algorithmic solutions", *Computer Methods and Programs in Biomedicine*, 83, 2007, pp.39-51.

[6] S. Ceri, R. Cochrane, and J. Widom, "Practical applications of triggers and constraints: Successes and lingering issues", *Proc. 26th VLDB*, 2000, pp.254-262.

[7] R. Correia, F. Kon, and R. Kon, "Borboleta: A mobile telehealth system for primary homecare", *Proc. ACM Symp. on Applied Computing*, 2008, pp.1343-1347.

[8] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for itemset mining", *Proc. KDD*, 2008, pp.204-212.

[9] S. Ghazavi and T. Liao, "Medical data mining by fuzzy modeling with selected features", *Artificial Intelligence in Medicine*, 43(3), 2008, pp.195-206.

[10] M. Huang, M. Chen, and S. Lee, "Integrating data mining with case-based reasoning for chronic diseases prognosis and diagnosis", *Expert Systems with Applications*, 32, 2007, pp.856-867.

[11] N. Hulse, G. Fiol, R. Bradshaw, L. Roemer, and R. Rocha, "Towards an on-demand peer feedback system for a clinical knowledge base: A case study with order sets", *J Biomedical Informatics*, 41, 2008, pp.152-164.

[12] E. Kretschmann, W. Fleischmann, and R. Apweiler, "Automatic rule generation for protein annotation with the C4.5 data mining algorithm applied on SWISS-PROT" *Bioinformatics*, 17(10), 2001, pp.920-926.

[13] D. Lee, W. Mao, H. Chiu, and W. Chu, "Designing triggers with trigger-by-example", *Knowledge and Information System*, 7, pp.110-134.

[14] F. Lin, S. Chou, S. Pan, and Y. Chen, "Mining time dependency patterns in clinical pathways", *Int. J Medical Informatics*, 62(1), 2001, pp.11-25.

[15] G. Nadathur and D. Miller, "Higher-order Horn clauses", *J ACM*, 37, 1990, pp.777-814.

[16] D. Nguyen, T. Ho, and S. Kawasaki, "Knowledge visualization in hepatitis study", *Proc. Asia-Pacific Symp. on Information Visualization*, 2006, pp.59-62.

[17] G. Noren, A. Bate, J. Hopstadius, K. Star, and I. Edwards, "Temporal pattern discovery for trends and transient effects: Its application to patient records", *Proc. KDD*, 2008, pp.963-971.

[18] S. Palaniappan and C. Ling, "Clinical decision support using OLAP with data mining", *Int. J Computer Science and Network Security*, 8(9), 2008, pp.290-296.

[19] J. Roddick, P. Fule, and W. Graco, "Exploratory medical knowledge discovery: experiences and issues", *ACM SIGKDD Explorations Newsletter*, 5(1), 2003, pp.94-99.

[20] J. Roddick, M. Spiliopoulou, D. Lister, and A. Ceglar, "Higher order mining", *ACM SIGKDD Explorations Newsletter*, 10(1), 2008, pp.5-17.

[21] T. Sahama and P. Croll, "A data warehouse architecture for clinical data warehousing", *Proc. 12th Australasian Symp. on ACSW Frontiers*, 2007, pp.227-232.

[22] A. Shillabeer and J. Roddick, "Establishing a lineage for medical knowledge discovery", *Proc. 6th Australasian Conf. on Data Mining and Analytics*, 2007, pp.29-37.

[23] A. Silva, P. Cortez, M. Santos, L. Gomes, and J. Neves, "Rating organ failure via adverse events using data mining in the intensive care unit", *Artificial Intelligence in Medicine*, 43(3), 2008, pp.179-193.

[24] J. Thongkam, G. Xu, Y. Zhang, and F. Huang, "Breast cancer survivability via AdaBoost algorithms", *Proc. 2nd Australasian Workshop on Health Data and Knowledge Management*, 2008, pp.55-64.

[25] K. Truemper, *Design of logic-based intelligent systems*, John Wiley & Sons, New Jersey, 2004.

[26] Z. Zhuang, L. Churilov, and F. Burstein, "Combining data mining and case-based reasoning for intelligent decision support for pathology ordering by general practitioners", European J Operational Research, 195(3), 2009, pp.662-675.

A DECLARATIVE PROGRAMMING PARADIGM AND THE DEVELOPMENT OF KNOWLEDGE MINING AGENTS

Nittaya Kerdprasop and Kittisak Kerdprasop
Data Engineering and Knowledge Discovery (DEKD) Research Unit,
School of Computer Engineering, Suranaree University of Technology,
Nakhon Ratchasima, Thailand

ABSTRACT

Agent is a conceptual entity designed to solve a complex problem. It differs from other software design concepts with its special capabilities of acting autonomously, adapting to changing circumstances, and communicating with other agents through high-level interactions. The significance of the agent-based approach in data mining, knowledge discovery, and Web intelligence has been realized by many researchers over the past decade. Several agent-based data mining tools have been developed. Most of them were implemented with imperative languages such as C and Java. We propose the agent model that has been implemented with a more powerful programming paradigm using declarative languages such as Haskell and Prolog. The advantages of these languages are their advancement in program structures, pattern matching and reasoning features, including higher order computation and meta-level programming. These language features are essential in developing intelligent agents. Even though the major drawback of most declarative languages is their computation speed, we have shown via experimental results that the percentage of speed decrease is insignificant comparing to imperative language implementation.

KEYWORDS

Knowledge mining agents, machine intelligence.

1. INTRODUCTION

Agents are key players in most current intelligent systems. According to Russell and Norvig [1995], agent is an entity (either a computer or a human) that perceives and acts in a particular environment. It is also defined [Wooldridge, 1997; Wooldridge and Jennings, 1995] as a computer system designed to work in some environment and has the capability to act autonomously in order to meet its designed goals. In complex and dynamic environments, multiagents are often utilized as a collaborative group of performers. A multiagent system [Weiss, 1999] is a group of entities working together to perform tasks that are beyond the individual capabilities of each entity. Agents may co-exist on a single processor, or they may be physically separated to perform activities on their own and build a community through communication. Intelligent agents [Wooldridge, 2002] employ additional capabilities of goal-directed task accomplishment, response due to changes in their environments, ability to interact with other agents, and learning to improve performances as they perform their assigned tasks.

To achieve intelligence, agents utilize several artificial intelligence techniques such as machine learning, inductive and deductive reasoning. On the contrary, intelligent agent technology can play an important role in the design and development of knowledge discovery, or data mining, systems. Knowledge discovery is the process of identifying valid, novel, potential useful and understandable patterns in data that may be distributed and heterogeneous in terms of content and structures [Fayyad et al., 1995; Han and Kamber, 2006]. This complex discovery process involves several phases including data selection, data preprocessing, data transformation, data analysis (or mining), interpretation and evaluation. These phases are iterative and adaptive in their nature, therefore it is a good setting for the application of intelligent agent technology. The ability of an agent to communicate, cooperate, and coordinate with other agents in multiagent system benefits the design of knowledge discovery tools to locate and mine potential knowledge in a distributed environment.

The application of agent technology as a major method to the implementation of data mining techniques has been studied by many researchers. Kargupta et al. [1997] applied agent technology to design a parallel and distributed data mining system named PADMA (Parallel Data Mining Agents). The system interfaces with users via a Web browser. Bose and Sugumaran [1999] designed an agent-based data mining system called the Intelligent Data Miner (IDM). They implemented a prototype of IDM using Java language and Java Agent Template Lite (JATLite available from http://java.stanford.edu) which is a set of Java templates and agent infrastructure. Some researchers proposed to employ heterogeneous techniques to perform data mining tasks. Recon [Kerber et al., 1995] is an example of a hybrid system containing inductive, clustering, case-based reasoning and statistical package for data mining. Zhang and Zhang [2004] also implemented data mining based hybrid intelligent systems. They demonstrated agent perspectives through the re-implementation of Weka system [Witten and Frank, 2005] using the agent communication language KQML (Knowledge Query and Manipulation Language) [Finin et al., 1997]. Gao et al. [2005] proposed a model called CoLe (Cooperative Learning) to handle the situation that agents employ different methods to access different types of information in heterogeneous data sets. Ong et al. [2005] also developed a multiagent system based on the concept of data stream processing to perform a data mining task in distributed dynamic environments.

An agent-based approach has been widely accepted as an appropriate paradigm to implement an intelligent system because of its flexibility, modularity and ability to take advantage of distributed resources. The integration of heterogeneous data source is one major characteristic of practical data mining systems that have to search for interesting patterns from huge amount of data, possibly locating at remote sites. An agent technology is thus the promising technique in the knowledge discovery setting that real-world data is evolving, distributed and non-homogeneous. Pursuing the same direction as other researchers, we also propose an agent-based model to implement knowledge discovery. We, however, consider a different paradigm on the agent-based data mining implementation. Instead of implementing with imperative paradigm using common languages such as C, Java, Visual Basic, we employ the declarative paradigm and implement the system with Haskell and Prolog languages. The power of declarative programming has paid of as shown in our experimental results. The rest of this paper is organized as follows. The next section briefly discusses the concepts of declarative versus imperative programming. We then present our agent model in section 3. The detail and some excerpts of our implementation are explained in section 4. Section 5 illustrates the experimental results. Section 6 concludes the paper.

2. 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 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. The evaluation of a program is simply the evaluation of functions. 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 examples of coding quick sort in C, Haskell and Prolog (figure 1) verify the previous statement.

One major task in data mining is searching for frequent patterns. 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 [Hudak et al., 1996].

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;}
```

Figure 1. Quick sort program in Haskell, Prolog, and C languages.

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          -- declaring a function that 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 n^{th} 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.

In Prolog, the feature of pattern matching can be defined through the use of arguments. For example, the following program demonstrates the fib function (in Prolog it is called predicate instead of function) to find the n^{th} number in the Fibonacci sequence. Last argument is normally a place holder for an output.

```
% Fibonacci function in Prolog
fib(0, 0).                  % pattern 1: input number is 0, then output is 0
fib(1, 1).                  % pattern 2: input number is 1, then output is 1
fib(N, F) :- N > 1,         % pattern 3: input number >1, then
             N1 is N-1, N2 is N-2,        % create new variables: N1 and N2
             fib(N1, F1), fib(N2, F2),    % recursively call fib
             F is F1 + F2.                % compute final result F
```

3. THE AGENT-BASED MODEL

We propose an agent-based knowledge discovery model (as shown in figure 2) to compose of three layers: data source layer, agent layer, and external layer. A community of agents is in the agent layer situated to help users to access and get only promising knowledge for their discovery tasks. Locating and accessing, filtering, and mining are three major activities of these agents.

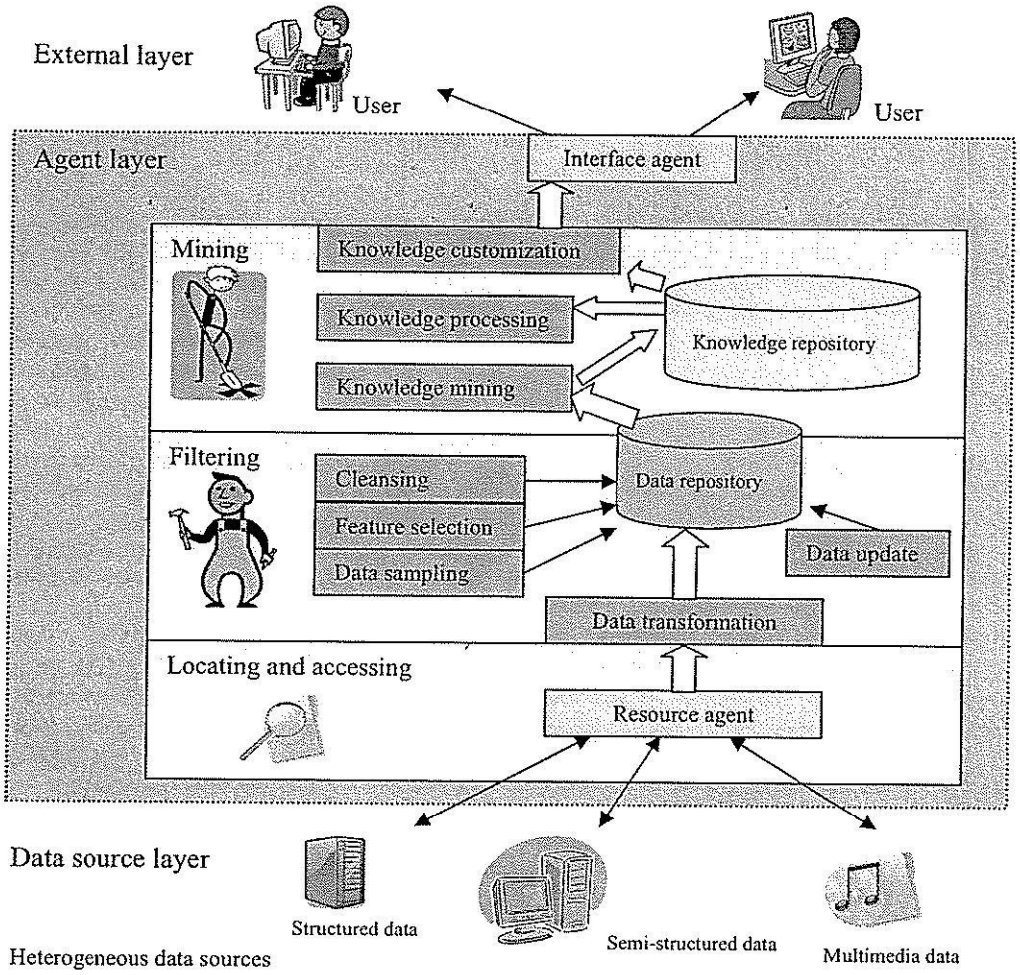


Figure 2. Agent-based model in a knowledge discovery system.

Locating and accessing. At the lowest level of the proposed framework, multiple heterogeneous data sources are located in an enterprise environment. These data sources may be distributed across a network such as intranet or internet. Resource agent is thus responsible for making the underlying data available to the data transformation agent in the upper filtering sub-layer. The resource agent also monitors changes in data contents to report any corresponding modification to the data-update agent. To implement the functions of resource agent, the following modules are required.

- *Data source specification.* The resource agent must be able to announce its location and the specification of its contents to other agents in the community.
- *Query processor.* The agent has to handle the update and the query upon the data contents. The query processor must also have the ability to reason whether its data contents match the needs announced by the knowledge mining agent.
- *Event-detection module.* This module is responsible for detecting the update on the data contents.
- *Data access module.* The resource agent assigns different access modules for different kinds of data sources.

Filtering. The agents in this class are the most autonomous and sophisticated ones due to the self-adjusting and specific functioning of each agent. The agents in this class are composed of:

- *Data update agent.* The agent communicates with resource agent to probe any changes in the environment and reflect those changes to data repository.

- *Data transformation agent.* Its main responsibility is to turn the input data to the right format.
- *Cleansing agent.* This agent is responsible for getting rid of any noise and handling missing values in the data contents.
- *Feature selection agent.* This agent efficiently evaluates and selects the most promising features out of the available data.
- *Data sampling agent.* This agent is invoked to obtain representatives appropriate for a specific mining task.

Mining. The agents in this class are mainly responsible for performing the data mining techniques. Data obtained from the filtering sub-layer will be turned into valuable and actionable knowledge by these agents:

- *Knowledge mining agent.* It is actually a group of agents, each agent performs a specific mining technique.
- *Knowledge processing agent.* Mined knowledge could be overwhelming or low accurate. It is thus the responsibility of this agent to post-process knowledge discovered by the mining agents.
- *Knowledge customization agent.* Some knowledge might be accurate but uninterested to the user. This agent is responsible for getting only knowledge pertaining to each user interest and delivers customized knowledge through the interface agent.

4. IMPLEMENTATION

We implemented association mining to discover frequent patterns with Apriori algorithm [Agrawal et al., 1993; Agrawal and Srikant, 1994]. Some parts of the program are shown in figure 3. 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 the 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. 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.

```
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]
```

(a) Haskell implementation

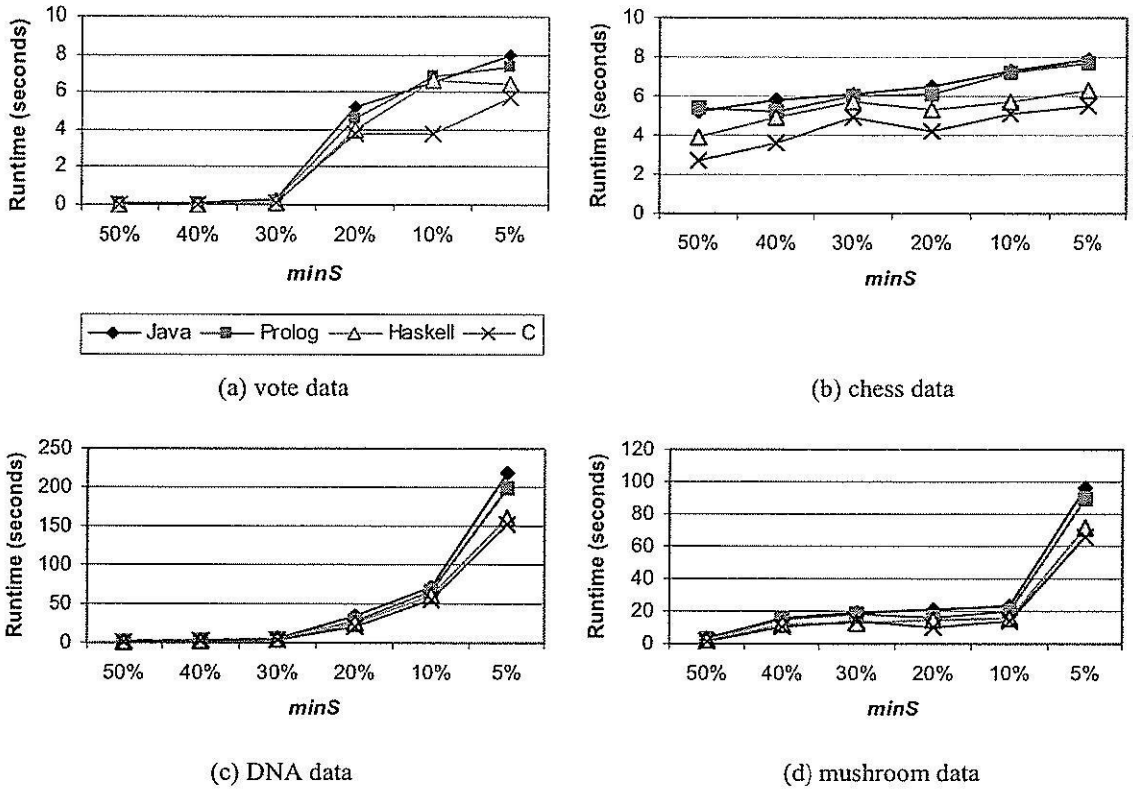
```
r1:- n(X), cL1(X).
r2(X):- cC2(X).
clear:-retractall(l1(_)),  retractall(c1(_)),
       retractall(c2(_)) , retractall(l2(_)).
% Create L1
cL1([]).
cL1([H|T]) :-  findall(X,f([H],X),L), length(L, Len),
               Len >= 2 , !,
               cL1(T), assert(l1(([H], Len)))
               ;
               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).
f(H, X) :-  item(X), subset(H, X).
```

(b) Prolog implementation

Figure 3. Frequent pattern discovery implemented with declarative languages.

5. EXPERIMENTATION

We comparatively study the performance of our implementations of frequent pattern discovery using Haskell and Prolog versus C and Java (source codes of C and Java implementations are taken from [Borgelt, 2003]). All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD. We tested the speed and memory usage of the programs on different datasets obtained from the UCI Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). Some results on four datasets, vote data (13.2 KB, 300 transactions, 17 items), chess data (237 KB, 2130 transactions, 37 items), DNA data (252 KB, 2000 transactions, 61 items), and mushroom data (916 KB, 5416 transactions, 23 items) are shown in this section. The frequent pattern discovery implementations have been tested on each dataset with various *minS* (minimum support) values.



(a) vote data

(b) chess data

(c) DNA data

(d) mushroom data

Figure 4. The comparison on computation speed of declarative versus imperative programming.

It can be noticed from the experimental results that on a speed comparison (figure 4), C implementation is the fastest, Haskell comes at a second fastest following by Prolog and Java. On the memory usage comparison (figure 5), 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 and almost negligible. 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.

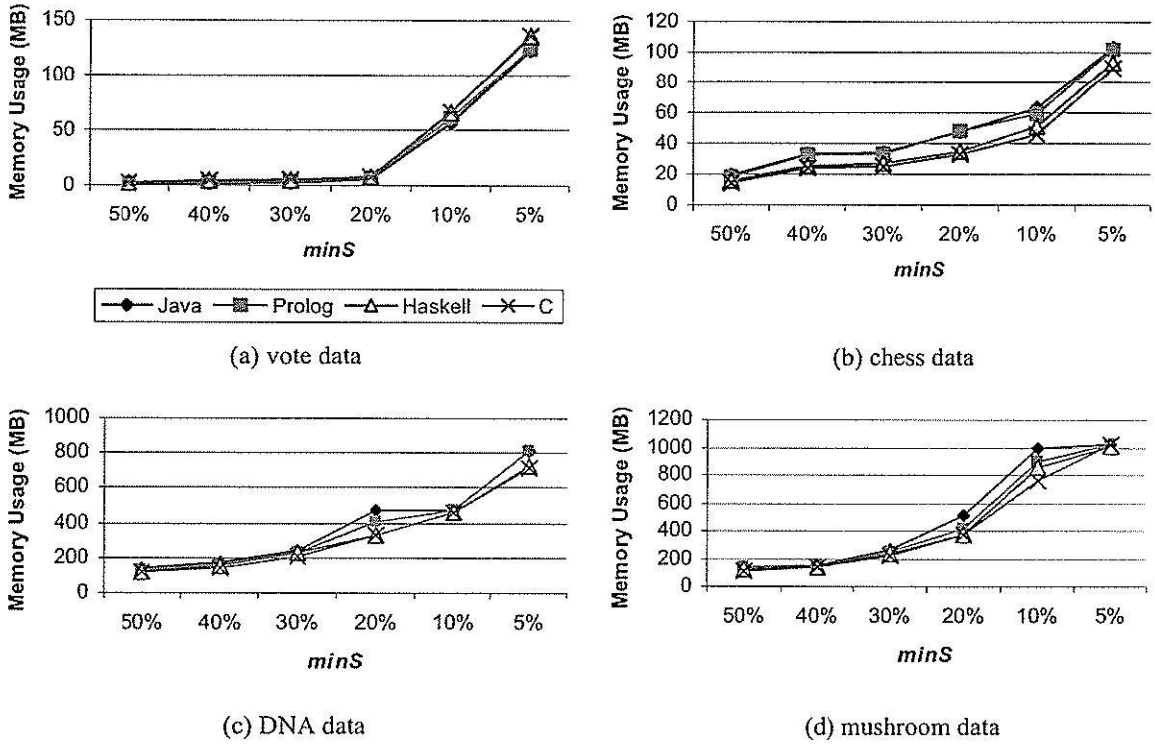


Figure 5. Memory usage comparison of declarative versus imperative programming.

6. CONCLUSION

The contribution of this paper is the design and implementation of a knowledge discovery system to provide an integrated, flexible, and efficient platform supported by a community of agents. This platform provides mechanisms of data browsing and extracting, data arrangement, data quality evaluation, data mining, knowledge processing and knowledge customization for the whole process of knowledge discovery. The agent model is designed with the three-layer architecture. Data source layer is at the back-end responsible for locating and accessing data from the remote sites. External layer is the user interface part. The core of our design is the agent layer which is in the middle between the external and the data source layers. Agent layer is divided into three sub-layers: locating and accessing, filtering, and mining. These agents work autonomously and cooperatively to deliver knowledge assets that meets specific interest of each user.

The proposed agent model has been implemented with declarative programming using Haskell and Prolog languages. We employ this paradigm with the intuitive idea that the problem of knowledge discovery should be efficiently and concisely implemented with high-level declarative languages. This idea has been tested on a specific problem of frequent pattern discovery which is a 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.

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 C or Java implementations in terms of speed and memory usage.

This preliminary study supports our belief regarding declarative programming paradigm towards a complex problem of knowledge 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 implementation in the course of concurrency to improve its performance.

Agents are designed to be active and intelligent. They are able to react appropriately to unpredictable situations, evaluate and apply their own problem solving strategies. However, the current design has to be extensively tested on various application domains. Several areas of extensions are currently being investigated. The functionalities of filtering agents can be extended to support new techniques of cleansing and adaptive sampling. Mining agents are also in the course of further improvement.

ACKNOWLEDGEMENT

This work was supported by the Thailand Research Fund under grant RMU-5080026 and the National Research Council of Thailand. The authors are with the Data Engineering and Knowledge Discovery (DEKD) research unit which is fully supported by research fund of Suranaree University of Technology.

REFERENCES

Agrawal, R. et al., 1993. Mining association rules between sets of items in large databases. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 207-216.

Agrawal, R. and Srikant, R., 1994. Fast algorithm for mining association rules. *Proceedings of International Conference on Very Large Data Bases*, pp. 487-499.

Borgelt, C., 2003. *Frequent Item Sets Miner for FIMI 2003*. http://www.borgelt.net/software.html.

Bose, R. and Sugumaran, V., 1999. Application of intelligent agent technology for managerial data analysis and mining. *In The Data Base for Advances in Information Systems*, Vol.30, No.1, pp. 77-94.

Fayyad, U. et al., 1995. From data mining to knowledge discovery: An overview. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth (eds.), *Advances in Knowledge Discovery and Data Mining*. AAAI Press, pp. 1-34.

Finin, T. et al., 1997. KQML as an agent communication language. In J. Bradshaw (ed.), *Software Agents*, AAAI Press/ The MIT Press, pp. 291-316.

Gao, J. et al., 2005. A cooperative multi-agent model and its application to medical data on diabetes. *Proceedings of International Workshop on Autonomous Intelligent Systems: Agents and Data Mining*, pp. 93-107.

Han, J. and Kamber, M., 2006. *Data Mining: Concepts and Techniques, 2nd edition.* Morgan Kaufmann.

Hudak, P. et al., 1996. A gentle introduction to Haskell. *Technical Report Yale U/DCS/RR-901*, Yale University.

Kargupta, H. et al., 1997. Scalable, distributed data mining using an agent based architecture. *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pp. 211-214.

Kerber, R. et al., 1995. A hybrid system for data mining. In S. Goonatilake and S. Khebbal (eds.), *Intelligent Hybrid System*, John Wiley & Sons, pp. 121-142.

Ong, K. et al., 2005. Agents and stream data mining: A new perspective. *In IEEE Intelligent Systems*, Vol.20, No.3, pp. 60-67.

Russell, S. and Norvig, P., 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Weiss, G. (ed.), 1999. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press.

Witten, I. and Frank, E., 2005. *Data Mining: Practical Machine Learning Tools and Techniques, 2nd edition*. Morgan Kaufmann.

Wooldridge, M., 1997. Agent-based software engineering. *In IEE Proceedings on Software Engineering*, Vol.144, No.1, pp. 26-37.

Wooldridge, M., 2002. *An Introduction to Multiagent Systems*. John Wiley & Sons.

Wooldridge, M. and Jennings, N., 1995. Intelligent agents: Theory and practice. *In The Knowledge Engineering Review*, Vol.10, No.2, pp. 115-152.

Zhang, Z. and Zhang, C., 2004. Constructing hybrid intelligent systems for data mining from agent perspectives. In N. Zhong and J. Liu (eds.), *Intelligent Technologies for Information Analysis*, Springer, pp. 333-359.

แนวทางการโปรแกรมเชิงฟังก์ชันกับการพัฒนาระบบเอสยูทีไมเนอร์

FUNCTIONAL PROGRAMMING PARADIGM AND THE DEVELOPMENT OF SUT MINER SYSTEM

กิตติศักดิ์ เกิดประสพ และ นิตยา เกิดประสพ

Kittisak Kerdprasop and Nittaya Kerdprasop

Data Engineering and Knowledge Discovery (DEKD) Research Unit, School of Computer Engineering, Suranaree University of Technology, Muang District, Nakhon Ratchasima 30000. E-mail: KittisakThailand@gmail.com

บทคัดย่อ: เอสยูทีไมเนอร์คือระบบเหมืองข้อมูลที่พัฒนาขึ้นที่มหาวิทยาลัยเทคโนโลยีสุรนารีเพื่อใช้เป็นเครื่องมือวิเคราะห์ข้อมูลอัจฉริยะสำหรับค้นหารูปแบบหรือสารสนเทศที่เป็นประโยชน์จากข้อมูลที่เก็บอยู่ในฐานข้อมูล ผู้วิจัยได้นำเสนอความก้าวหน้าของการพัฒนาเอสยูทีไมเนอร์ที่จัดเป็นระบบเหมืองข้อมูลสมบูรณ์แบบ เนื่องจากได้รวมส่วนการประมวลผลก่อนและหลังการทำเหมืองข้อมูลเข้าไว้ในระบบเดียวกัน ในบทความนี้ผู้วิจัยได้นำเสนอโครงร่างของระบบและวิธีการพัฒนาระบบในส่วนของโปรแกรมการทำเหมืองโดยใช้วิธีการโปรแกรมเชิงฟังก์ชันด้วยภาษาฮาสเกล วิธีการโปรแกรมเชิงประกาศของภาษาฮาสเกลช่วยให้การเขียนคำสั่งทำได้สั้นและชัดเจน นอกจากนี้การสนับสนุนการทำแพทเทิร์นแมทชิงของภาษาฮาสเกลยังเป็นข้อได้เปรียบอย่างมากสำหรับงานการค้นหารูปแบบ

Abstract: SUT Miner is a data mining system developed at Suranaree University of Technology as an intelligent data analysis tool to discover patterns and extract useful information from facts stored in databases. We present work in progress on the development of the SUT Miner, a complete data mining system. In addition to the mining engine, our system incorporates the pre-mining and post-mining parts. In this paper, we describe a framework of SUT Miner and present the implementation scheme on the mining engine part using a functional programming paradigm, a Haskell language in particular. A high-level declarative style of Haskell facilitates a clear and concise coding. The language also supports pattern matching, which is a big advantage for a task of pattern discovery.

Introduction: *Data Mining* (DM) or *Knowledge Discovery in Databases* (KDD) has been defined [3] as the automatic discovery of previously unknown patterns (or models, relationships) in large and complex datasets. *Pattern* is an expression describing a subset of the data, *e.g.* $f(x) = 3x^2 + 3$ is a pattern induced from a given dataset {(0,3), (1,6), (2,15), (3,30)}, whereas the term *model* refers to a representation of the source generating the data, *e.g.* $f(x) = ax^2 + b$. However, in his paper we use the term pattern and model interchangeably. The process of DM, thus, involves fitting models to, or determining patterns from, observed data. Finding patterns has become an important task because those patterns reveal associations, correlations, and many other interesting relationships hidden in stored data. Most of the proposed mining algorithms have been implemented with imperative programming languages such as C, C++, Java.

The imperative paradigm is significantly inefficient when a dataset is large and the hidden pattern is long. We suggest a high-level declarative style of programming using a functional language. Our supposition is that the problem of pattern discovery can be efficiently and concisely implemented via a functional paradigm since pattern matching is a fundamental feature supported by most functional languages.

Methodology: At a high level of our framework, we design the SUT-Miner system to be comprised of three main phases: pre-DM, DM, and post-DM. The pre-DM phase performs

data preparation tasks such as to locate and access relevant data set(s), transform the data format, clean the data if there exists noise and missing values, reduce the data to a reasonable and sufficient size with only relevant attributes. The DM phase performs mining tasks including classification, clustering, and association. The post-DM phase involves knowledge evaluation, based on corresponding measurement metrics, of the mining results. DM is an iterative process in that some parameters can be adjusted and then restart the whole process to produce a better result. The post-DM phase is composed of knowledge evaluator, knowledge reducer, and knowledge integrator. These three components perform major functionalities aiming at a feasible knowledge deployment. The overall architecture of our SUT-Miner system is presented in figure 1.

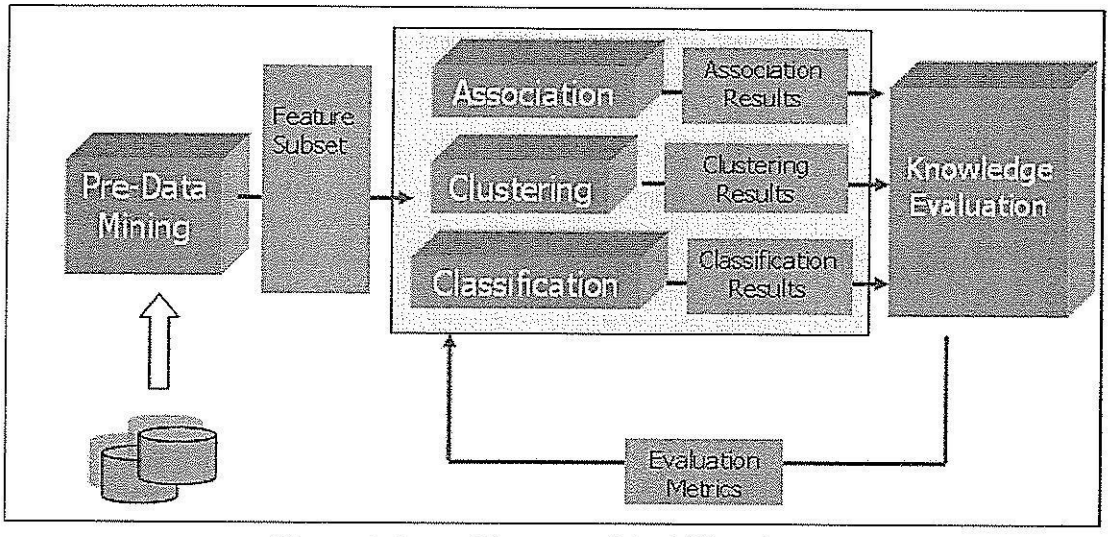


Figure 1 An architecture of the SUT Miner system

The implementation of SUT-Miner system is mainly based on the functional programming paradigm using Haskell language [2], [4]. Functional languages (FL) offer a number of advantages over imperative languages (IL). FL can be used to express specifications of problems in a more concise form than IL. This results in the creation of program source codes that are shorter and easier to understand. The example in figure 2 shows C versus Haskell codes to compute a list of fibonacci numbers starting with zero.

C-code

```
int * fib (int n)
{  int a = 0, b = 1, i, temp;
   int * fibsequence;
   fibsequence = (int *) malloc ((sizeof int) *n);
   for (i = 0; i<n; i++)
      {  fibsequence[i] = a;
         temp = a + b;
         a = b;
         b = temp;
      }
   return fibsequence;
}
```

Haskell-code

```
fib :: [Int]
fib = 0: 1: [ a+b | (a, b) <- zip fib (tail fib) ]
```

Figure 2 C versus Haskell codes to compute fibonacci numbers

Haskell is a pure FL having a polymorphic type system, *i.e.* a data type can take type variables as parameters. This feature provides a high level abstraction leading to generic programming. Haskell is also a lazy FL, *i.e.* a value is evaluated only when it is needed. This feature allows infinite structures, such as an infinite sequence of fibonacci numbers, to be defined.

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 [5], [7]. 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 n^{th} 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.

We implement Apriori algorithm [1] and K-means clustering [6] using Haskell language as shown in figures 3 and 4, respectively. In figure 3, each item is represented by the item identifier which is an integer. Thus, an itemset 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 itemset. Functions listC and listC' perform the task of enumerating candidate frequent itemsets. Only itemsets that satisfy the *minSup* threshold are reported from the functions listL and listL' as frequent itemsets. It can be seen that the discovery of frequent itemsets in association mining using Haskell functional language takes only 20 lines of code.

```
itemSet :: [Set Int]
itemSet =[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<-itemSet]
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>=minSup]
listL'::Int->[Set Int]
listL' n =[x|(x,_)<-listL n]
```

Figure 3 Frequent itemset discovery implemented with Haskell

```
cluster::(Center,Center)->[Point]->((Center,Center),[Point],[Point])
cluster (c1,c2) pts = ((newc1,newc2),cluster1,cluster2)
  where cluster1=[point | point<-pts, inC1 (c1,c2) point]
        newc1=(newx1,newy1)
        newx1 =div (sumx cluster1) l1
        newy1 =div (sumy cluster1) l1
        newc2=(newx2,newy2)
        cluster2=[point | point<-pts, not (inC1 (c1,c2) point)]
        newx2 =div (sumx cluster2) l2
        newy2 =div (sumy cluster2) l2
        l1=length cluster1
        l2=length cluster2
        sumx pts =sum[x | (x,y)<-pts]
        sumy pts =sum[y | (x,y)<-pts]
        inC1 ((x1,y1),(x2,y2))(x,y)
            | len (x1,y1) (x,y) <len (x2,y2) (x,y) =True
            | otherwise =False
            where len (x1, y1) (x,y)  =  ( abs(x-x1) + abs(y-y1) )
```

Figure 4 K-means clustering on two-dimensional data implemented in Haskell

Results, Discussion and Conclusion: We present work in progress on the development of the SUT-Miner, a complete data mining system. The system is complete in that the pre-DM and post-DM phases are also included in the DM process. Most DM packages contain only the DM modules, while some systems incorporate a pre-DM module as a data preparation phase.

According to our knowledge, a post-DM phase is omitted in most systems. Post-processing of DM is very essential to the success of DM utilization. This is due to the fact that discovered knowledge is sometimes voluminous and redundant. At present, knowledge evaluation and filtration have to be done by human experts. We thus design our system to include this knowledge processor as another major component of the mining system.

The implementation of the SUT-Miner system uses a Haskell functional language. The functional programming is a paradigm of our choice because of its advantages on modularity, conciseness, polymorphism, and formal specification which supports the proof of program correctness. We plan to extend our design to produce an approximate model by means of progressive mining. We currently investigate the feasibility of applying a Markov Chain Monte Carlo method in our approximate data mining scheme.

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. Bird, Introduction to Functional Programming using Haskell, Prentice Hall, 1998.
3. U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery: An Overview," in *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.
4. P. Hudak, J. Fasel, and J. Peterson, "A gentle introduction to Haskell," Yale University, *Technical Report Yale U/DCS/RR-901*, 1996.
5. P. Jones and J. Hughes (eds.), *Standard Libraries for the Haskell 98 Programming Languages*. Available: http://www.haskell.org/library/.
6. J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symposium on Multivariate Statistics and Probability*, 1967, pp.281-297.
7. S. Thompson, *Haskell: The Craft of Functional Programming* (2nd ed.), Addison Wesley, 1999.

Keywords: data mining, pattern discovery, Haskell, functional programming

Acknowledgements: This work was fully supported by research fund of Suranaree University of Technology granted to the Data Engineering and Knowledge Discovery (DEKD) Research Unit, in which Kittisak Kerdprasop is a director and Nittaya Kerdprasop is a member and researcher. The authors are also partly supported by National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF). The authors would like to thank all the research assistants who participated in the SUT-Miner project.

ประวัติผู้วิจัย

รองศาสตราจารย์ ดร.กิตติศักดิ์ เกิดประสพ สำเร็จการศึกษาในระดับปริญญาเอกสาขา Computer Science จาก Nova Southeastern University เมือง Fort Lauderdale รัฐฟลอริดา ประเทศสหรัฐอเมริกา เมื่อปีพุทธศักราช 2542 (ค.ศ. 1999) ด้วยทุนการศึกษาของทบวงมหาวิทยาลัย (หรือสำนักงานคณะกรรมการการอุดมศึกษาในปัจจุบัน) โดยทำวิทยานิพนธ์ระดับปริญญาเอกในหัวข้อเรื่อง "Active database rule set reduction by knowledge discovery" หลังสำเร็จการศึกษาได้ปฏิบัติงานในตำแหน่งอาจารย์ ประจำสาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ปัจจุบันดำรงตำแหน่งหัวหน้าหน่วยปฏิบัติการวิจัยด้านวิศวกรรมข้อมูลและการค้นหาความรู้ (Data Engineering and Knowledge Discovery Research Unit – DEKD) สำนักวิชาวิศวกรรมศาสตร์ ดำเนินการวิจัยประยุกต์เกี่ยวกับการออกแบบและพัฒนาระบบเหมืองข้อมูลประสิทธิภาพสูงที่สามารถทนต่อข้อมูลรบกวน และการวิจัยพื้นฐานเกี่ยวกับเทคนิคการจัดกลุ่มข้อมูล และเทคนิคการวิเคราะห์ข้อมูลขนาดใหญ่ด้วยฮิวริสติก โดยมีผลงานวิจัยตีพิมพ์ในวารสารวิชาการและเอกสารการประชุมวิชาการ จำนวนมากกว่า 30 เรื่อง ในสาขาระบบฐานความรู้ ฐานข้อมูลแอคทีฟ ฐานข้อมูลนิรนัย การทำเหมืองข้อมูลและการค้นหาความรู้

รองศาสตราจารย์ ดร.นิตยา เกิดประสพ สำเร็จการศึกษาในระดับปริญญาเอกสาขา Computer Science จาก Nova Southeastern University เมือง Fort Lauderdale รัฐฟลอริดา ประเทศสหรัฐอเมริกา เมื่อปีพุทธศักราช 2542 (ค.ศ. 1999) ด้วยทุนการศึกษาของกระทรวงวิทยาศาสตร์ฯ โดยทำวิทยานิพนธ์ระดับปริญญาเอกในหัวข้อเรื่อง "The application of inductive logic programming to support semantic query optimization" หลังสำเร็จการศึกษาได้ปฏิบัติราชการในตำแหน่งอาจารย์ ประจำสาขาคอมพิวเตอร์ ภาควิชาคณิตศาสตร์ คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ต่อมาในปีพุทธศักราช 2543 ได้มาปฏิบัติงานในตำแหน่งอาจารย์ประจำสาขาวิชาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยเทคโนโลยีสุรนารี จนถึงปัจจุบัน งานวิจัยที่ทำในขณะนี้คือการพัฒนาระบบเหมืองข้อมูลประสิทธิภาพสูงที่สามารถทนต่อข้อมูลรบกวน และการเพิ่มความสามารถในการจัดการความรู้ของระบบเหมืองข้อมูล