      $groupNo = groupNo + 1$; $S[j] = \emptyset$          // but count all groups, including singleton ones
   endfor;
   $groupNo = groupNo - 1$;
   $i = i + 1$
endfor;
/* Clearing of array $T$ for later use */
for $i$-th group $G$ in partition $A.\pi$ do
   for each element $oid \in G$ do $T[oid] =$ null endfor
endfor;
return $groupNo$;
```

4 Experimental Results

We have performed a number of experiments on a few data sets available in UCI Repository datasets (http://www.ics.uci.edu/˜mlearn/MLRepository.html) and other used by the Rough Sets community. We have reported the times of discovering reducts by four variants of *Fun*, as well as, the *TANE*, *SAVGeneticReducer* and *RSESExhaustiveReducer* algorithms. We used the implementation of *TANE* provided by its authors. *SAVGeneticReducer* and *RSESExhaustiveReducer*, used for experiments, come from the Rosetta toolkit. Because of Rosetta limitations, we did not perform experiments with *RSESExhaustiveReducer* on datasets larger than 500 records.

As follows from Table 2, *Fun* is faster than *TANE* and much faster than the both algorithms from Rosetta. The performance of the four variants of Fun is similar. In Figure 1, we plotted times of the performance of *Fun*, *TANE* and *SAVGeneticReducer* for the nursery dataset in a logarithmic scale. The time performance of *Fun* and *TANE* is linear wrt. the number of objects in the dataset. The time performance of *SAVGeneticReducer* is 2 to 3 orders of magnitude greater and is non-linear wrt. the number of objects. In Figures 2-4, we presented the time performance of *Fun* and *TANE* in a linear scale for the nursery,

Table 2. Comparison of *Fun*, *TANE*, *SAVGeneticReducer* and *RSESExhaustiveReducer*. Time is given in milliseconds ($^+$ - originally, time measured in seconds); * - a data set does not contain an object id; F - # of min. functional dependencies; P - applied optional pruning step in *FunGen*; S - applied stripped partitions

Data set $DT = (O, AT \cup \{d\})$			Fun	Fun	Fun	Fun	TANE	SAV Genetic	RSES Exhaustive	
Name	$\|O\|$	$\|AT\|$	-	P	S	PS	S	Reducer	Reducer	F
diabetic.33	33	12	10	10	10	10	30	<500 (or 0 sec)$^+$	<500 (or 0 sec)$^+$	2
diabetic.33*	33	11	20	10	10	10	20	<500 (or 0 sec)$^+$	<500 (or 0 sec)$^+$	10
diabetic	107	12	50	30	30	30	40	<500 (or 0 sec)$^+$	<500 (or 0 sec)$^+$	9
diabetic*	107	11	40	40	20	20	30	<500 (or 0 sec)$^+$	<500 (or 0 sec)$^+$	14
nursery.500	500	9	20	10	10	10	10	<500 (or 0 sec)$^+$	18000 (or 18 sec)$^+$	8
nursery.500*	500	8	20	10	10	10	10	<500 (or 0 sec)$^+$	17000 (or 17 sec)$^+$	2
nursery	12960	9	451	539	471	481	681	274000 (or 274 sec)$^+$	not available$^+$	1
nursery*	12960	8	441	450	451	481	701	247000 (or 247 sec)$^+$	not available$^+$	2
krkopt	8056	6	250	251	260	250	420	1296000(or 1296 sec)$^+$	not available$^+$	1

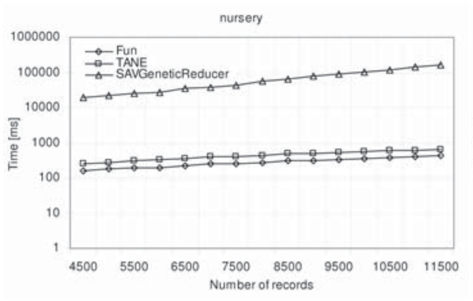

Fig. 1. nursery - logarithmic scale

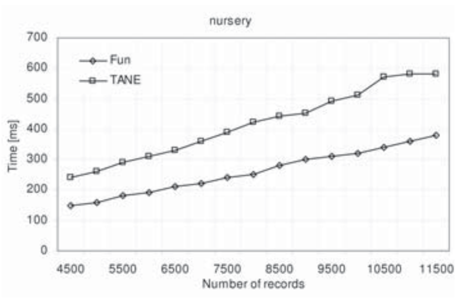

Fig. 2. nursery - linear scale

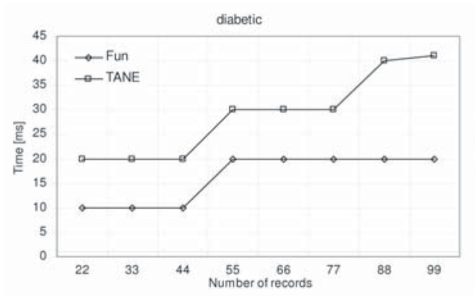

Fig. 3. diabetic - linear scale

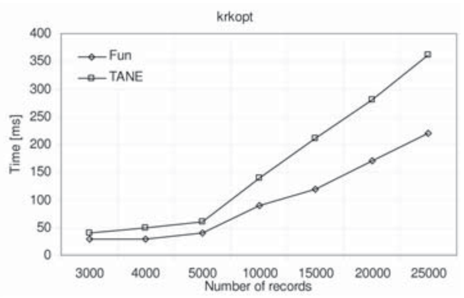

Fig. 4. krkopt - linear scale

diabetic and krkopt datasets, respectively. On average, *TANE* is approximately by 60%, 80%, and 60% slower than *Fun* for the respective datasets.

5 Conclusions and Future Work

We have proposed the *Fun* algorithm for discovering minimal sets of conditional attributes functionally determining a decision attribute, and in particular for computing certain, generalized decision, and μ-distribution reducts. *Fun* is consistently faster than *TANE*, which computes all minimal functional dependencies, and is orders of magnitude faster than *SAVGeneticReducer* and *RSES-ExhaustiveReducer* from Rosetta. The four variants of *Fun*, we have implemented and tested, show similar performance. We are going to continue testing their performance on a diverse large datasets. We intend to specify categories of datasets and appropriate (fastest) variants of *Fun* for them.

References

[1] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast Discovery of Association Rules. In: Advances in KDD. AAAI, Menlo Park, California, pp. 307–328 (1996)

[2] Armstrong, W.W.: Dependency Structures of Data Based Relationships. In: Proc. of IFIP Congress. Geneva, Switzerland, pp. 580–583 (1974)

[3] Bazan, J., Skowron, A., Synak, P.: Dynamic Reducts as a Tool for Extracting Laws from Decision Tables. In: Raś, Z.W., Zemankova, M. (eds.) ISMIS 1994. LNCS, vol. 869, pp. 346–355. Springer, Heidelberg (1994)

[4] Bazan, J., Nguyen, H.S., Nguyen, S.H., Synak, P., Wroblewski, J.: Rough Set Algorithms in Classification Problem. In: Rough Set Methods and Applications, pp. 49–88. Physica- Verlag, Heidelberg (2000)

[5] Huhtala, Y., Karkkainen, J., Porkka, P., Toivonen, H.: TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. The. Computer Journal 42(2), 100–111 (1999)

[6] Jelonek, J., Krawiec, K., Stefanowski, J.: Comparative study of feature subset selection techniques for machine learning tasks. In: Proc. of IIS, Malbork, Poland, pp. 68–77 (1998)

[7] Kryszkiewicz, M.: Comparative Study of Alternative Types of Knowledge Reduction in Inconsistent Systems. Intl. Journal of Intelligent Systems 16(1), 105–120 (2001)

[8] Kryszkiewicz, M.: Certain, Generalized Decision, and Membership Distribution Reducts versus Functional Dependencies in Incomplete Systems. RSEISP, LNAI (2007)

[9] Kryszkiewicz, M., Cichon, K.: Towards Scalable Algorithms for Discovering Rough Set Reducts. In: Peters, J.F., Skowron, A., Grzymała-Busse, J.W., Kostek, B.., Świniarski, R.W., Szczuka, M. (eds.) Transactions on Rough Sets I. Journal Subline, LNCS, vol. 3100, pp. 120–143. Springer, Heidelberg (2004)

[10] Lin, T.Y.: Rough Set Theory in Very Large Databases. In: Proc. of CESA IMACS, Lille, France, vol. 2, pp. 936–941 (1996)

[11] Lopes, S., Petit, J.-M., Lakhal, L.: Efficient Discovery of Functional Dependencies and Armstrong Relations. In: Proc. of EDBT, pp. 350–364 (2000)

[12] Nguyen, S.H., Skowron, A., Synak, P., Wroblewski, J.: Knowledge Discovery in Databases: Rough Set Approach. In: Proc. of IFSA, vol. 2, pp. 204–209, Prague, (1997)

[13] Pawlak, Z.: Rough Sets: Theoretical Aspects of Reasoning about Data, vol. 9. Kluwer Academic Publishers, Boston (1991)

[14] Shan, N., Ziarko, W., Hamilton, H.J., Cercone, N.: Discovering Classification Knowledge in Databases Using Rough Sets. In: Proc. of: KDD, pp. 271–274 (1996)

[15] Skowron, A.: Boolean Reasoning for Decision Rules Generation. ISMIS, 295–305 (1993)

[16] Slezak, D.: Approximate Reducts in Decision Tables. In: Proc. of IPMU, Granada, Spain, vol. 3, pp. 1159–1164 (1996)

[17] Slezak, D.: Searching for Frequential Reducts in Decision Tables with Uncertain Objects. In: Proc. of RSCTC, Warsaw, 1998, pp. 52–59. Springer, Heidelberg (1998)

[18] Slowinski, R. (ed.): Intelligent Decision Support, Handbook of Applications and Advances of the Rough Sets Theory, vol. 11. Kluwer Academic Publishers, Boston (1992)

[19] Stepaniuk, J.: Approximation Spaces, Reducts and Representatives. In: Rough Sets in Data Mining and Knowledge Discovery, Springer, Berlin (1998)

[20] Wroblewski, J.: Finding Minimal Reducts Using Genetic Algorithms. In: Proc. of JCIS, September/October 1995, Wrightsville Beach, NC, pp. 186–189 (1995)