Mining Determining Sets for Partially Defined Functions

Dan A. Simovici, Dan Pletea, and Rosanne Vetro

Univ. of Massachusetts Boston, Dept. of Comp. Science, 100 Morrissey Blvd. Boston, Massachusetts 02125 USA {dsim,dpletea,rvetro} at cs.umb.edu

Abstract. This paper describes an algorithm that determines the minimal sets of variables that determine the values of a discrete partial function. The Apriori-like algorithm is based on the dual hereditary property of determining sets. Experimental results are provided that demonstrate the efficiency of the algorithm for functions with up to 24 variables. The dependency of the number of minimal determining sets on the size of the specification of the partial function is also examined.

1 Introduction

Partially defined finite functions are studied by both mathematicians and engineers due to their many technical applications, particularly in designing switching circuitry. They model such diverse circuits as logical programmable arrays, or content addressable memory. The performance of such circuits (including wiring complexity, power dissipation, etc.) is heavily influenced by the number of arguments on which the function implemented by the circuit depends effectively.

The goal of this paper is to present an algorithm to generate various sets of input variables on which a partial function depends using an approach inspired by Apriori, a well-known data mining algorithm developed for determining frequent item sets in transactional databases [1–3].

The proposed algorithm is based on the fundamental observation that and superset of a determining set for a partially defined function f is itself a determining set for f. We use this dual heredity property of determining sets to formulate an Apriori-like algorithm that computes the determining sets by traversing the lattice of subsets of the set of variables.

This problem has been addressed in [4] using an algebraic minimization algorithm that applies to functions that depend on small number of variables. Our approach is distinct and involves techniques inspired by data mining. Additionally, it has the advantage of not being linked to any value of the input or output radix of the partial function f.

The rest of the paper is organized as follows. In Section 2 we introduce the notion of determining set for a partial function and examine a few properties of these sets that play a role in our algorithm. This algorithm is presented in Section 3. Section 4 discusses experimental work related to the algorithm. In the last section (Section 5) we present conclusions and future work.

2 Determining Sets for Partially Defined Functions

We denote the finite set $\{0, 1, \ldots, n-1\}$ by $\mathbf{n}$. The partial functions that we study have as domain a subset of the finite set $\mathbf{r}^n$ and as range a subset of the finite set $\mathbf{p}$ for some positive natural numbers r and p, referred to as the *input radix* and the *output radix* of the function, respectively. The set of all such partial functions is denoted by $\mathsf{PF}(\mathbf{r}^n, \mathbf{p})$. If $f \in \mathsf{PF}(\mathbf{r}^n, \mathbf{p})$ we denote by $\mathrm{Dom}(f)$ the set of all n-tuples $(a_1, \ldots, a_n)$ in $\mathbf{r}^n$ for which $f(a_1, \ldots, a_n)$ is defined.

A partial function $f \in \mathsf{PF}(\mathbf{r}^n, \mathbf{p})$ is specified as a table T_f having columns labelled by the argument variables $x_1, \ldots, x_n$ and by the output variable y. If $f(a_1, \ldots, a_n) = b$ we have in the table T_f the $(n+1)$-tuple $t = (a_1, \ldots, a_n, b)$. For example, in Table 1 we show a partial function defined on all triplets in $\mathbf{3}^3$ that contain at least two non-zero elements, and ranging in the set $\mathbf{4}$: The number of rows of the table that represents

Table 1. Tabular Representation of a Partial Function

x_1	x_2	x_3	y
0	1	1	0
0	1	2	1
0	2	1	2
0	2	2	2
1	0	1	3
1	0	2	3
2	0	1	3
2	0	2	3
1	1	0	2
1	2	0	2
2	1	0	1
2	2	0	0

a partial function defined on $\mathbf{r}^n$ can range between 0 and r^n. Usually, the number of rows of such a function is smaller than r^n and, often this number is much smaller. Tuples $(a_1, \ldots, a_n)$ that do not belong to the definition domain of f are considered as "don't care" tuples, that is, as input sequences that are unlikely to occur as inputs of the functions, or the output of the function for such inputs is indifferent to the designer.

For a tuple t in T_f and a set of variables $U \subseteq \{x_1, \ldots, x_n, y\}$ we denote by $t[U]$ the *projection* of t on U, that is, the restriction of t to the set U. If U consists of one variable we denote the projection $t[\{z\}]$ just by $t[z]$.

Definition 1. *A set of variables $V = \{x_{i_0}, \ldots, x_{i_{p-1}}\}$ is a* determining set *for the partial function f if for every two tuples t and s from T_f, $t[V] = s[V]$ implies $t[y] = s[y]$.*

In other words, V is a determining set for the partial function f if $t = (a_0, \ldots, a_{n-1}, b)$ and $s = (c_0, \ldots, c_{n-1}, d)$ in T_f such that $a_{i_k} = c_{i_k}$ for $1 \leq k \leq p$ implies $b = d$. The collection of determining sets for f is denoted by $\mathsf{DS}(f)$.

V is a *minimal determining set for* f if V is a determining set for f and there is no strict subset of V that is a determining set for f. The set of minimal determining sets of f is denoted by $\mathsf{MDS}(f)$. Our main purpose is to present an algorithm that extracts the minimal determining sets for a partially specified function.

We introduce a partial order relation "$\sqsubseteq$" on the set of partial $\mathsf{PF}(\mathbf{r}^n, \mathbf{p})$ by defining $f \sqsubseteq g$ if $\mathrm{Dom}(f) \subseteq \mathrm{Dom}(g)$ and $f(a_1, \ldots, a_n) = g(a_1, \ldots, a_n)$ for every $(a_1, \ldots, a_n)$. In other words, we have $f \sqsubseteq g$ if g is an extension of f.

The following simple statement is crucial to the proposed algorithm.

Theorem 1. *Let f and g be two partial functions in $\mathsf{PF}(\mathbf{r}^n, \mathbf{p})$. If $V \in \mathsf{DS}(f)$ and $V \subseteq W$, then $W \in \mathsf{DS}(f)$. Furthermore, if $f \sqsubseteq g$, then $\mathsf{DS}(g) \subseteq \mathsf{DS}(f)$.*

Proof. If V and W are two sets of variables such that $V \subseteq W$ and t, s are two tuples in T_f, then $t[W] = s[W]$ implies $t[V] = s[V]$. Therefore, if V is a determining set for f and $t[W] = s[W]$, it follows that $t[V] = s[V]$, which implies $t[y] = s[y]$. Thus, W is a determining set for f.

For the second part of the theorem, observe that if $f \sqsubseteq g$ and $V \in \mathsf{DS}(g)$, then $t[V] = s[V]$ implies $t[y] = s[y]$, for every $t, s \in \mathrm{Dom}(g)$. Since $\mathrm{Dom}(f) \subseteq \mathrm{Dom}(g)$, the same implication holds for any two tuples in $\mathrm{Dom}(f)$, so $V \in \mathsf{DS}(f)$. $\square$

Note that if $f \sqsubseteq g$ and $V \in \mathsf{MDS}(g)$, then there exists $Z \in \mathsf{MDS}(f)$ such that $Z \subseteq V$.

3 An Apriori-like Algorithm for Mining MDSs

Our algorithm uses Rymon trees (see [5, 6]) also known as set enumeration trees, a device that is useful for the systematic enumeration of the subsets of a set S. The subsets of a set S (which constitute the power set of S, $\mathcal{P}(S)$) are listed using a pre-imposed total order on the underlying set S. The total order on S is specified by an one-to-one function $\mathsf{ind} : E \longrightarrow N$.

For every subset $U \subseteq S$, define its view as

$$\mathsf{view}(\mathsf{ind}, U) = \{s \in S \mid \mathsf{ind}(s) > \max_{u \in U} \mathsf{ind}(u)\}$$

Definition 2. *Let $\mathcal{F}$ be a collection of sets closed under inclusion (i.e., if $U \in \mathcal{F}$ and $U \subseteq V$, then $V \in \mathcal{F}$).*

The labelled tree $\mathcal{T}$ is a Rymon tree for $\mathcal{F}$ if

(i) *the root of $\mathcal{T}$ is labelled by the empty set $\emptyset$, and*
(ii) *the children of a node labelled U in $\mathcal{T}$ are $\{U \cup \{e\} \in \mathcal{F} \mid e \in \mathsf{view}(\mathsf{ind}, U)\}$.*

In Figure 1 we show the Rymon tree for the complete power set of a set $S = \{1, 2, 3, 4\}$. The proposed algorithm takes as input a partially defined function f and outputs a collection of sets with minimum number of variables that f depends on. The algorithm performs a breadth-first search on the Rymon tree for the power-set of the set of variables $E = \{x_1, x_2, ..., x_n\}$ of f. The search stops when all the sets with the minimum number of variables that functions f depends on are found; these sets are referred to as determining sets. The minimum number corresponds to the lowest level in the Rymon

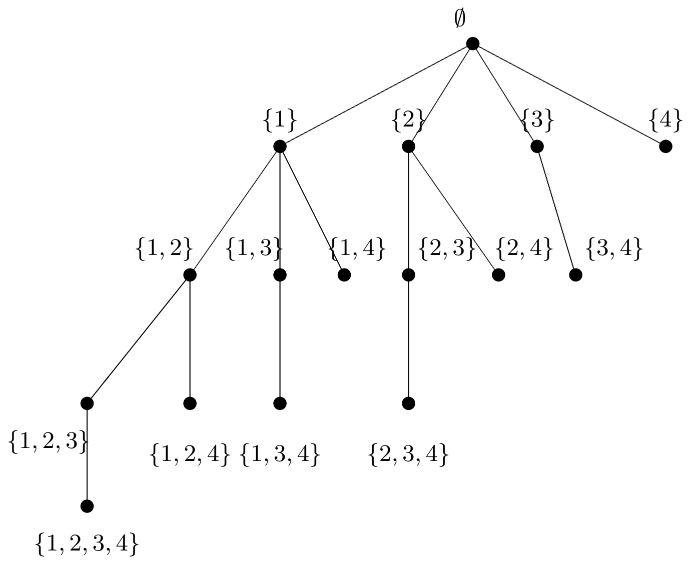

Fig. 1. The Rymon tree for the $\mathcal{P}(\{1,2,3,4\})$

tree where the first solution set is found since all the levels below have nodes containing sets with a higher number of variable than any level above it.

In Algorithm 2 we denote the breadth first search queue by Q, the array of children of a node X by $\mathsf{Child}[X]$, and the tree level of the determining sets by dLevel. The algorithm is using the following methods:

ENQUEUE(Q,V)	inserts node V in queue Q;
DEQUEUE(Q)	removes the first element of queue Q;
LEVEL(V)	returns the level of node V in the tree;
IS_DSET(v)	informs if the set of variables corresponding to node V is a determining set for the partially defined function f;
ADD($\mathcal{D},V$)	add the set of variables corresponding to node V to $\mathcal{D}$.

The core of the algorithm is the procedure IS_DSET(V) that has as an input argument a set of variables V and returns **true** if V is a determining set and false, otherwise. In principle, if T_f is a database table, the implementation of IS_DSET could be done using embedded SQL by running the query

$$\textbf{select}\ \mathsf{count}(\textbf{distinct}\ y)\ \textbf{from}\ T_f\ \textbf{group by}\ V.$$

It is clear that if all values returned for count(**distinct** y) equal 1, then V is a determining set for f. In practice, the overhead entailed by using the database facility impacts negatively on the performance of the algorithm, which lead us to another solution that is based on storing T_f as a file and searching that file.

Fig. 2. Computing $\mathsf{MDS}(f)$

```
Input: A partially defined function $f$
Result: A collection $\mathcal{D}$ of minimal determining variables sets
begin
    dLevel $\longleftarrow \infty$
    ENQUEUE($Q$,$\emptyset$)
    while $Q \neq \emptyset$ do
        $X \longleftarrow$ DEQUEUE($Q$)
        foreach $V \in Child[X]$ do
            ENQUEUE($Q$,$V$)
            if $\mathcal{D} = \emptyset$ or $LEVEL(v) \leq dLevel$ then
                if IS_DSET[V] then
                    ADD($\mathcal{D}, V$)
                    if $dLevel = \infty$ then
                        dLevel $= LEVEL(V)$
            else
                break
end
```

The procedure *IS_DSET* shown in Figure 3 makes use of a hash table *MAP*, where the key is determined by the chosen set of variables. The following methods are used:

GET_VARIABLES(V): retrieves the set of variables corresponding to node V
GET_VALUES$(tuple, S)$: retrieves the values of the variables in S
ELEMENT(MAP, key): returns the object instance stored in *MAP*
that contains a certain *key*
GET_FVALUE(y): returns the function value of the object instance y
$F(tuple)$: returns the function value of a certain tuple
ADD$(MAP, key, F(tuple))$: adds an object instance containing
a key and function value to the *MAP*.

The following variables are used in the IS_DSET procedure:

S	set of variables
v	a node in the tree
$File$	input file containing the tuples of a partially defined function
tuple	a row of the input file
key	a set with the values of the variables in S
MAP	a hash structure that stores objects containing a key and a function value
y	an object instance stored in the MAP

4 Experimental Results

We carried out experiments on a Windows Vista 64-bit machine with 8Gb RAM and 2 $\times$ Quad Core Xeon Proc E5420, running at 2.50 GHz with a 2$\times$6Mb L2 cache. The algorithm was written in Java 6.

Fig. 3. Procedure IS_DSET(V)

```
Input: A node containing a subset of the variables set
Output: true if the set is a determining one, false, otherwise
begin
    $S \longleftarrow GET\_VARIABLES(V)$
    foreach $tuple \in File$ do
        $key \longleftarrow GET\_VALUES(tuple, S)$
        if $key \in MAP$ then
            $y \longleftarrow ELEMENT(MAP, key)$
            if $F(tuple) \neq GET\_FVALUE(y)$ then
                return false
                break
        else
            ADD(MAP,key,F(tuple))
    return true
end
```

We analyze the results in terms of running time, minimum number of variables of a determining set, and the number of determining sets as a function of the number of tuples in T_f.

A program that randomly generates comma separated text files representing partially defined functions with 8, 16 or 24 variables was developed. These values were chosen based on the experiments made in the related work of T. Sasao [4].

One hundred files were randomly generated for each type of partially defined function (with 8, 16, and 24 variables) using an input radix $r = 3$ and an output radix $p = 5$.

Note that a totally defined function with 8 variables and $r = 3$ has $3^8 = 6561$ tuples. In our experiments, we randomly generated 1000 tuples for partially defined functions with 8 variables. For functions that depend on 16 and 24 arguments we generated 5000 tuples because the number of tuples for completely defined functions with 16 or 24 variables is much higher.

In the experiments, we evaluate the performance of the algorithm with a varying number of tuples. By Theorem 1, if $(f_1, f_2, \ldots, f_k)$ is a sequence of functions such that

$$f_1 \sqsubseteq f_2 \sqsubseteq \cdots \sqsubseteq f_k,$$

we have

$$\mathsf{DS}(f_k) \subseteq \cdots \subseteq \mathsf{DS}(f_2) \subseteq \mathsf{DS}(f_1).$$

In other words, when we start with a partial function f_1 with a small specification table T_{f_k} and we expend sequentially the specification of the functions, the number of determining sets will decrease. The experiments compare the results for files with 8, 16 and 24 variables and they contain averages of the values corresponding to time, minimum number of variables the function depends on, and number of sets with minimum number of elements the function depends on as a function of the number of tuples. In our case, $k \in \{10, 15, 20, 30, 40, 50, 75, 90, 100, 200\}$. The averages are evaluated over 100 functions within each group of generated functions (8, 16 and 24 variables).

As shown in Fig. 4, the running time of the algorithm to find a solution increases with the number of tuples because in most cases, the algorithm needs to search deeper in the Rymon tree. Also, the time increases exponentially with the number of variables. The algorithm performs a breadth-first search and functions with more variables will have trees with a larger branching factor.

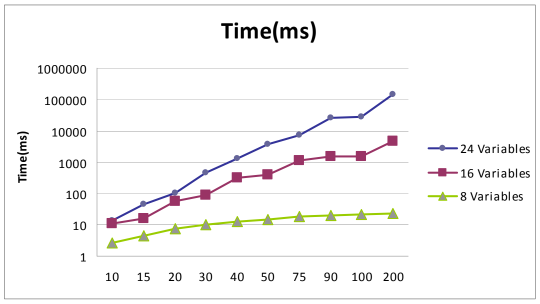

Fig. 4. Dependency of average time on number of tuples

Fig 5 shows that the minimum number of variables the function depends on is related to the number of tuples k. As k increases, the constraints imposed on the problem become more extensive, and the minimum number of variables that determine the value of the function increases.

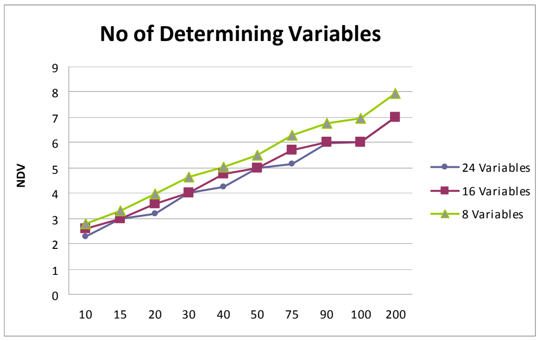

Fig. 5. Average size of minimal determining set for 8, 16 and 24 variables, as a function of the number of tuples.

Finally, the experiments also show that the average number of minimal determining sets decreases as we extend the partial functions by introducing more tuples. Fig 6 illustrates this decrease for functions with 8 and 16 variables. The decrease is not as noticeable for functions with 24 variables because these functions have a large number of possible tuples and this behavior can only be observed for a much higher value of k than the maximum used in experiments presented here.

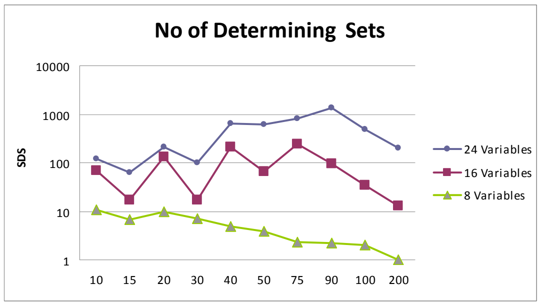

Fig. 6. Average size of $\mathsf{MDS}(f)$ for 8, 16 and 24 variables, as a function of the number of tuples.

5 Concluding Remarks

This paper introduces an algorithm to determine the minimum number of variables on which a partially defined function depends on, as well as determine all sets of variables with minimum number of elements that define the function. The algorithm is based on traversing Rymon trees using a breadth-first search technique. We believe that the algorithm will be helpful for digital circuit design since it allows to determine the possible sets of variables on which a partial function depends starting from a tabular specification of the function

We intend to approach the same problem using a clustering technique for discrete functions starting from a semi-metric that measures the discrepancy between the kernel partitions of these functions.

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In Buneman, P., Jajodia, S., eds.: Proceedings of the 1993 International Conference on Management of Data, Washington, D.C., ACM, New York (1993) 207–216
2. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Technical Report C-1997-8, University of Helsinki (1997)

3. Zaki, M.J., Hsiao, C.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering **17** (2005) 462–478
4. Sasao, T.: On the number of variables to represent sparse logic functions. In: 17th International Workshop on Logic and Synthesis (IWLS-2008), Lake Tahoe, California, USA, IEEE-CS (2008) 233–239
5. Rymon, R.: Search through systematic set enumeration. In Nebel, B., Rich, C., Swartout, W.R., eds.: Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, Cambridge, MA, Morgan Kaufmann, San Mateo, CA (1992) 539–550
6. Simovici, D.A., Djeraba, C.: Mathematical Tools for Data Mining – Set Theory, Partial Orders, Combinatorics. Springer-Verlag, London (2008)