Calculating a New Data Mining Algorithm for Market Basket Analysis

Zhenjiang Hu[1], Wei-Ngan Chin[2], Masato Takeichi[1]

[1] Department of Information Engineering
University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: {hu,takeichi}@ipl.t.u-tokyo.ac.jp

[2] Department of Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
Email: chinwn@comp.nus.edu.sg

Abstract. The general goal of data mining is to extract interesting correlated information from large collection of data. A key computationally-intensive subproblem of data mining involves finding frequent sets in order to help mine association rules for market basket analysis. Given a bag of sets and a probability, the frequent set problem is to determine which subsets occur in the bag with some minimum probability. This paper provides a convincing application of program calculation in the derivation of a completely new and fast algorithm for this practical problem. Beginning with a simple but inefficient specification expressed in a functional language, the new algorithm is calculated in a systematic manner from the specification by applying a sequence of known calculation techniques.

1 Introduction

Program derivation has enjoyed considerable interests over the past two decades. Early work concentrated on deriving programs in imperative languages, such as Dijkstra's Guarded Command Language, but now it has been realized that functional languages offer a number of advantages over imperative ones.

- Functional languages are so abstract that they can express the specifications of problems in a more concise way than imperative languages, resulting in programs that are shorter and easier to understand.
- Functional programs can be constructed, manipulated, and reasoned about, like any other kind of mathematics, using more or less familiar known algebraic laws.
- Functional languages can often be used to express both clear specification and its efficient solution, so the derivation can be carried out within a single formalism. In contrast, the derivation for imperative languages often rely on a separate (predicate) calculus for capturing both specification and program properties.

Such derivation in a single formalism is often called *program calculation* [Bir89, BdM96], as opposed to simply program derivation. Many attempts have been made to apply the program calculation for the derivation of various kinds of efficient programs [Jeu93], and for the construction of optimization passes of compilers [GLJ93, TM95]. However, people are still expecting more *convincing* and *practical* applications where program calculation can give a better result, while other approaches could falter.

This paper aims to illustrate a practical application of program calculation, by deriving a completely new algorithm to solve the problem for finding frequent sets - an important building block of data mining applications [AIS93, MT96]. In this problem, we are given a set of items and a large collection of transactions which are essentially subsets of these items. The task is to find all sets of items that occur in the transactions frequently enough - exceeding a given threshold. More concrete explanation of the problem can be found in Section 2.

The most well-known classical algorithm for finding frequent set is the Apriori algorithm [AIS93] (from which many improved versions have been proposed) which relies on the property that a set can only be frequent if and only if all of its subsets are frequent. This algorithm builds a tree of frequent sets in a level-wise fashion, starting from the leaves of the tree. Firstly, it counts all the 1-item sets (sets with a single item), and identifies those counts which exceed the threshold, as frequent 1-item sets. Then it combines these to form candidate (potentially frequent) 2-item sets, counts them in order to determine the frequent 2-item sets. It continues by combining the frequent 2-item sets to form candidate 3-item sets, counting them before determining which are the frequent 3-item sets, and so forth. The Apriori algorithm stops when there are no more frequent n-set found.

Two important factors, which govern the performance of this algorithm, are the number of passes made over the transactions, and the efficiency of each of these passes.

- The database that records all transactions is likely to be very large, so it is often beneficial for as much information to be discovered from each pass, so as to reduce the total number of passes [BMUT97].
- In each pass, we hope that counting can be done efficiently and less candidates are generated for later check. This has led to the studies of different pruning algorithms as in [Toi96, LK98].

Two essential questions arise; what is the least number of passes for finding all frequent sets, and could we generate candidates that are so necessary that they will not be pruned later? Current researches in data mining, as far as we are aware, have not adequately address both these issues. Instead, they have been focusing on the improvement of the Apriori algorithm, while taking for granted that database of transactions should only be traversed transaction by transaction.

We shall show that program calculation indeed provides us with a nice framework to examine into these practical issues for data mining application. In this

framework, we can start with a straightforward functional program that solve the problem. This initial program may be terribly inefficient or practically infeasible. We then try to improve it by applying a sequence of calculations such as fusion, tabulation, and accumulation, in order to reduce the number of passes and to avoid generating unnecessary candidates. As will be shown later in this paper, our program calculation can yield a completely novel algorithm for finding frequent sets, which uses only a single pass of the transactions, and generates only necessary candidates during execution. Furthermore, the new algorithm is guaranteed to be correct with respect to the initial straightforward program due to our use of correctness-preserving calculation.

The rest of this paper is organized as follows. We begin by giving a straightforward functional program for finding frequent sets in Section 2. We then go to apply the known calculation techniques of fusion, accumulation, base-case filter promotion and tabulation to the initial functional program to derive an efficient program in Section 3. Discussion on the features of our derived program and the conclusion of the paper are given in Section 4 and 5 respectively.

2 Specification

Within the area of data mining, the problem of deriving associations from data has received considerable attention [AIS93, Toi96, BMUT97], and is often referred to as the "market-basket" problem. One common formulation of this problem is finding association rules which are based on *support* and *confidence*. The support of an itemset (a set of items) I is the fraction of transactions that the itemset occurs in (is a subset of). An itemset is called *frequent* if its support exceeds a given threshold σ. An association rule is written as $I \to J$ where I and J are itemsets. The confidence of the rule is the fraction of the transaction I that also contains J. For the association rule $I \to J$ to hold, $I \cup J$ must be frequent and the confidence of rule must exceed a given confidence threshold, γ. Two important steps for mining association rules are thus:

- Find frequent itemsets for a given support threshold, σ.
- Construct rules that exceed the confidence threshold, γ, from the frequent itemsets.

Of these two steps, finding frequent sets is the more computationally-intensive subproblem, and have received the lion share of data mining community's attention. Let us now formalize a specification for this important subproblem.

Suppose that a shop has recorded the set of objects purchased by each customer on each visit. The problem of finding frequent sets is to compute all subsets of objects that appear frequently in customers' visits with respect to a specific threshold. As an example, suppose a shop has the following object set:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

and the shop recorded the following customers' visits:

$$\begin{array}{ll} \text{visit 1:} & \{1,2,3,4,7\} \\ \text{visit 2:} & \{1,2,5,6\} \\ \text{visit 3:} & \{2,9\} \\ \text{visit 4:} & \{1,2,8\} \\ \text{visit 5:} & \{5,7\} \end{array}$$

We can see that 1 and 2 appear together in three out of the five visits. Therefore we say that the subset $\{1,2\}$ has frequency ratio of 0.6. If we set the frequency ratio threshold to be 0.3, then we know that the sets of

$$\{1\},\ \{2\},\ \{5\},\ \{7\} \text{ and } \{1,2\}$$

pass this threshold, and thus they should be returned as the result of our frequent set computation.

To simplify our presentation, we impose some assumption on the three inputs, namely object set os, customers' visits vss, and threshold $least$. We shall represent the objects of interest using an ordered list of integers without duplicated elements, e.g.,

$$os = [1,2,3,4,5,6,7,8,9,10,11]$$

and represent customers' purchasing visits by a list of the sublists of os, e.g.,

$$vss = [[1,2,3,4,7],[1,2,5,6],[2,9],[1,2,8],[5,7]].$$

Furthermore, for threshold, we will use an integer, e.g.,

$$least = 3$$

to denote the *least* number of appearances in the customers' visits, rather than using a probability ratio.

Now we can solve the frequent set problem straightforwardly by the following pseudo Haskell program[3]

$$\begin{array}{ll} fs & :: [Int] \to [[Int]] \to Int \to \{[Int]\} \\ fs\ os\ vss\ least & = (fsp\ vss\ least) \triangleleft (subs\ os). \end{array}$$

It consists of two passes that can be read as follows.

1. First, we use $subs$ to enumerate all the sublists of the object list os, where $subs$ can be defined by

$$\begin{array}{ll} subs & :: [a] \to \{[a]\} \\ subs\ [\,] & = \{[\,]\} \\ subs\ (x:xs) & = subs\ xs \cup (x:) * subs\ xs. \end{array}$$

[3] We assume that the readers are familiar with the Haskell notation in this paper. In addition, we say that our Haskell programs are "pseudo" in the sense that they include some additional notations for sets.

We use the infix $*$ to denote the map function on sets. Similar to the *map* function on lists, it satisfies the so-called *map-distributivity* property (we use $\circ$ to denote function composition):

$$(f*) \circ (g*) = (f \circ g)*.$$

2. Then, we use the predicate fsp to filter the generated sublists to keep only those that appear frequently (exceeding the threshold $least$) in customers' visits vss. Such fsp can be easily defined by

$$\begin{array}{ll} fsp & :: [[Int]] \to Int \to [Int] \to Bool \\ fsp\ vss\ least\ ys & = \#((ys\ `isSublist`) \triangleleft vss) \geq least \end{array}$$

Note that for ease of program manipulation, we use the shorten notation: $\#$ to denote function $length$, and $p\triangleleft$ to denote $filter\ p$. The filter operator enjoys the *filter-element-map* property (that is commonly used in program derivation e.g. [Bir84]):

$$(p\triangleleft) \circ ((x :)*) = ((x :)*) \circ ((p \circ (x :))\triangleleft)$$

and the *filter-pipeline* property:

$$(p\triangleleft)\ \circ\ (q\triangleleft) = (\lambda x.(p\ x\ \wedge\ q\ x))\triangleleft.$$

In addition, $xs\ `isSublist`\ ys$ is true if xs is a sublist of ys, and false otherwise:

$$\begin{array}{ll} [\,]\ `isSublist`\ ys & = True \\ (x : xs)\ `isSublist`\ ys & = xs\ `isSublist`\ ys\ \wedge\ x\ `elem`\ ys\ \wedge \end{array}$$

So much for our specification program which is simple, straightforward, and easy to understand. No attention has been paid to efficiency or to implementation details. In fact, this initial functional program is practically infeasible for all but the very small object set, because the search space of potential frequent sets consists of $2^{\#os}$ sublists.

3 Derivation

We shall demonstrate how the exponential search space of our initial concise program can be reduced dramatically via program calculation. Specifically, we will derive an *efficient* program for finding frequent sets from the specification

$$fs\ os\ vss\ least = (fsp\ vss\ least) \triangleleft (subs\ os)$$

by using the known calculation techniques of fusion [Chi92], generalization (accumulation) [Bir84, HIT99], base-case filter promotion [Chi90], and tabulation [Bir80, CH95].

3.1 Fusion

Fusion is used to merge two passes (from nested recursive calls) into a single one, by eliminating intermediate the data structure passing between the two passes. Notice that our fs has two passes, and the intermediate data structure is huge containing all the sublists of os. We shall apply the fusion calculation to eliminate this huge intermediate data structure by the following calculation via an induction on os.

$$
\begin{array}{ll}
 & fs\ [\,]\ vss\ least \\
= & \quad \{\ \text{def. of } fs\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ [\,]) \\
= & \quad \{\ \text{def. of } subs\ \} \\
 & (fsp\ vss\ least) \triangleleft \{[\,]\} \\
= & \quad \{\ \text{def. of } \triangleleft \text{ and } fsp\ \} \\
 & \text{if } \#(([\,]\ `isSublist`) \triangleleft vss) \geq least \text{ then } \{[\,]\} \text{ else } \{\,\} \\
= & \quad \{\ isSublist\ \} \\
 & \text{if } \#((\lambda ys.True) \triangleleft vss) \geq least \text{ then } \{[\,]\} \text{ else } \{\,\} \\
= & \quad \{\ \text{simplification}\ \} \\
 & \text{if } \#vss \geq least \text{ then } \{[\,]\} \text{ else } \{\,\}
\end{array}
$$

And

$$
\begin{array}{ll}
 & fs\ (o : os)\ vss\ least \\
= & \quad \{\ \text{def. of } fs\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ (o : os)) \\
= & \quad \{\ \text{def. of } subs\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os \cup (o :) * (subs\ os)) \\
= & \quad \{\ \text{def. of } \triangleleft\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os)\ \cup \\
 & (fsp\ vss\ least) \triangleleft ((o :) * (subs\ os)) \\
= & \quad \{\ \text{by } \textit{filter-element-map} \text{ property}\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os)\ \cup \\
 & (o :) * ((fsp\ vss\ least \circ (o :)) \triangleleft (subs\ os)) \\
= & \quad \{\ \text{calculation for equation (1)}\ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os)\ \cup \\
 & (o :) * ((fsp\ ((o\ `elem`) \triangleleft vss)\ least) \triangleleft (subs\ os))
\end{array}
$$

To complete the above calculation, we need to show that

$$fsp\ vss\ least \circ (o :)\ =\ fsp\ ((o\ `elem`) \triangleleft vss)\ least. \tag{1}$$

This can be easily shown by the following calculation.

$$
\begin{array}{ll}
 & fsp\ vss\ least \circ (o :) \\
= & \quad \{\ \text{def. of } fsp\ \} \\
 & (\lambda ys.(\#((ys\ `isSublist`) \triangleleft vss) \geq least)) \circ (o :) \\
= & \quad \{\ \text{function composition}\ \} \\
 & \lambda ys.(\#(((o : ys)\ `isSublist`) \triangleleft vss) \geq least) \\
= & \quad \{\ \text{def. of } isSublist\ \} \\
 & \lambda ys.(\#((\lambda xs.(ys\ `isSublist`\ xs\ \wedge\ o\ `elem`\ xs)) \triangleleft vss) \geq least) \\
= & \quad \{\ \text{by } \textit{filter-pipeline} \text{ property}\ \} \\
 & \lambda ys.(\#((ys\ `isSublist`) \triangleleft ((o\ `elem`) \triangleleft vss)) \geq least) \\
= & \quad \{\ \text{def. of } fsp\ \} \\
 & fsp\ ((o\ `elem`) \triangleleft vss)\ least
\end{array}
$$

To summarize, we have obtained the following program, in which the intermediate result used to connect the two passes have been eliminated.

$$\begin{array}{ll} fs\ [\,]\ vss\ least & = \text{if } \#vss \geq least \text{ then } \{[\,]\} \text{ else } \{\,\} \\ fs\ (o : os)\ vss\ least & = fs\ os\ vss\ least \cup \\ & \quad \underline{(o :)*}(fs\ os\ ((o\ `elem`) \triangleleft vss)\ least) \end{array}$$

3.2 Generalization/Accumulation

Notice that the underlined part in the above program for insert o to every element of a list will be rather expensive if the the list consists of a large number of elements. Fortunately, this could be improved by introducing an accumulating parameter in much the same spirit as [Bir84, HIT99]. To this end, we generalize fs to fs', by introducing an accumulating parameter as follows.

$$fs'\ os\ vss\ least\ r = (r\ ++\ ) * (fs\ os\ vss\ least)$$

And clearly we have

$$fs\ os\ vss\ least = fs'\ os\ vss\ least\ [\,].$$

Calculating the definition for fs' is easy by induction on os, and thus we omit the detailed derivation. The end result is as follows.

$$\begin{array}{ll} fs'\ [\,]\ vss\ least\ r & = \text{if } \#vss \geq least \text{ then } \{r\} \text{ else } \{\,\} \\ fs'\ (o : os)\ vss\ least\ r & = fs'\ os\ vss\ least\ r \cup \\ & \quad fs'\ os\ ((o\ `elem`) \triangleleft vss)\ least\ (r\ ++\ [o]) \end{array}$$

The accumulation transformation has successfully turned an expensive map operator of $(o :)*$ into a simple operation that just appends o to r. In addition, we have got a nice side-effect from the accumulation transformation in that fs' is defined in an *almost* tail recursive form, in the sense that each recursive call produces independent part of the resulting list. This kind of recursive form is used by the base-case filter promotion technique of [Chi90].

3.3 Base-case Filter Promotion

From the second equation (inductive case) of fs', we can see that computation of

$$fs'\ os\ vss\ least\ rs$$

will need $2^{\#os}$ recursive calls to $(fs'\ [\,]\ \ldots)$ after recursive expansion. In fact, not all these recursive calls are necessary for computing the final result, because the first equation (base case) of fs' shows that those recursive calls of $fs'\ [\,]\ vss\ least\ r$ will not contribute to the final result if

$$\#vss < least.$$

The base-case filter promotion [Chi90] says that the base case condition could be promoted to be a condition for the recursive calls, which is very helpful

in pruning unnecessary recursive calls. Applying the base-case filter promotion calculation gives the following program:

$$\begin{array}{lll} fs'\ []\ vss\ least\ r & = & \text{if } \#vss \geq least \text{ then } \{r\} \text{ else } \{\,\} \\ fs'\ (o:os)\ vss\ least\ r & = & (\text{if } \#vss \geq least \\ & & \quad \text{then } fs'\ os\ vss\ least\ r \text{ else } \{\,\}) \cup \\ & & (\text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\ & & \quad \text{then } fs'\ os\ ((o\ `elem`) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\ & & \quad \text{else } \{\,\}) \end{array}$$

and accordingly fs changes to

$$fs\ os\ vss\ least = \text{if } \#vss \geq least \text{ then } fs'\ os\ vss\ least\ [] \text{ else } \{\,\}.$$

Now propagating the condition of $\#vss \geq least$ backwards from the initial call of fs' to its recursive calls, we obtain

$$\begin{array}{lll} fs'\ []\ vss\ least\ r & = & \{r\} \\ fs'\ (o:os)\ vss\ least\ r & = & fs'\ os\ vss\ least\ r \text{ else } [] \cup \\ & & (\text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\ & & \quad \text{then } fs'\ os\ ((o\ `elem`) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\ & & \quad \text{else } \{\,\}) \end{array}$$

in which any recursive call $fs'\ os\ vss\ least\ r$ that does not meet the condition of $\#vss \geq least$ would be selectively pruned.

3.4 Tabulation

Although much improvement has been achieved through fusion, accumulation, and base-case filter promotion, there still remains a source of serious inefficiency because the inductive parameter *os* is traversed multiple times by fs'. We want to share some computation among all recursive calls to fs', by using the tabulation calculation [Bir80, CH95].

The purpose of our tabulation calculation is to exploit the relationship among recursive calls to fs' so that their computation could be shared. The difficulty in such tabulation is to determine which values should be tabulated. Now, taking a close look at the derived definition for fs'

$$\begin{array}{lll} fs'\ (o:os)\ \underline{vss}\ least\ \underline{r} & = & fs'\ os\ \underline{vss}\ least\ \underline{r} \cup \\ & & (\text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\ & & \quad \text{then } fs'\ os\ \underline{((o\ `elem`) \triangleleft vss)}\ least\ \underline{(r +\!\!+ [o])} \\ & & \quad \text{else } \{\,\}) \end{array}$$

reveals some dependency of the second and the fourth arguments of fs' among the left and the right recursive calls to fs', as indicated by the underlined parts. Moreover these two arguments will be used to produce the final result, according to the base case definition of fs'. This hints us to keep (memoize) all *necessary* intermediate results of the second and the fourth parameters:

$$(r_1, vss_1), (r_2, vss_2), \ldots.$$

According to base-case filter promotion, each element (r_i, vss_i) meets the *invariant* property

$$\#vss_i \geq least.$$

We could store all these pairs using a list. But a closer look at the second equation of fs' reveals that along with the extension of r, the corresponding number of vss decreases with each filtering. More precisely, for any two intermediate results of (r_i, vss_i) and (r_j, vss_j), $r_i \subseteq r_j$ implies $vss_i \subseteq vss_j$. This observation suggests us to organize all these pairs into a tree. To do so, we define the following tree data structure for memoization:

$$Tree = Node\ ([Int], [[Int]])\ [Tree].$$

In this tree, each node, tagged with a pair storing (r_i, vss_i), can have any number of children.

Now we apply the tabulation calculation to fs' by defining

$$\begin{aligned} tab\ os\ least\ (Node\ (r, vss)\ [\,]) &= fs'\ os\ vss\ least\ r \\ tab\ os\ least\ (Node\ (r, vss)\ ts) &= fs'\ os\ vss\ least\ r \cup \\ &\quad\ flattenMap\ (tab\ os\ least)\ ts \end{aligned}$$

where $flattenMap$ is defined by

$$flattenMap\ f = foldr\ (\cup)\ \{\,\}\ \circ\ map\ f.$$

Clearly fs' is a special case of tab:

$$fs'\ os\ vss\ least\ r = tab\ os\ least\ (Node\ (r, vss)\ [\,])$$

We hope to synthesize a new definition that defines tab inductively on os where os is traversed only once (it is now traversed by both fs' and tab). The general form for this purpose should be

$$\begin{aligned} tab\ [\,]\ least\ t &= select\ least\ t \\ tab\ (o : os)\ least\ t &= tab\ os\ least\ (add\ o\ least\ t) \end{aligned}$$

where $select$ and add are two newly introduced functions that are to be calculated. We can synthesize $select$ by induction on tree t. From

$$\begin{aligned} & tab\ [\,]\ least\ (Node\ (r, vss)\ [\,]) \\ = & \quad \{\ \text{def. of } tab\ \} \\ & fs'\ [\,]\ vss\ least\ r \\ = & \quad \{\ \text{def. of } fs'\ \} \\ & \{r\} \end{aligned}$$

and

$$\begin{aligned} & tab\ [\,]\ least\ (Node\ (r, vss)\ ts) \\ = & \quad \{\ \text{relation between } tab \text{ and } fs', \text{ the invariant tells: } \#vss \geq least\ \} \\ & fs'\ [\,]\ vss\ least\ r \cup flattenMap\ (tab\ [\,]\ least)\ ts \\ = & \quad \{\ \text{def. of } fs', \text{ and the above invariant } \} \\ & \{r\} \cup flattenMap\ (tab\ [\,]\ least)\ ts \\ = & \quad \{\ \text{relation between } tab \text{ and } select\ \} \\ & \{r\} \cup flattenMap\ (select\ least)\ ts \end{aligned}$$

we soon have

$$
\begin{array}{l}
select\ least\ (Node\ (r, vss)\ [\,]) = \{r\} \\
select\ least\ (Node\ (r, vss)\ ts) = \{r\}\ \cup\ flattenMap\ (select\ least)\ ts.
\end{array}
$$

The definition of *add* can be inferred in a similar fashion. For the base base:

$$
\begin{array}{ll}
 & tab\ (o : os)\ least\ (Node\ (r, vss)\ [\,]) \\
= & \quad \{\ \text{def. of } tab\ \} \\
 & fs'\ (o : os)\ vss\ least\ r \\
= & \quad \{\ \text{def. of } fs'\ \} \\
 & fs'\ os\ vss\ least\ r \cup \\
 & (\text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
 & \quad \text{then } fs'\ os\ ((o\ `elem`) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\
 & \quad \text{else } \{\,\}) \\
= & \quad \{\ \text{by } if \text{ property}\ \} \\
 & \text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
 & \quad \text{then } fs'\ os\ vss\ least\ r\ \cup\ fs'\ os\ ((o\ `elem`) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\
 & \quad \text{else } fs'\ os\ vss\ least\ r \\
= & \quad \{\ \text{relation between } tab \text{ and } fs'\ \} \\
 & \text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
 & \quad \text{then } tab\ os\ least\ (Node\ (r, vss)\ [Node\ ((r +\!\!+ [o]), (o\ `elem`) \triangleleft vss)\ [\,]]) \\
 & \quad \text{else } tab\ \ os\ least\ (Node\ (r, vss)\ [\,]) \\
= & \quad \{\ \text{by } if \text{ property}\ \} \\
 & tab\ os\ least \\
 & \quad (\text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
 & \quad\ \text{then } Node\ (r, vss)\ [Node\ ((r +\!\!+ [o]), (o\ `elem`) \triangleleft vss)\ [\,]] \\
 & \quad\ \text{else } Node\ (r, vss)\ [\,])
\end{array}
$$

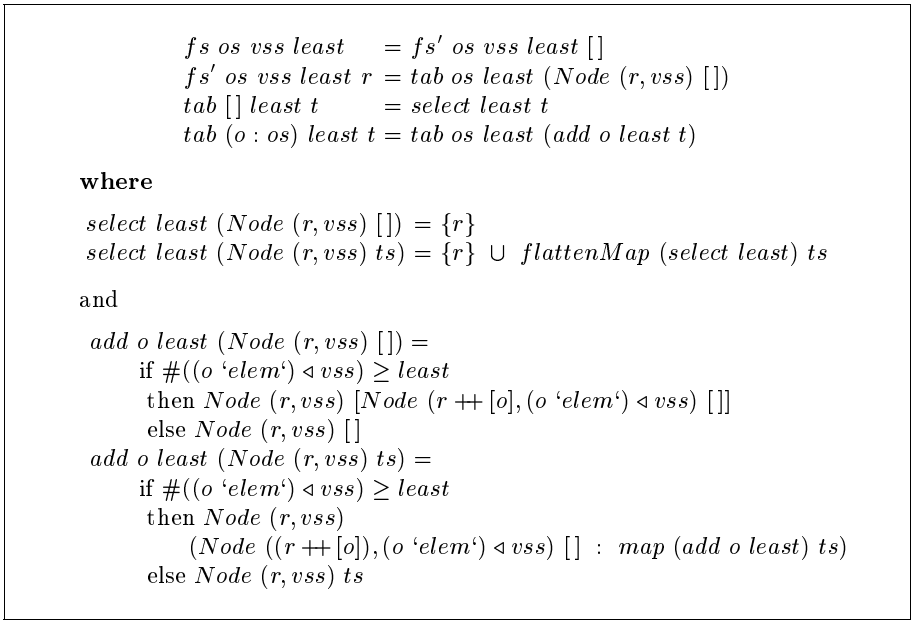
we thus get:

$$
\begin{array}{l}
add\ o\ least\ (Node\ (r, vss)\ [\,]) = \\
\quad \text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
\quad\ \text{then } Node\ (r, vss)\ [Node\ ((r +\!\!+ [o]), (o\ `elem`) \triangleleft vss)\ [\,]] \\
\quad\ \text{else } Node\ (r, vss)\ [\,].
\end{array}
$$

Similarly, for the inductive case, we can derive the following result, whose detailed derivation is omitted.

$$
\begin{array}{l}
add\ o\ least\ (Node\ (r, vss)\ ts) = \\
\quad \text{if } \#((o\ `elem`) \triangleleft vss) \geq least \\
\quad\ \text{then } Node\ (r, vss) \\
\quad\quad (Node\ ((r +\!\!+ [o]), (o\ `elem`) \triangleleft vss)\ [\,]\ :\ map\ (add\ o\ least)\ ts) \\
\quad\ \text{else } Node\ (r, vss)\ ts
\end{array}
$$

Comparing the two programs before and after tabulation calculation, we can see that the latter is more efficient in that it shares the computation for checking the invariant conditions; when an object o is added to the tree, it checks from the root and if it fails at a node, it does not check its descendants. Now putting all together, we get the final result in Figure 1.



$$
\begin{aligned}
&fs\ os\ vss\ least &&= fs'\ os\ vss\ least\ [\,]\\
&fs'\ os\ vss\ least\ r &&= tab\ os\ least\ (Node\ (r, vss)\ [\,])\\
&tab\ [\,]\ least\ t &&= select\ least\ t\\
&tab\ (o : os)\ least\ t &&= tab\ os\ least\ (add\ o\ least\ t)
\end{aligned}
$$

where

$$
\begin{aligned}
&select\ least\ (Node\ (r, vss)\ [\,]) = \{r\}\\
&select\ least\ (Node\ (r, vss)\ ts) = \{r\}\ \cup\ flattenMap\ (select\ least)\ ts
\end{aligned}
$$

and

$$
\begin{aligned}
&add\ o\ least\ (Node\ (r, vss)\ [\,]) =\\
&\quad \text{if } \#((o\ `elem`) \triangleleft vss) \geq least\\
&\quad\ \text{then } Node\ (r, vss)\ [Node\ (r +\!\!+ [o], (o\ `elem`) \triangleleft vss)\ [\,]]\\
&\quad\ \text{else } Node\ (r, vss)\ [\,]\\
&add\ o\ least\ (Node\ (r, vss)\ ts) =\\
&\quad \text{if } \#((o\ `elem`) \triangleleft vss) \geq least\\
&\quad\ \text{then } Node\ (r, vss)\\
&\qquad (Node\ ((r +\!\!+ [o]), (o\ `elem`) \triangleleft vss)\ [\,]\ :\ map\ (add\ o\ least)\ ts)\\
&\quad\ \text{else } Node\ (r, vss)\ ts
\end{aligned}
$$

Fig. 1. Our Final Program for Finding Frequent Sets

4 Discussion

We shall clarify three features of the derived algorithm, namely correctness, simplicity, efficiency and inherited parallelism, and highlight how to adapt the algorithm to practical use.

Correctness

The correctness follows directly from the basic property of program calculation. Our derived algorithm is *correct* with respect to the initial straightforward specification, because the whole derivation is done in a semantics-preserving manner. In contrast, the correctness of existing algorithms, well summarized in [Toi96], are often proved in an ad-hoc manner.

Simplicity

Our derived algorithm is surprisingly *simple*, compared to the existing algorithms which pass over the database many times and use complicated and costly manipulation (generating and pruning) of candidates of frequent sets. A major difference is that our algorithm traverse the database only once, object by object (i.e., item by item) rather than the traditional processing of transaction by transaction. Put it in another way, our algorithm traverses the database vertically while the traditional algorithms traverse the database horizontally, if we assume that the database is organized by transactions.

The vertical traversal of database comes naturally from our initial straightforward specification, where we were not at all concerned with efficiency and implementation details. In comparison, traditional algorithms were designed with an implicit assumption that database should be scanned horizontally, which we believe is not essential. We can preprocess the database to fit our algorithm by transposing it through a single pass. This preprocessing can be done in an efficient way even for a huge database saved in external storage (see [Knu97]). In fact, as such preprocessing need only be done once for a given transaction database, we can easily amortize its costs over many data mining runs for the discovery of interesting information/rules.

Efficiency

To see how efficient our algorithm is in practice, we shall not give a formal study of the cost. Such a study needs to take account of both the distribution as well as the size of data sample. Rather we use a *simple* experiment to compare our algorithm with an existing improved Apriori algorithm [MT96], one of the best algorithms used in the data mining community.

We start by considering the case of a small database which can be put in the memory. We tested three algorithms in Haskell: our initial specification program, the functional coding of the existing improved Apriori algorithm in [HLG$^+$99], and our final derived program. Our initial and final algorithms can be directly coded in Haskell by representing sets using lists.

The input sample data was extracted from the Richard Forsyth's zoological database, which is available in the UCI Repository of Machine Learning Databases [BM98]. It contains 17 objects (corresponding to 17 boolean attributes in the database) and 101 transactions (corresponding to 101 instances). We set the threshold to be 20 (20% of frequency), and did experiment with Glasgow Haskell Compiler and its profiling mechanism. The experimental result is as follows.

	total time (secs)	memory cells (mega bytes)
Our Initial Specification	131.2	484.1
An Apriori Algorithm	10.88	72.0
Our Final Algorithm	0.44	2.5

It shows that our final algorithm has been dramatically improved comparing to our initial one, and that it is also much more efficient than the functional coding of an existing algorithm (about 20 times faster but using just 1/30 of memory cells).

What if the database is so huge that only part of database can be read into memory at one time? Except for the preprocessing of the database to match our algorithm (to be done just once as discussed above), our algorithm can deal with the partitioning of database very well. If the database has N objects (items), our algorithm allow it to be partitioned into N smaller sections. We only require that each of these sections be read into memory, one at a time, which poses no problem practically.

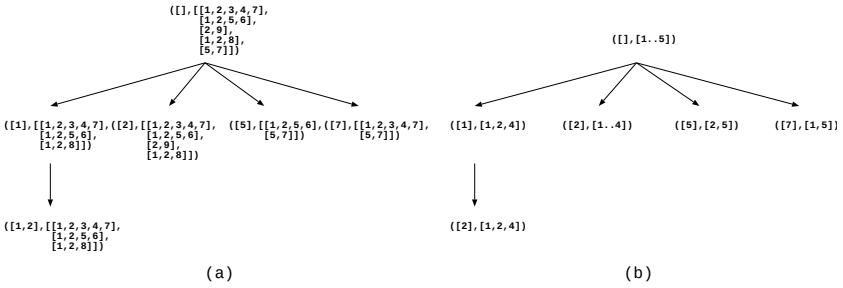


Fig. 2. Tree Structure for Tabulation

Parallelism

Our algorithm is quite suitable for parallel computation, which can be briefly explained as follows. Suppose that we have objects from 1 to N, and M processors of $P_1, \ldots, P_M$. We can decompose the objects into M groups, say 1 to N/M, $N/M + 1$ to $2N/M$, ..., and use P_i to compute the tabulation tree for items from $(i-1)N/M+1$ to iN/M. Certainty all the processors can do this in parallel. After that, we may propagate information of the single-item frequent sets from processor P_{i+1} to P_i for all i, to see whether these single-item frequent sets in P_{i+1} could be merged with frequent sets computed in P_i. Note that this parallel algorithm can be obtained directly from the sequential program *tab* in Figure 1 by parallelization calculation [HTC98], which is omitted here.

Practical Issues

The derived algorithm can be used practically to win over the existing algorithms. To be able to compare our results more convincingly with those in data mining field, we are mapping the algorithm to a C program and testing it on the popular benchmark of sample database. The detailed results will be summarized in another paper. Here, we only highlight one practical consideration.

A crucial aspect in practical implementation of the derived algorithm is the design of an efficient data structure to represent the tabulation tree to keep memory usage down. In fact, we can refine the current structure of tabulation tree to use less space. Notice that each node of the tabulation tree is attached with a pair (r, vss) where r represents a frequent set, and vss represents all the visits that contain r. Naive implementation would take much space. To be concrete, consider the example given in the beginning of Section 2. After traversing all objects from 1 to 11, we get the tree (a) in Figure 2. The vss part in each node consumes much space. In fact, it is not necessary to store the detailed visit content in each node. Instead, it is sufficient to store a list of indices to the visits, as shown in tree (b) in Figure 2. Practically, the number of indices in each node is not so big except for the root where we use the range

notation to represent it cheaply, and this would become smaller with each step down from parent to its children.

We can do further in many ways to reduce the size of the tabulation tree. (1) In preprocessing phase, we may sort the objects of the database by decreasing frequency, which should allow subrange notation of indices to be used in a maximized fashion. (2) If the size of *vss* is within twice of the threshold *least* at a particular node, we may keep negative information at the children nodes, as these lists would be shorter than the threshold. (3) As nodes for 1-itemset take the most memory and these should perhaps be kept offline in virtual memory and be paged in when required.

5 Conclusion

In this paper, we have addressed a practical application of program calculation of functional programs by deriving novel algorithms that are also practically fast. We have chosen an important subproblem of finding frequent sets as our target. This problem is of practical interest, and have been extensively researched by the data mining community in the last six years. Many researchers have devoted much time and energy to discover clever and fast algorithms. By program calculation of functional programs, we have successfully obtained a completely new algorithm that is also practically fast.

Our derivation of a new frequent set algorithm did not depend on new tricks. Instead, it is carried out using a sequence of standard calculation techniques such as fusion, accumulation, filter promotion and tabulation. These calculation techniques are quite well-known in the functional programming community.

Acknowledgments

This paper owes much to the thoughtful and inspiring discussions with David Skillicorn, who argued that program calculation should be useful in derivation of data mining algorithms. He kindly explained to the first author the problem as well as some existing algorithms. We would also like to thank Christoph Armin Herrmann who gave us his functional coding of an existing (improved) Apriori algorithm, and help test our Haskell code with his HDC system.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *1993 International Conference on Management of Data (SIGMOD'93)*, pages 207–216, May 1993.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[Bir80] R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

[Bir84] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[BM98] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *1997 International Conference on Management of Data (SIGMOD'97)*, pages 255–264, AZ, USA, 1997. ACM Press.

[CH95] W. Chin and M. Hagiya. A transformation method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115, 1995.

[Chi90] W.N. Chin. *Automatic Methods for Program Transformation.* Phd thesis, Department of Computing, Imperial College of Science, Technology and Medicone, University of London, May 1990.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HLG+99] C. Herrmann, C. Lengauer, R. Gunz, J. Laitenberger, and C. Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultat fur Mathematik und Informatik, Universitat Passau, May 1999.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[Knu97] D. Knuth. *The Art of Computer Programming: Volume 3 / Sorting and Searching.* Addison-Wesley, Longman, 1997. Second Edition.

[LK98] D. Lin and Z. Kedem. Princer Search: A new algorithm for discovering the maximum frequent set. In *VI Intl. Conference on Extending Database Technology*, Valencia, Spain, March 1998.

[MT96] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 189 – 194, Portland, Oregon, August 1996. AAAI Press.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[Toi96] H. Toivonen. *Discovery of Frequent Patterns in Large Data Collections.* Ph.D thesis, Department of Computer Science, University of Helsinki, 1996.

This article was processed using the LaTeX macro package with LLNCS style