On Pattern-Based Programming towards the Discovery of Frequent Patterns

Kittisak Kerdprasop, and Nittaya Kerdprasop

Abstract—The problem of frequent pattern discovery is defined as the process of searching for patterns such as sets of features or items that appear in data frequently. Finding such frequent patterns has become an important data mining task because it reveals associations, correlations, and many other interesting relationships hidden in a database. Most of the proposed frequent pattern mining algorithms have been implemented with imperative programming languages. Such paradigm is inefficient when set of patterns is large and the frequent pattern is long. We suggest a high-level declarative style of programming apply to the problem of frequent pattern discovery. We consider two languages: Haskell and Prolog. Our intuitive idea is that the problem of finding frequent patterns should be efficiently and concisely implemented via a declarative paradigm since pattern matching is a fundamental feature supported by most functional languages and Prolog. Our frequent pattern mining implementation using the Haskell and Prolog languages confirms our hypothesis about conciseness of the program. The comparative performance studies on line-of-code, speed and memory usage of declarative versus imperative programming have been reported in the paper.

Keywords—Frequent pattern mining, functional programming, pattern matching, logic programming.

I. INTRODUCTION

THE problem of frequent pattern discovery is an important problem in various fields such as data mining, business intelligence, pattern mining. Frequent pattern discovery refers to an attempt to find regularities or common relationships frequently occurred in a database. A set of market basket transactions [1], [2] is a common database used in frequent pattern analysis. A database is in a table format (as shown in Fig. 1). Each row is a transaction, identified by a transaction identifier *TID*. A transaction contains a set *I* of items bought by a customer.

Kittisak Kerdprasop is a director of the Data Engineering and Knowledge Discovery (DEKD) research unit, School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224349; fax: +66-44-224602; e-mail: kerdpras@ sut.ac.th, KittisakThailand@gmail.com).

Nittaya Kerdprasop is a principal researcher of DEKD research unit and an associate professor at the School of Computer Engineering, Suranaree University of Technology, 111 University Ave., Nakhon Ratchasima 30000, Thailand (e-mail: nittaya@sut.ac.th, nittaya.k@gmail.com).

TID	Items
1	{Cereal, Milk}
2	{Beer, Cereal, Diaper, Egg}
3	{Beer, Diaper, Milk}
4	{Beer, Cereal, Diaper, Milk}
5	{Diaper, Milk}

Fig. 1 An example of transactional database

Most of the proposed frequent pattern mining algorithms have been implemented with imperative programming languages such as C, C++, Java. The imperative paradigm is inefficient when the size of pattern set is large and the pattern is long [9],[10]. We thus propose to switch the programming paradigm towards the declarative programming, that is, functional and logic programming. These two programming languages provide big advantage of pattern-based computation since pattern matching is fully supported by both languages.

The rest of this paper is organized as follows. The next section presents preliminaries of the frequent pattern discovery problem and the basic algorithm in which our paper is based upon. Section 3 discusses the pattern matching features in Haskell and Prolog. Section 4 presents the comparison of declarative programming style against the imperative style. Section 5 explains our implementations including excerpts of source codes of frequent pattern discovery in Haskell and Prolog. Section 6 shows the experimental results regarding speed and memory usage of imperative paradigm versus declarative paradigm. Section 7 concludes the paper.

II. FREQUENT PATTERN DISCOVERY

The problem of frequent discovery is defined as [1], [2], [5], [6] the search for recurring relationships or correlations between items in a database. Let $I = \{i_1, i_2, i_3, ..., i_m\}$ be a set of *m* items and $DB = \{ T_1, T_2, T_3, ..., T_n \}$ be a transactional database of *n* transactions and each transaction contains items in *I*. A *pattern* is a set of items that occur in a transaction. The number of items in a pattern is called the length of the pattern. the pattern such as {Beer, Cereal, Diaper} is thus o pattern of length three or a 3-item pattern.

To search for all valid patterns of length 1 up to *m* in large database is computational expensive. It can be seen in Fig. 2 that a transactional database containing different combinations of five items (I = {Beer(B), Cereal(C), Diaper(D), Milk (K), Egg(E)})

can generate a search space of $2^5 - 1 = 31$ possible patterns. Thus, for a set I of m different items, the search space for all distinct patterns can be as huge as $2^m - 1$.
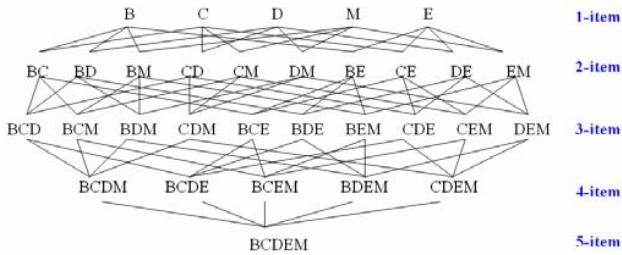

Fig. 2 A search space for finding patterns from the set of five items

To reduce the size of the search space, the *support* measurement has been introduced [1],[2]. The support $s(P)$ of a pattern P is defined as a number of transactions in DB containing P. Thus, $s(P) = |\{T \mid T \in \text{DB}, \ P \subseteq T \}|$.

A pattern P is called *frequent pattern* if the support value of P is not less than a predefined minimum support threshold *minS*. It is the *minS* constraints that help reducing the computational complexity of frequent pattern generation. Suppose we specify $minS = 2/5 = 40\%$ on a set of transactions shown in Fig. 1, then the pattern {Egg} is infrequent and so do all the set of patterns having Egg(E) as their member. All the infrequent patterns can be pruned (as shown in Fig. 3) to reduce the search space.
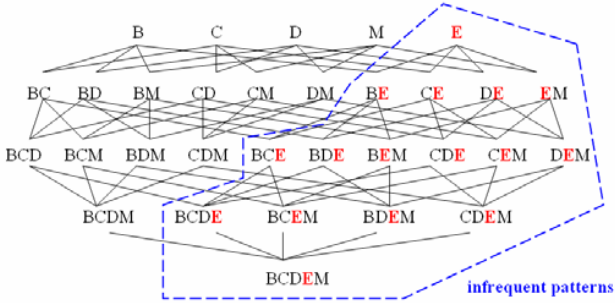

Fig. 3 A pruning of all patterns that contain an infrequent item E

The pruning strategy shown in Fig. 3 is called an anti-monotone property and is applied as a basis for searching frequent patterns in algorithm Apriori [1],[2]. The pseudocode in Fig. 4 sketches the outline of the algorithm.

We propose that frequent patterns can be mined efficiently using high-level programming languages such as Haskell and Prolog that provides a full support for pattern matching functionality.

$P_1 = \{x \mid x \text{ is an item in } I \text{ and } s(P) \geq minS \}$ // *1-item pattern*
For (k=1; $P_k \neq \varnothing$; k++) do
   $C_{k+1} = Generate\_candidate(P_k)$
   For each $T_i \in$ DB do

     Increment the count c of all candidates in C_{k+1}
       that are contained in T_i
   $P_k = \{c \mid c \in C_k \text{ and c.count} \geq minS\}$
 Return $\cup_k P_k$ //return all sets of frequent patterns

Generate_candidate (P_{i-1})
 $C_i = \varnothing$ // initialize candidate frequent pattern set as empty
For each pattern J in P_{i-1} do
   For each pattern K in P_{i-1} and $K \neq J$ do
     If i-2 of the elements in J and K are equal then
      If all subsets of $\{K \cup J\}$ are in P_{i-1} then
        $C_i = C_i \cup \{K \cup J\}$
Return C_i

Fig. 4 Apriori algorithm to generate frequent patterns

III. PATTERN MATCHING IN HASKELL AND PROLOG

A problem in frequent pattern discovery is to determine how often a candidate pattern occurs. A pattern is a set of items co-occurrence across a database. Given a candidate pattern, the task of pattern matching is to search for its frequency looking for the patterns that are frequent enough. The outcome of this search is frequent patterns that suggest strong co-occurrence relationships between items in the dataset.

The search for patterns of interest can be efficiently programmed using the Haskell language. Haskell has evolved as a strongly typed, lazy, pure functional language since 1987 [7], [8], [11]. The language is named after the mathematician Haskell B. Curry whose work on lambda calculus provides the basis for most functional languages. A program in functional languages is made up of a series of function definitions. The evaluation of a program is simply the evaluation of functions. Haskell is a pure functional language because functions in Haskell have no side effect, i.e. given the same arguments, the function always produces the same result. As an example, we can define a simple function to square an integer as follows:

```
square :: Int -> Int        -- type declaration
square x = x * x            -- function definition
```

The first line of the definition declares the type of the thing being defined; Haskell is a strongly typed language. This states that square is a function taking one integer argument (the first Int) and returning an integer value (the second Int). The arrow symbol denotes mapping from an argument to a result and the symbol "::" can be read "has type". The statement or phrase following the symbol "--" is a comment. The second line gives the definition of function square, i.e. given an integer x, the function returns the value of x*x. To apply the function, we provide the function an actual argument such as square 5 and the result 25 can be expected.

Pattern matching is one of the most powerful features of Haskell. Defining functions by specifying argument patterns is a common practice in programming with Haskell. As an illustration, consider the following example:

```
fib :: Int -> Int       -- a function takes one Int
                        -- and returns an Int

fib 0 = 0               -- pattern 1: argument is 0
fib 1 = 1               -- pattern 2: argument is 1
```

fib n = fib (n-2) + fib (n-1)   -- *pattern 3: argument is Int*
                                 --   *other than 0 and 1*

The function fib returns the nth number in the Fibonacci sequence. The left hand sides of function definitions contain patterns such as 0, 1, n. When applying a function these patterns are matched against actual parameters. If the match succeeds, the right hand side is evaluated to produce a result. If it fails, the next definition is tried. If all matches fail, an error is returned.

Pattern matching is a language feature commonly used with a list data structure. For instance, [1, 2, 3] is a list containing three integers. It can also be written as 1:2:3:[],where [] represents an empty list and ":" is a list constructor. The following example defines length function to count the number of elements in a list.

```
length :: [Int] -> Int
        -- This function takes a list of Int as its argument and
        -- returns the number of elements in the list

length [ ] = 0        -- pattern 1: length of an empty list is 0

length (x:xs) = 1 + length xs
                -- pattern 2: length of a list whose first
                -- element is called x and remainder is
                -- called xs is 1 plus the length of xs
```

The pattern [] is defined to match the case of an empty list argument. The pattern x:xs will successfully match a list with at least one element, i.e. xs can be a list of zero or more elements.

In Prolog, the feature of pattern matching can be defined through the use of arguments. For example, the following program [4] demonstrates the length function (in Prolog it is called predicate instead of function) to count the number of elements in a list. Last argument is normally a place holder for an output. Variables in Prolog start with an uppercase letter such as Xs, L, X. Each statement in Prolog is called a clause and every statement ends with period. The symbol ',' is a logical connective AND. The symbol ':-' is an implication and it may be pronounced as 'if'. Thus the last statement of length predicate may be read as "length of the list (X|Xs) is L is length of the list (Xs) is M and L is M+1."

```
length([ ], 0).        -- pattern 1: length of an empty list is 0

length( (X|Xs), L) :-  length( Xs, M), L is M+1.

        -- pattern 2: length of a list whose first
        -- element is X and remainder is Xs is 1+ length of xs
```

Programs to square an integer and to find the nth number in the Fibonacci sequence in Prolog can be done through a pattern as well. The Prolog codes are illustrated as follows.

```
% square function in Prolog

square(X, Y) :- Y is X*X.


% Fibonacci function in Prolog

fib(0, 0).
fib(1, 1).
fib(N, F) :-  N > 1,  N1 is N-1, N2 is N-2,
              fib(N1, F1), fib(N2, F2),
```

F is F1 + F2.

IV. DECLARATIVE VERSUS IMPERATIVE PROGRAMMING

In declarative languages such as Haskell and Prolog, programs are sets of definitions and recursion is the main control structure of the program computation. In imperative languages (also called procedural languages) such as C and Java, programs are sequences of instructions and loops are the main control structure. A functional programming language like Haskell is a declarative language in which programs are sets of *function* definitions. A logic programming language like Prolog is a declarative language in which programs are sets of *predicate* definitions. Predicates are truce or false when applied to an object or set of objects, while functions return a result. A predicate typically has one more argument (to serve as a returned value) than the equivalent function. Either function or predicate definitions, each definition has a dual meaning: (1) it describes what is the case, and (2) it describes the way to compute something.

Declarative languages are mathematically sound. It is easy to prove that a declarative program meets its specification which is a very important requirement in software industry. Declarative style makes a program better engineered, that is, easier to debug, easier to maintain and modify, and easier for other programmers to understand. The following examples of coding quick sort in C, Haskell and Prolog verify the previous statement.

Haskell
```
    sort []     = []
    sort (x:xs) = sort [y | y<-xs, y<x ] ++ [x] ++
                  sort [y | y<-xs, y>=x]
```
Prolog
```
    qs([],[]).
    qs([ X | Xs]) :- part(X, Xs, Littles, Bigs),
                 qs( Littles, Ls),
                 qs( Bigs, Bs),
                 append(Ls, [X| Bs], Ys).

    part(_, [], [], []).
    part(X, [Y|Xs], [Y|Ls], Bs) :- X>Y, part(X, Xs, Ls, Bs).
    part(X, [Y|Xs], Ls, [Y|Bs]) :- X=<Y, part(X, Xs, Ls, Bs).
```

C
```
    int partition(int y[], int f, int l);
    void quicksort(int x[], int first, int last) {
        int pivIndex = 0;
        if(first < last) {   pivIndex = partition(x,first, last);
                   quicksort(x,first,(pivIndex-1));
                   quicksort(x,(pivIndex+1),last);
                   }   }
     int partition(int y[], int f, int l) {
        int up,down,temp;   int cc;    int piv = y[f];
        up = f;
        down = l;
        do { while (y[up] <= piv && up < l) {up++;}
            while (y[down] > piv  ) {down--; }
          if (up < down ) {   temp = y[up];
                         y[up] = y[down];
                         y[down] = temp;    }
        } while (down > up);
        temp = piv;    y[f] = y[down];    y[down] = piv;
```

```
      return down;
   }
```

V. IMPLEMENTATION

We implement Apriori algorithm [1], [2] using Haskell and Prolog languages as shown some parts of the programs in Figs. 5 and 6, respectively. In Haskell, each item is represented by the item identifier which is an integer. Thus, a set of patterns (patternset) is denoted as a set of Int declared in the first line of our Haskell code. The function sumi is defined to count the number of occurrence of each element in patternSet. Functions listC and listC' perform the task of enumerating candidate frequent patternSet. Only patternSet that satisfy the *minS* threshold are reported from the functions listL and listL' as frequent patternSet. The complete implementation of frequent pattern discovery using Haskell functional language takes only 37 lines of code.

Prolog implementation to discover frequent patterns contains around 58 lines of code. Its conciseness is approximately the same as Haskell codes. In Prolog, data type definition is not necessary because Prolog is weakly typed. Thus, pattern matching in Prolog is more general than that of Haskell. We use the set union to construct candidate patterns of length two or more as in Haskell implementation. However, the concept of computation in Prolog which is built upon logic is totally different from Haskell which is on the basis of mathematical function.

```
patternSet :: [Set Int]
patternSet =[Set.singleton x | x<-[1..9]]
sumi::Set Int->[Set Int]->Int
sumi s [] =0
sumi s (y:ys) |(Set.isSubsetOf s y)= 1+(sumi s ys)
                           |otherwise = (sumi s ys)
listC ::Int->[(Set Int,Int)]
listC 1=[let n=(sumi s dataB) in (s,n)  |s<-patternSet]
listC n=[let n=(sumi s dataB) in (s,n)  |s<-
                              Set.toList(listC' n)]
listC' :: Int->Set(Set Int)
listC' 2=Set.fromList
    [(Set.union x y) |x<-(listL' 1),y<-(listL' 1),x/=y]
listC' n=Set.fromList
   [(Set.union  x  y)  |x<-(listL'  (n-1)),  y<-(listL'  (n-1)),
     x/=y, (Set.size(Set.union x y))==n]
listL ::Int->[(Set Int,Int)]
listL n=[(x,y)|(x,y)<-listC n, y>=minS]
listL'::Int->[Set Int]
listL' n =[x|(x,_)<-listL n]
```

Fig 5 Frequent pattern discovery implemented with Haskell

```
r1:- n(X), cL1(X).
r2(X):- cC2(X).
r3(X):- cC3(X).
l:- listing.
```

```
c:- clear.
clear:-retractall(l1(_)),  retractall(c1(_)),
     retractall(c2(_)) , retractall(l2(_)),
     retractall(c3(_)), retractall(l3(_)).

% Create L1
cL1([]).
cL1([H|T]) :-  findall(X, f([H],X), L), length(L, Len),
          Len >= 2 , !,
          cL1(T),
          assert(l1(([H], Len)))
          ;
          cL1(T).
% Create C2, L2
cC2(X) :- l1((X,_)), l1((X2,_)),
        X\==X2, write(X-X2),
        union(X, X2, Res),
        assert(c2((Res))), retract(l1((X,_))),  nl.
 crC2(L) :- findall(X,c2(X),L).

cL2([]).
cL2([H|T]) :- findall(X,f(H,X),L), length(L,Len),
          Len >= 2 ,!,
          cL2(T),
          assert(l2((H,Len)))
          ;
          cL2(T).
cC3(X) :- l2((X,_)),l2((X2,_)),
        X\==X2, write(X-X2),
        union(X, X2, Res),
        assert(c3((Res))),  retract(l2((X,_))), nl.
crC3(L):- findall(X,c3(X),L).
cL3([]).
cL3([H|T]) :- findall(X,f(H,X),L), length(L,Len),
          Len >= 2 ,!,
          cL3(T),
          assert(l3((H,Len)))
          ;
          cL3(T).
f(H, X) :-  item(X), subset(H, X).
```

Fig. 6 Frequent pattern discovery implemented with Prolog

VI. PERFORMANCE STUDIES

We comparatively study the performance of our implementations of frequent pattern discovery using Haskell and Prolog versus C and Java (source codes C and Java implementations are taken from [3]). All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD. We select four datasets

from the UCI Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html) to test the speed and memory usage of the programs. The details of selected datasets are summarized in Table I. The frequent pattern discovery programs have been tested on each dataset with various *minS* values. Performance comparisons of declarative (Haskell and Prolog) and imperative (C and Java) implementations on four datasets are shown in Figs. 7 and 8.

TABLE I
DATASET CHARACTERISTICS

Dataset	File size	# Transactions	# Items
Vote	13.2 KB	300	17
Chess	237 KB	2,130	37
DNA	252 KB	2,000	61
Mushroom	916 KB	5,416	23

The comparison results of run time and memory usage using different styles of programming are shown in Figs. 6 and 7, respectively. It can be noticed from the experimental results that on a speed comparison C implementation are the fastest, Haskell comes second following by Prolog and Java. On the memory usage comparison the ordering is the same as those on the speed comparison. However, it can be noticed from the results that the degree of difference is insignificant. When taking into consideration the length of the source codes, Haskell: 37 lines, Prolog: 58 lines, C: 352 lines, Java: 663 lines, the declarative style of coding absolutely consumes less effort and development time than the coding with imperative style.

VII. CONCLUSIONS AND DISCUSSION

Frequent pattern discovery is one major problem in the areas of data mining and business intelligence. The problem concerns finding frequent patterns hidden in a large database. Frequent patterns are patterns such as set of items that appear in data frequently. Finding such frequent patterns has become an important data mining task because it reveals associations, correlations, and many other interesting relationships among items in the transactional databases.

The idea to discover an association of items frequently co-occur was first proposed in 1993 by R. Agrawal, T. Imielinski, and A. Swami and the well known Apriori algorithm was proposed by R. Agrawal and A. Swami in 1994. Since then many variations of Apriori have been proposed. Most algorithms are implemented with imperative programming languages such as C, C++, Java. We, on the other hand, suggest that the problem of frequent pattern discovery can be efficiently and concisely implemented with high-level declarative languages such as Haskell and Prolog.
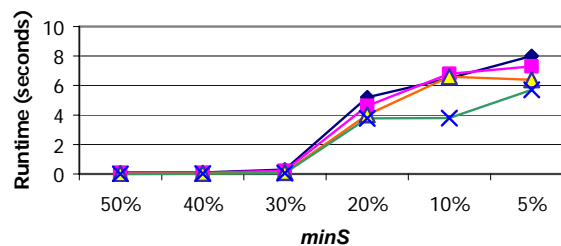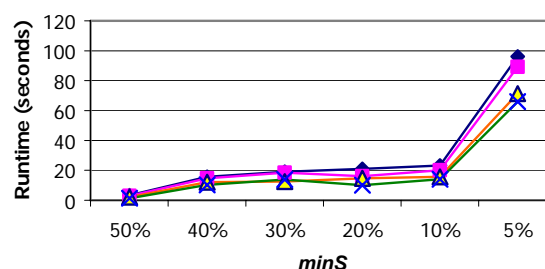

(a) Vote data
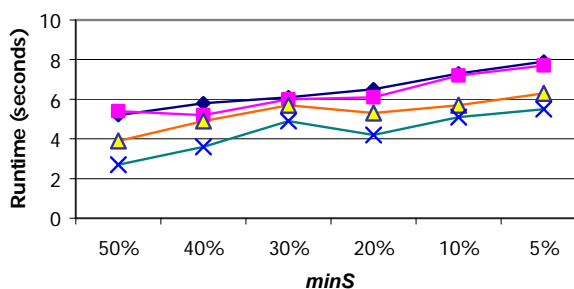
(b) Chess data
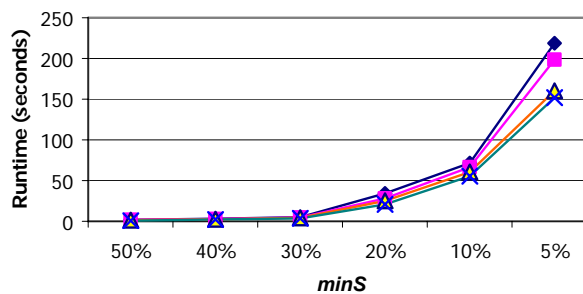
(c) DNA data

(d) Mushroom data

Fig. 7 Speed comparison of different programming styles

Coding in declarative style takes less effort because pattern matching is a fundamental feature supported by functional and logic languages. The implementations of Apriori algorithm using Haskell and Prolog confirm our hypothesis about conciseness of the program. The performance studies also support our intuition on efficiency because our implementations are not significantly less efficient than the C and Java implementations in terms of speed and memory usage.

This preliminary study supports our belief regarding declarative programming paradigm towards frequent pattern discovery. We focus our future research on the design of data organization to optimize the speed and storage requirement. We also consider the extension of Apriori in the course of concurrency to improve its performance.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1993, pp. 207–216.
[2] R. Agrawal and R. Srikant, "Fast algorithm for mining association rules," in *Proc. Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.
[3] C. Borgelt, "Frequent item sets miner for FIMI 2003," 2003. http://www.borgelt.net/software.html
[4] I. Bratko, *Prolog Programming for Artificial Intelligence* (3rd ed.), Pearson, 2001.
[5] A. Ceglar and J. Roddick, "Association mining," *ACM Computing Surveys*, vol. 38, no.2, 2006.
[6] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* (2nd ed.), Morgan Kaufmann, 2006.
[7] P. Hudak, J. Fasel, and J. Peterson, "A gentle introduction to Haskell," Yale University, *Technical Report Yale U/DCS/RR-901*, 1996.
[8] P. Jones and J. Hughes (eds.), *Standard Libraries for the Haskell 98 Programming Languages*. Available: http://www.haskell.org/library/.
[9] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging vertical mining of large databases," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2000, pp. 22–33.
[10] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison Wesley, 2005.
[11] S. Thompson, *Haskell: The Craft of Functional Programming* (2nd ed.), Addison Wesley, 1999.

Kittisak Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. He received his bachelor degree in Mathematics from Srinakarinwirot University, Thailand, in 1986, master degree in computer science from the Prince of Songkla University, Thailand, in 1991 and doctoral degree in computer science from Nova Southeastern University, USA, in 1999. His current research includes Data mining, Artificial Intelligence, Functional Programming, Computational Statistics.

Nittaya Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science from Nova Southeastern University, USA, in 1999. She is a member of ACM and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, AI, Logic Programming, Deductive and Active Databases.
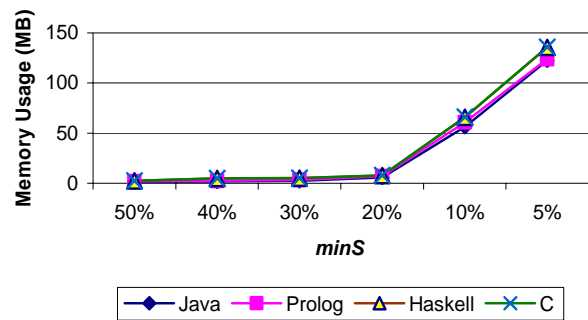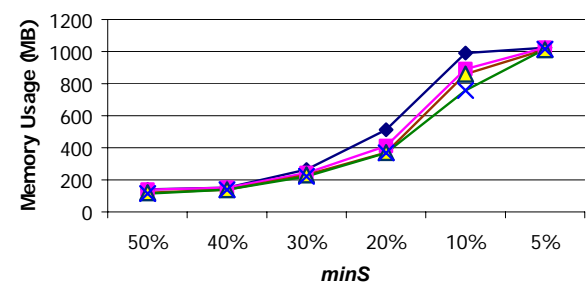
(a) Vote data
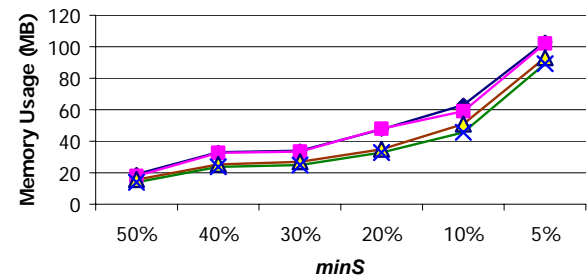
(b) Chess data
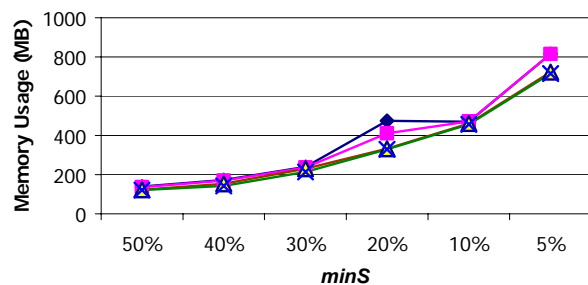
(c) DNA data

(d) Mushroom data

Fig. 8 Space comparison of different programming styles