เอกสารอ้างอิง

R. Agrawal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61:350-371, 2001.

R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'93)*, pages 207-216, Washington, DC, May 1993.

R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. In *Research Report RJ 9839*, IBM Almaden Research Center, San Jose, CA, June 1994.

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487-499, Santiago, Chile, Sept. 1994.

R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Engineering*, 8:962-969, 1996.

B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Model and issues in data stream systems. In *Proc. ACM Symp. Principles of Database Systems (PODS'02)*, 2002.

Y. Cai, G.Pape, J. Han, M. Welge, and L. Auvil. MAIDS: Mining alarming incidents from data streams. In *Proc. Int. Conf. on Management of Data*, June 2004.

J. Chang and W. Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering*; July 2004.

M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, Jan. 2004.

D. W. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proc. 1996 Int. Conf. Parallel and Distributed Information Systems*, pages 31-44, Miami Beach, Florida, Dec. 1996.

D. W. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 1996 Int. Conf. Data Engineering (ICDE'96)*, pages 106-114, New Orleans, Louisiana, Feb. 1996.

Y. Chi, H. Wang, P. Yu, and R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proc. IEEE Int. Conf. on Data Mining*, Nov. 2004.

M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Resource-aware knowledge discovery in data streams. In *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, Sept. 2004.

M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2), June 2005.

A. Ghoting and S. Parthasarathy. Facilitating interactive distributed data stream processing and mining. In *Proc. IEEE Int. Symposium on Parallel and Distributed Processing Systems*, Apr. 2004.

G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Proc. *ICDM'03 Int. Workshop on Freuquent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

S. Guha, N. Koudas, and K. Shim. Data streams and histograms. In *Proc. ACM Symposium on Theory of Computing*, 2001.

M. Halatchev and L. Gruenwald. Estimating missing values in related sensor data streams. In *Proc. Int. Conf. on Management of Data*, Jan. 2005.

J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 420-431, Zurich, Switzerland, Sept. 1995.

J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1-12, Dallas, TX, May 2000.

M. Jiang and L. Gruenwald. Research issues in data stream association mining. *ACM SIGMOD Record*, 35(1): 14-19, 2006.

H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy. VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring. In *Proc. SIAM Int. Conf. on Data Mining*, 2004.

K. Kerdprasop, N. Kerdprasop, and P. Sattayatham. Density-biased clustering based on reservoir sampling. In *Proc. 16th Int. Workshop on Database and Expert Systems Applications (DEXA)*, pages 1122-1126, Copenhagen, Denmark, Aug. 22-26, 2005.

K. Kerdprasop, N. Kerdprasop, and P. Sattayatham. A Monte Carlo method to data stream analysis. *Enformatika Transactions on Engineering, Computing and Technology*, 14: 240-245, Aug. 2006.

H.-F. Li, S.-Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, Sept. 2004.

C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, and A. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proc. SIAM Int. Conf. on Data Mining*, April 2005.

J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 239-248, Edmonton, Canada, July 2002.

G. Mao, X. Wu, C. Liu, X. Zhu, G. Chen, Y. Sun, and X. Liu. Online mining of maximal frequent itemsequences from data streams. *Technical Report CS-05-07*, University of Vermont, Computer Science, June 2005.

J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'95)*, pages 175-186, San Jose, CA, May 1995.

J. S. Park, M. S. Chen, and P. S. Yu. Efficient parallel mining for association rules. In *Proc. 4th Int. Conf. Information and Knowledge Management*, pages 31-36, Baltimore, Maryland, Nov. 1995.

J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215-224, Heidelberg, Germany, April 2001.

A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95), pages 432-443, Zurich, Switzerland, Sept. 1995.

W.-G. Teng, M.-S. Chen, and P. Yu. Resource-aware mining with variable granularities in data streams. In *Proc. SIAM Int. Conf. on Data Mining*, 2004.

H. Toivonen. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 134-145, Bombay, India, Sept. 1996.

J. S. Vitter. Random sampling with a reservoir. *ACM Transaction on Mathematical Software*, 11(1): 37-57, 1985.

J. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proc. Int. Conf. on Very Large Databases*, 2004.

M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithm for discovery of association rules. *Data Mining and Knowledge Discovery*, 1:343-374, 1997.

ผลผลิตจากโครงการวิจัยที่ได้รับทุนจาก สกอ. และ สกว.

1. ผลงานตีพิมพ์ในหนังสือวิชาการ (Book chapters)

1) K. Kerdprasop and N. Kerdprasop (2009). Knowledge mining with a higher-order logic approach. In K. Nakamatsu, G. Phillips-Wrens, L.C. Jain, R.J. Howlett (Eds.), *New Advances in Intelligent Decision Technologies*, pp. 151-159, Springer. (ISBN: 978-3-642-00908-2, DOI: 10.1007/978-3-642-00909-9)

2) N. Kerdprasop, S. Pilabutr, and K. Kerdprasop (2009). Improving medical database consistency with induced trigger rules. In K. Nakamatsu, G. Phillips-Wrens, L.C. Jain, R.J. Howlett (Eds.), *New Advances in Intelligent Decision Technologies*, pp. 265-274, Springer. (ISBN: 978-3-642-00908-2, DOI: 10.1007/978-3-642-00909-9)

2. ผลงานตีพิมพ์ในวารสารวิชาการนานาชาติ (Journal articles)

1) N. Pannurat, N Kerdprasop, and K. Kerdprasop (2010). Database reverse engineering based on association rule mining. *International Journal of Computer Science Issues (IJCSI)*, Volume 7, Issue 2, March 2010, pp. 10-15.

2) N. Kerdprasop and K. Kerdprasop (2009). Knowledge induction from medical databases with higher-order programming. *WSEAS Transactions on Information Science and Applications*, Issue 10, Volume 6, October 2009, pp. 1719-1728.

3) K. Kerdprasop and N. Kerdprasop (2008). Approximate frequent pattern discovery over data stream. *International Journal of Computer Science and Engineering (IJCSE)*, Volume 2, Number 1, pp. 28-32.

4) K. Kerdprasop and N. Kerdprasop (2007). On pattern-based programming towards the discovery of frequent patterns. *International Journal of Computer Science (IJCS)*, Volume 2, Number 4, pp. 268-273.

5) N. Kerdprasop and K. Kerdprasop (2007). Mining frequent patterns with functional programming. *International Journal of Computer and Information Science and Engineering (IJCISE)*, Volume 1, Number 2, pp. 66-71.

6) N. Kerdprasop and K. Kerdprasop (2007). Moving data mining tools toward a business intelligence system. *International Journal of Intelligent Technology (IJIT)*, Volume 2, Number 2, pp. 99-104.

Cited by:

Z. Zhu, J. Gu, L. Zhang, W. Song, R. Gao. (2009). Research on domain-driven actionable knowledge discovery. *Proc. 20th Int. Conf. on Cutting-Edge Research Topics on Multiple Criteria Decision Making (MCDM 2009)*, 21-26 June 2009, China, pp.176-183.

M. Al-Noukari and W. Al-Hussan. (2007). Using data mining techniques for predicting future car market demand: dcx case study. *Proc. 3rd Int. Conf. on Information and Communication Technology (ICTTA 2008)*, 7-11 April 2008, pp.1-5.

3. การเสนอผลงานในที่ประชุมวิชาการและตีพิมพ์ผลงานใน Conference Proceedings

1) K. Kerdprasop and N. Kerdprasop (2009). Automated induction of frequent patterns with knowledge-based software engineering. *Proceedings of the Joint Conference of ASCM 2009 (Asian Symposium on Computer Mathematics) and MACIS 2009 (Mathematical Aspects of Computer and Information Sciences)*, 14-17 December 2009, Fukuoka, Japan, pp. 431-434.

2) N. Kerdprasop and K. Kerdprasop (2008). A declarative programming paradigm and the development of knowledge mining agents. *Proceedings of IADIS Multi Conference on Computer Science and Information Systems*, Amsterdam, Netherlands, 22-27 July 2008, pp. 45-52.

4. การนำผลงานวิจัยไปใช้ประโยชน์เชิงวิชาการ

1) ใช้ประกอบการเรียนการสอนในรายวิชา 423681 Selected Topics in Computer Engineering I ซึ่งเป็นรายวิชาเลือกในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์

2) สร้างนักวิจัยใหม่ด้วยการนำส่วนหนึ่งของงานวิจัยไปเป็นหัวข้อวิทยานิพนธ์ "การปรับรูปแบบบรรทัดฐานในฐานข้อมูลเชิงสัมพันธ์ด้วยเทคนิคการวิเคราะห์ความสัมพันธ์" (Normalization in relational database with association analysis technique) โดยนายณัฐพล พันนุรัตน์ นักศึกษาในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์

ภาคผนวก ก

บทความตีพิมพ์จากโครงการวิจัย

Book Chapters

1) K. Kerdprasop and N. Kerdprasop (2009). Knowledge mining with a higher-order logic approach. In K. Nakamatsu, G. Phillips-Wrens, L.C. Jain, R.J. Howlett (Eds.), *New Advances in Intelligent Decision Technologies*, pp. 151-159, Springer. (ISBN: 978-3-642-00908-2, DOI: 10.1007/978-3-642-00909-9)

2) N. Kerdprasop, S. Pilabutr, and K. Kerdprasop (2009). Improving medical database consistency with induced trigger rules. In K. Nakamatsu, G. Phillips-Wrens, L.C. Jain, R.J. Howlett (Eds.), *New Advances in Intelligent Decision Technologies*, pp. 265-274, Springer. (ISBN: 978-3-642-00908-2, DOI: 10.1007/978-3-642-00909-9)

Journal Articles

1) N. Pannurat, N Kerdprasop, and K. Kerdprasop (2010). Database reverse engineering based on association rule mining. *International Journal of Computer Science Issues (IJCSI)*, Volume 7, Issue 2, March 2010, pp. 10-15.

2) N. Kerdprasop and K. Kerdprasop (2009). Knowledge induction from medical databases with higher-order programming. *WSEAS Transactions on Information Science and Applications*, Issue 10, Volume 6, October 2009, pp. 1719-1728.

3) K. Kerdprasop and N. Kerdprasop (2008). Approximate frequent pattern discovery over data stream. *International Journal of Computer Science and Engineering (IJCSE)*, Volume 2, Number 1, pp. 28-32.

4) K. Kerdprasop and N. Kerdprasop (2007). On pattern-based programming towards the discovery of frequent patterns. *International Journal of Computer Science (IJCS)*, Volume 2, Number 4, pp. 268-273.

5) N. Kerdprasop and K. Kerdprasop (2007). Mining frequent patterns with functional programming. *International Journal of Computer and Information Science and Engineering (IJCISE)*, Volume 1, Number 2, pp. 66-71.

6) N. Kerdprasop and K. Kerdprasop (2007). Moving data mining tools toward a business intelligence system. *International Journal of Intelligent Technology (IJIT)*, Volume 2, Number 2, pp. 99-104.

Cited by:

i. Z. Zhu, J. Gu, L. Zhang, W. Song, and R. Gao. (2009). Research on domain-driven actionable knowledge discovery. *Proc. 20th Int. Conf. on Cutting-Edge Research Topics on Multiple Criteria Decision Making (MCDM 2009)*, 21-26 June 2009, China, pp.176-183.

ii. M. Al-Noukari and W. Al-Hussan. (2008). Using data mining techniques for predicting future car market demand: dcx case study. *Proc. 3rd Int. Conf. on Information and Communication Technology (ICTTA 2008)*, 7-11 April 2008, pp.1-5.

International Conference Proceedings

1) K. Kerdprasop and N. Kerdprasop (2009). Automated induction of frequent patterns with knowledge-based software engineering. *Proceedings of the Joint Conference of ASCM 2009 (Asian Symposium on Computer Mathematics) and MACIS 2009 (Mathematical Aspects of Computer and Information Sciences)*, 14-17 December 2009, Fukuoka, Japan, pp. 431-434.

2) N. Kerdprasop and K. Kerdprasop (2008). A declarative programming paradigm and the development of knowledge mining agents. *Proceedings of IADIS Multi Conference on Computer Science and Information Systems*, Amsterdam, Netherlands, 22-27 July 2008, pp. 45-52.

Knowledge Mining with a Higher-Order Logic Approach

Kittisak Kerdprasop and Nittaya Kerdprasop

Abstract. Knowledge mining is the process of deriving new and useful knowledge from vast volumes of data and patterns previously discovered and stored as background knowledge. We propose a knowledge-mining system as a repertoire of tools for discovering strong and useful patterns. A pattern is strong if it represents frequently occurring relationships. Usefulness is achieved through constraints guided by users. To be able to derive strong and useful patterns from underlying data and background knowledge we consider employing the concept of higher-order logic as a major approach of our implementation. Higher-order logic can greatly reduce the burden of programmers as it is a very high level programming scheme suitable for the development of knowledge-intensive tasks. We have shown in this paper frequent pattern mining implemented with higher-order logic. The implementation is applied to mine breast cancer data. Our design of a logic-based knowledge-mining system is intended to support higher-order and constraint mining which is the next step of our research direction.

1 Introduction

Knowledge is a valuable asset to most organizations as a substantial source to support better decisions and thus to enhance organizational competency. Researchers and practitioners in the area of knowledge management view knowledge in a broad sense as a state of mind, an object, a process, an access to information, or a capability [2, 11]. The term *knowledge assets* [17, 19] is used to refer to any organizational intangible assets related to knowledge such as know-how, expertise, intellectual property. In clinical companies and computerized healthcare applications knowledge assets include order sets, drug-drug interaction rules, guidelines for practitioners, and clinical protocols [10].

Knowledge assets can be stored in data repositories either in implicit or explicit form. Explicit knowledge can be managed through the existing tools available in the current database technology. Implicit knowledge, on the contrary, is harder to

Kittisak Kerdprasop and Nittaya Kerdprasop
Data Engineering and Knowledge Discovery Research Unit,
School of Computer Engineering, Suranaree University of Technology,
Nakhon Ratchasima 30000, Thailand

K. Nakamatsu et al. (Eds.): New Advan. in Intel. Decision Techno., SCI 199, pp. 151–159.
springerlink.com

achieve and retrieve. Specific tools and suitable environments are needed to extract such knowledge.

Implicit knowledge acquisition can be achieved through the availability of the knowledge-mining system. *Knowledge mining* is the discovery of hidden knowledge stored possibly in various forms and places in large data repositories. In health and medical domains, knowledge has been discovered in different forms such as association rules, classification, clustering, trend or temporal pattern analysis [20]. The discovered knowledge facilitates expert decision support, diagnosis and prediction.

In this paper we present the design of a complete knowledge-mining system to support a high-level decision not only in medical domains but also in any domain that requires a knowledge-based decision support. A rapid prototyping of the proposed system is also provided to highlight the fact that higher-order logic is an appropriate approach to the implementation of a complex knowledge-mining system. The intuitive idea of our design and implementation is that for such a complicated knowledge-based system program coding should be done declaratively at a high level to alleviate the burden of programmers. The declarative style of programming also eases the future extension of our system to cover the concepts of higher-order mining [16] and constraint programming [6] that should naturally applied to the task of knowledge mining.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 is the architecture of SUT-Miner, the proposed knowledge-mining system. Section 4 shows the implementation of association mining [1] using higher-order logic programming scheme with some running examples. Section 5 concludes the paper and discusses our future research directions.

2 Related Works

In recent years we have witnessed increasing number of applications devising database technology and machine learning techniques to mine knowledge from biomedicine, clinical and health data. Roddick et al [15] discussed the two categories of mining techniques applied over medical data: explanatory and exploratory. Explanatory mining refers to techniques that are used for the purpose of confirmation or making decisions. Exploratory mining is data investigation normally done at an early stage of data analysis in which an exact mining objective has not yet been set.

Explanatory mining in medical data has been extensively studied in the past decade employing various learning techniques. Bojarczuk et al [3] applied genetic programming method with constrained syntax to discover classification rules from medical data sets. Thongkam et al [21] studied breast cancer survivability using AdaBoost algorithms. Ghazavi and Liao [7] proposed the idea of fuzzy modeling on selected features medical data. Huang et al [9] introduced a system to apply mining techniques to discover rules from health examination data. Then they employed a case-based reasoning to support the chronic disease diagnosis and treatments. The recent work of Zhuang et al [25] also combined mining with case-based reasoning, but applied a different mining method. They performed data clustering based on self-organizing maps in order to facilitate decision support on solving new cases of pathology test ordering problem. Biomedical discovery

support systems are recently proposed by a number of researchers [4, 5, 8, 23, 24]. Some works [14, 18] extended medical databases to the level of data warehouses.

Exploratory, as oppose to explanatory, is rarely applied to medical domains. Among the rare cases, Nguyen and Kawasaki [13] introduced knowledge visualization in the study of hepatitis patients. Palaniappan and Ling [14] applied the functionality of OLAP tools to improve visualization.

It can be seen from the literature that most medical knowledge discovery systems have applied only some mining techniques such as classification rules mining, association mining, data clustering to discover hidden patterns and knowledge. We, on the contrary, design a knowledge-mining system aiming at providing a suite of tools to facilitate users and medical practitioners on discovering different kinds of knowledge from their data and background knowledge repositories.

3 SUT-Miner: A Knowledge-Mining system

Knowledge mining is a complex and possibly iterative process that involves many different steps. The main input to the process is data from heterogeneous sources, and the final output is the useful information desired by the users. For the medical domains, we design the system to be composed of two main phases as shown in figure 1. Knowledge induction phase is the back-end of the system responsible for acquiring and discovering new and useful knowledge. Usefulness is to be validated at the final step by human experts. Discovered knowledge is stored in the knowledge base to be applied to solve new cases in knowledge inferring phase which is the front-end of the proposed system.

SUT-Miner in a knowledge induction phase is comprised of three main modules: pre-DM, DM, post-DM. The term data mining (DM) means automatic learning of patterns or models from specific data. *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.* $f(x) = ax^2 + b$. In his paper we refer to both patterns and models as new knowledge discovered from data sources.

The pre-DM module performs data preparation tasks such as to locate and access relevant data sets, 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 module performs mining tasks including classification, prediction, clustering, and association. We adopt the ontology concept at this step to guide the mining methodology selection. A simple form of mining method ontology is shown in figure 2. The post-DM module is composed of two main components: knowledge evaluator and knowledge integrator. These components perform major functionalities aiming at a feasible knowledge deployment which is important for the applications in medical diagnosis and predicting. Knowledge evaluator involves evaluation, based on corresponding measurement metrics, of the mining results. Knowledge integrator examines the induced patterns to remove redundant knowledge. Ontology has also been applied at this step to provide essential semantics regarding the domain problems.

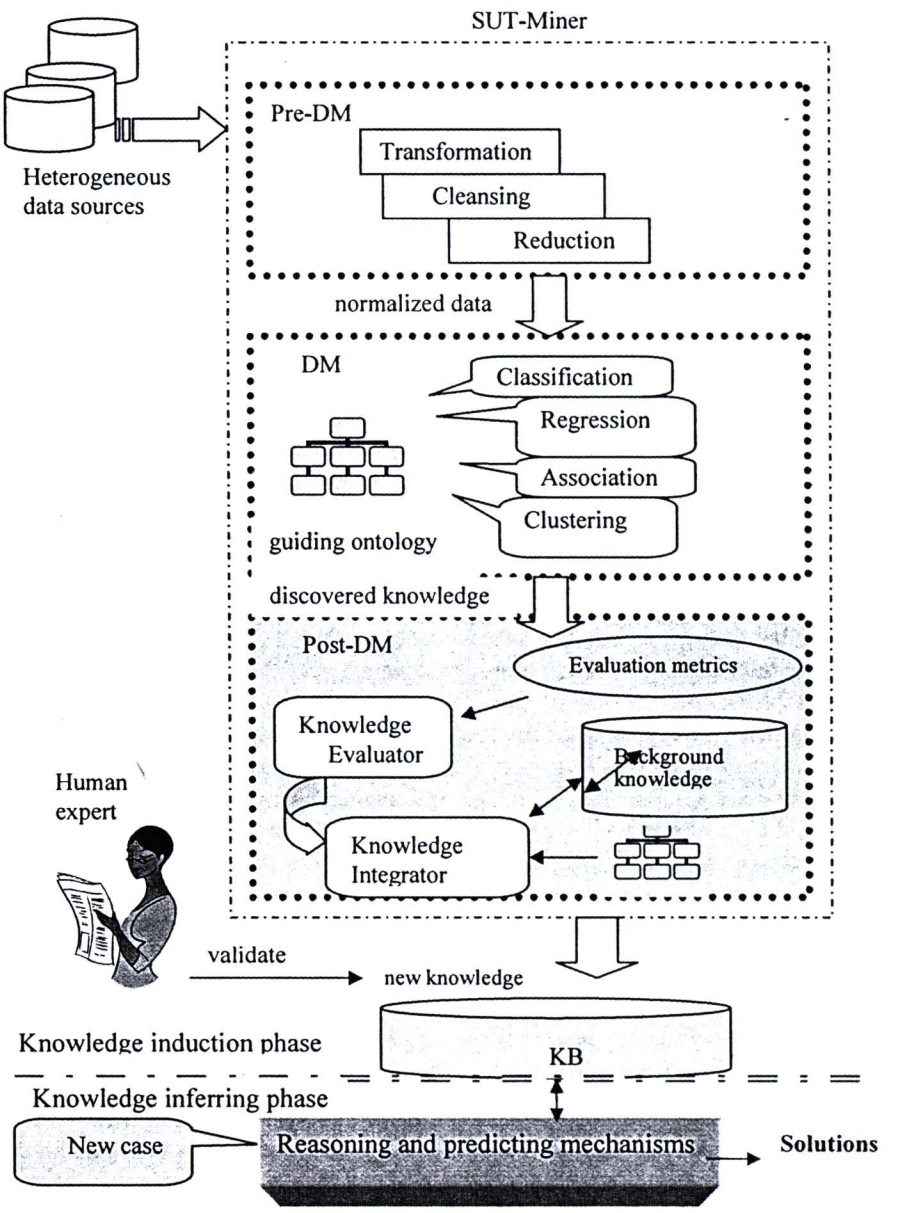


Fig. 1 Architecture of a knowledge-mining system

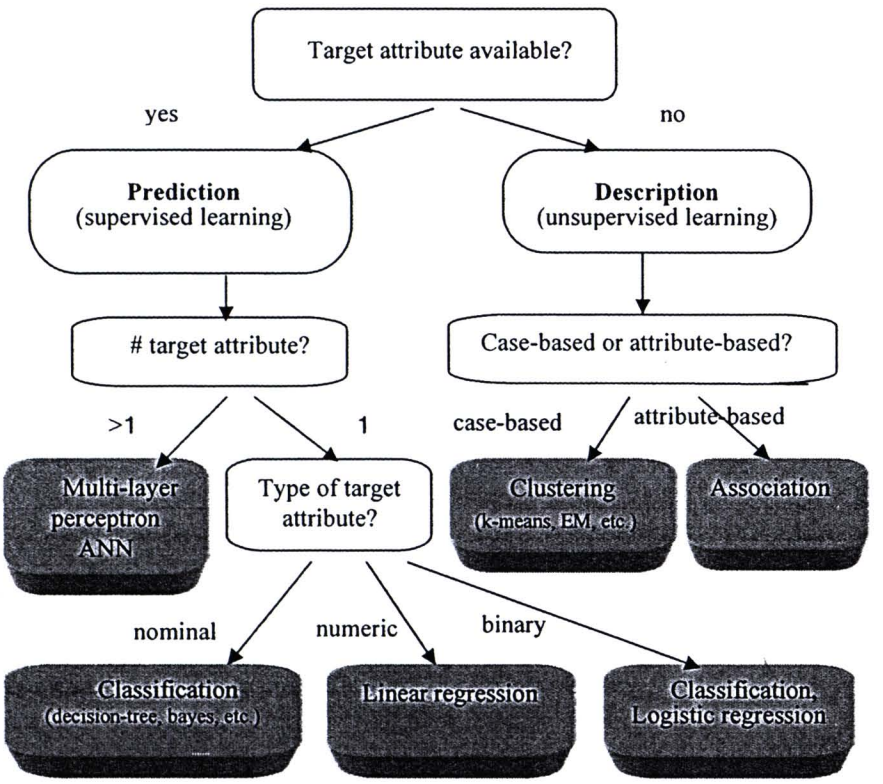


Fig. 2 Ontology for guiding mining method selection at the DM step

4 Implementation and Running Example

The SUT-Miner system has been implemented based on the concept of higher-order logic which is an extension of first-order logic to be more expressive and more powerful. First-order logic has been extensively used in intelligent systems [22] as an inference mechanism to deduce new facts. A classic example is that given a general rule $\forall x$ *Man(x)* => *Mortal(x)* and a known fact *Man(Socretes)*, we can deduce new fact that *Mortal(Socretes)*.

Despite its successful application to many computational problems such as natural language processing, first-order logic poses a restriction on the type of variables appearing in quantifications to exclude predicates. Higher-order logic [12], on the other hand, allows variables to quantify over predicates. With such relaxation, higher-order logic facilitates the implementation of a knowledge-intensive system that takes other knowledge as its input in a closed form. An exemplar in figure 3 demonstrates a higher-order logic-based implementation of frequent pattern mining problem. The coding is based on the syntax of SWI prolog (www.swi-prolog.org).

```
frequent_pattern_mining:-
   min_support(V),                          % set minimum support
   makeC1(C),                               % create candidate 1-itemset
   makeL(C,L),                              % compute large itemset
   apriori_loop(L,1).                       % recursively run apriori

apriori_loop(L,N):- length(L) is 1,!.       % base case of recursion
apriori_loop(L,N):-                         % inductive step
   N1 is N+1,
   makeC(N1,L,C),    makeL(C,Res),
   apriori_loop(Res,N1).

makeC1(Ans):-
   input(D),                        % input data as a list, e.g. [[a], [a,b]]
   allComb(1,ItemList,Ans2),              % make combination of itemset
   maplist(countSS(D),Ans2,Ans).        % scan database
                                    % countSS is predicate passing as argument
                                    % to a higher-order predicate maplist
makeC(N,ItemSet,Ans):-
   input(D),
   allComb(2,ItemSet,Ans1),
   maplist(flatten,Ans1,Ans2),
   maplist(list_to_ord_set,Ans2,Ans3),
   list_to_set(Ans3,Ans4),
   include(len(N),Ans4,Ans5),      % include is also a higher-order predicate
   maplist(countSS(D),Ans5,Ans). % scan database find: List+N

makeL(C,Res):-                      %for all L creation
   include(filter,C,Ans),           % call higher predicates include
   maplist(head,Ans,Res).           %       and maplist

filter(_+N):- input(D),length(D,I),min_support(V), N>=(V/100)*I.
head(H+_,H).                        % filter and head are for pattern matching

% an arbitrary subset of the set containing given number of elements.
comb(0,_,[]).
comb(N,[X|T],[X|Comb]):-N>0,N1 is N-1,comb(N1,T,Comb).
comb(N,[_|T],Comb):-N>0,comb(N,T,Comb).

allComb(N,I,Ans):-
   setof(L,comb(N,I,L),Ans).    % setof is a second-order predicate

countSubset(A,[],0).
countSubset(A,[B|X],N):-not(subset(A,B)),countSubset(A,X,N).
countSubset(A,[B|X],N):-subset(A,B),countSubset(A,X,N1),N is N1+1.

countSS(SL,S,S+N):-countSubset(S,SL,N).
len(N,X):-length(X,N).
```

Fig. 3 Frequent-pattern mining implemented with a higher-order logic approach

```
[tumor-size=20-24, inv-nodes=0-2] =>[irradiate=no]                          conf:(1)
[age=50-59, inv-nodes=0-2, node-caps=no] =>[irradiate=no]                   conf:(1)
[age=50-59, menopause=ge40, node-caps=no] =>[irradiate=no]                  conf:(1)
[menopause=ge40, inv-nodes=0-2, deg-malig=1]=>[irradiate=no]                conf:(1)
...
[node-caps=yes, breast=left, recurrence=yes]=>[irradiate=yes]               conf:(0.63)
[breast-quad=left_low, recurrence=yes]=>[irradiate=yes]                     conf:(0.5)
[menopause=premeno, node-caps=yes]=>[irradiate=yes]                         conf:(0.5)
[node-caps=yes, breast=left]=>[irradiate=yes]                               conf:(0.5)
...
[tumor-size=10-14, node-caps=no]=>[recurrence=no]                           conf:(1)
[tumor-size=10-14, irradiate=no]=>[recurrence=no]                           conf:(1)
[tumor-size=10-14, inv-nodes=0-2]=>[recurrence=no]                          conf:(1)
...
[node-caps=yes, breast=left, irradiate=yes]=>[recurrence=yes]               conf:(0.91)
[menopause=premeno, node-caps=yes, irradiate=no]=>[recurrence=yes] conf:(0.83)
[menopause=premeno, node-caps=yes, breast=left]=>[recurrence=yes] conf:(0.83)
[menopause=premeno, node-caps=yes, deg-malig=3]=>[recurrence=yes] conf:(0.79)
[breast=left, breast-quad=left_low, irradiate=yes]=>[recurrence=yes]        conf:(0.77)
```

Fig. 4 Some part of a breast-cancer frequent pattern mining result

Frequent pattern mining is the discovery of 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 $DB = \{ C_1, C_2, C_3, ..., C_n\}$ be a database of n cases or observations and each case contains items in I. A *pattern* is a set of items that occur in a case. The number of items in a pattern is called the length of the pattern. To search for all valid patterns of length 1 up to m in large database is computational expensive. For a set I of m different items, the search space for all distinct patterns can be as huge as 2^m-1. To reduce the size of the search space, the *support* measurement has been introduced [1]. The function *support*(P) of a pattern P is defined as a number of cases in DB containing P. Thus, $support(P) = |\{T \mid T \in DB,\ P \subseteq T\}|$. 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. The *minS* metric has an anti-monotone property and is applied as a basis for reducing search space of mining frequent patterns in algorithm Apriori [1].

We have tested our implementation with the breast cancer dataset taken from the UCI repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). This medical data set is a collection of 191 observations on follow-up patients examining on a recurrence of breast cancer. Each patient's record contains ten attributes: age, menopause, tumor-size, inv-nodes, node-caps, deg-malignant, breast (left, right), breast-quad, irradiate (yes, no), recurrence (yes, no). We run frequent-pattern mining with minimum support 0.001 and show some of the discovered association rules in figure 4. Each rule is attached with the confidence value as a metric to evaluate accuracy of the induced rule. For the rule $A => B$, confidence is

computed from proportion of *support(A&B)* to *support(A)*. The confidence value 1.0 thus implies a 100% accurate association rule.

The mining results shown in figure 4 are in the form of implication. Rule interpretation is straightforward. Taking the last rule in the figure as an example, it can be interpreted as "a patient who has cancer at her left breast in left lower quadrant position and had been treated with radiation implies that she will have a 77-percent chance of cancer recurrence." During the process of searching for frequent patterns, the support values of frequently occurred patterns can also be out put to quantify supporting evidence of the induced association rules.

5 Conclusions and Discussion

We have proposed the design of SUT-Miner, a knowledge-mining system. The system is intended to support knowledge acquisition in medical data and other domains that require new knowledge to support better decisions. The proposed system is an environment for knowledge discovery composing of tools and methods suitable for various kinds of mining such as data classification, regression, clustering, association mining. The intelligence of the system is supported by ontology technology to provide semantics for mining technique selection as well as for knowledge integration at a post-data mining step.

The implementation of the proposed system has been done based on the concept of higher-order logic which is an extension of the well-known first-order logic. With the expressive power of higher-order logic, program coding of the designed system is very concise as demonstrated in the paper. Program conciseness directly contributes to program verification and validation which are important issues in software engineering. The closed form of higher-order logic also supports constraint mining and higher-order mining, i.e. mining from previously discovered knowledge. Our future research is to extend the design of SUT-Miner to facilitate constraint and higher-order mining. Implementing a mechanism for knowledge inferring is also one of our research plans.

Acknowledgments. This work has been fully supported by research fund from Suranaree University of Technology (SUT) granted to the Data Engineering and Knowledge Discovery research unit. This research is also partly supported by grants from the National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF) under grant number RMU 5080026.

References

[1] Agrawal, R., Srikant, R.: Fast algorithm for mining association rules. In: Proc. VLDB, pp. 487–499 (1994)

[2] Alavi, M., Leidner, D.E.: Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues. MIS Quarterly 25(1), 107–136 (2001)

[3] Bojarczuk, C.C., Lopes, H.S., Freitas, A.A., et al.: A constrained-syntax genetic programming system for discovering classification rules: Application to medical data sets. Artificial Intelligence in Medicine 30, 27–48 (2004)

[4] Bratsas, C., Koutkias, V., Kaimakamis, E., et al.: KnowBaSIGS-M: An ontology-based system for semantic management of medical problems and computerised algorithmic solutions. Computer Methods and Programs in Biomedicine 83, 39–51 (2007)
[5] Correia, R., Kon, F., Kon, R.: Borboleta: A mobile telehealth system for primary homecare. In: Proc. ACM Symposium on Applied Computing, pp. 1343–1347 (2008)
[6] De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proc. KDD, pp. 204–212 (2008)
[7] Ghazavi, S., Liao, T.W.: Medical data mining by fuzzy modeling with selected features. Artificial Intelligence in Medicine 43(3), 195–206 (2008)
[8] Hristovski, D., Peterlin, B., Mitchell, J.A., et al.: Using literature-based discovery to identify disease candidate genes. Int. J. Medical Informatics 74, 289–298 (2005)
[9] Huang, M.J., Chen, M.Y., Lee, S.C.: Integrating data mining with case-based reasoning for chronic diseases prognosis and diagnosis. Expert Systems with Applications 32, 856–867 (2007)
[10] Hulse, N.C., Fiol, G.D., Bradshaw, R.L., et al.: Towards an on-demand peer feedback system for a clinical knowledge base: A case study with order sets. J. Biomedical Informatics 41, 152–164 (2008)
[11] Kakabadse, N.K., Kouzmin, A., Kakabadse, A.: From tacit knowledge to knowledge management: Leveraging invisible assets. Knowledge and Process Management 8(3), 137–154 (2001)
[12] Nadathur, G., Miller, D.: Higher-order Horn clauses. J. ACM 37, 777–814 (1990)
[13] Nguyen, D., Ho, T., Kawasaki, S.: Knowledge visualization in hepatitis study. In: Proc. Asia-Pacific Symposium on Information Visualization, pp. 59–62 (2006)
[14] Palaniappan, S., Ling, C.S.: Clinical decision support using OLAP with data mining. Int. J. Computer Science and Network Security 8(9), 290–296 (2008)
[15] Roddick, J.F., Fule, P., Graco, W.J.: Exploratory medical knowledge discovery: experiences and issues. ACM SIGKDD Explorations Newsletter 5(1), 94–99 (2003)
[16] Roddick, J.F., Spiliopoulou, M., Lister, D., et al.: Higher order mining. ACM SIGKDD Explorations Newsletter 10(1), 5–17 (2008)
[17] Ruppel, C.P., Harrington, S.J.: Sharing knowledge through intranets: A study of organizational culture and intranet implementation. IEEE Transactions on Professional Communication 44(1), 37–51 (2001)
[18] Sahama, T.R., Croll, P.R.: A data warehouse architecture for clinical data warehousing. In: Proc. 12th Australasian Symposium on ACSW Frontiers, pp. 227–232 (2007)
[19] Satyadas, A., Harigopal, U., Cassaigne, N.P.: Knowledge management tutorial: An editorial overview. IEEE Transactions on Systems, Man and Cybernetics, Part C 31(4), 429–437 (2001)
[20] Shillabeer, A., Roddick, J.F.: Establishing a lineage for medical knowledge discovery. In: Proc. 6th Australasian Conf. on Data Mining and Analytics, pp. 29–37 (2007)
[21] Thongkam, J., Xu, G., Zhang, Y., et al.: Breast cancer survivability via AdaBoost algorithms. In: Proc. 2nd Australasian Workshop on Health Data and Knowledge Management, pp. 55–64 (2008)
[22] Truemper, K.: Design of logic-based intelligent systems. John Wiley & Sons, New Jersey (2004)
[23] Uramoto, N., Matsuzawa, H., Nagano, T., et al.: A text-mining system for knowledge discovery from biomedical documents. IBM Systems J. 43(3), 516–533 (2004)
[24] Zhou, X., Liu, B., Wu, Z.: Text mining for clinical Chinese herbal medical knowledge discovery. In: Discovery Science 8th Int. Conf., pp. 396–398 (2005)
[25] Zhuang, Z.Y., Churilov, L., Burstein, F.: Combining data mining and case-based reasoning for intelligent decision support for pathology ordering by general practitioners. European J. Operational Research 195(3), 662–675 (2009) doi: 10.1016/j.ejor.2007.11.003

Improving Medical Database Consistency with Induced Trigger Rules

Nittaya Kerdprasop, Sirikanjana Pilabutr, and Kittisak Kerdprasop

Abstract. The concept of triggers has been around for more than two decades. Despite their diverse potential usages, trigger rules are difficult to define correctly and have to be carefully hand-coded by database programmers. We suggest an automatic way of trigger rule creation by the advanced technology of data mining. We propose a framework of trigger rule induction as well as a method for trigger conflict resolution. On trigger firing the problem may arise if several trigger rules are eligible for execution. We propose a conflict resolution scheme that incorporates derived knowledge as a major part of the trigger rule prioritization. By means of trigger scheduling, deterministic behavior of the trigger processing can be guaranteed. We demonstrate the utilization of our proposed method on enhancing medical database consistency.

1 Introduction

A database is a collection of objects such as patient records together with a set of integrity constraints on these objects. Integrity constraints are predicates defined by the database designer as a requirement for database to be true on any database state. The values of objects in the database at any given time determine the state of the database. The state changes if there is a modification in the value of a database object. A database state is *consistent* if the values of the objects satisfy the specified integrity constraints. Database consistency is an important property to guarantee its reliability on any application.

To prevent the database from being inconsistent, data objects have to be accessed and modified only through the transactions. A transaction is a set of operations such as INSERT, DELETE, UPDATE that causes the database to change from one consistent state to another. On transaction processing, integrity constraints pertaining to the transaction are evaluated. If the constraints evaluate to false, called constraint violation, then the transaction that causes this event is undone. Integrity constraints are, however, capable of ensuring simple events such as domain integrity, referential integrity. To impose complex enforcement such as

Nittaya Kerdprasop, Sirikanjana Pilabutr, and Kittisak Kerdprasop
Data Engineering and Knowledge Discovery (DEKD) Research Unit,
School of Computer Engineering, Suranaree University of Technology,
Nakhon Ratchasima 30000, Thailand

K. Nakamatsu et al. (Eds.): New Advan. in Intel. Decision Techno., SCI 199, pp. 265–274.
springerlink.com

business rules or complicated update constraints across applications, trigger rules are deployed as a powerful and expressive tool to enforce integrity checking and thus enabling the filtering of state changes that violate database consistency.

Triggers, also known as event-condition-action (ECA) rules [26], are one major concept of active databases which extend traditional database systems with the mechanism to respond automatically to some specific events. The events may take place either inside or outside the database system. Upon the occurrence of the specified event, the rule condition is evaluated. If the condition is satisfied, then some actions are performed.

Although triggers are regarded as an important database feature on consistency monitoring, their deployment is still limited. This is due to the fact that creating complex trigger rules is not an easy task [9, 17, 19]. Tools and environments to aid users and database programmers are certainly needed. It is thus our aim to provide a method to automatically generating trigger rules from current database contents by means of data mining techniques. The induced trigger rules can be viewed as supplementary constraints to help increasing database consistency.

This paper is organized as follows. After the introduction section, we review some background on triggers and their related issues. Section 3 is the proposed framework of inducing trigger rules from database contents and its application to medical database. We also provide an algorithm to solve trigger conflict problem together with detailed explanation of the algorithm in section 4. Section 5 discusses other work related to ours. Section 6 concludes the paper.

2 Triggers and Related Issues

In SQL standard [11, 14], triggers are expressed by means of event-condition-action rules, as presented in figure 1. Each trigger is identified by a name. It is possible to specify whether a trigger must be executed BEFORE or AFTER its triggering event. SQL triggers allow only the INSERT, DELETE, and UPDATE as triggering event, and limit to a single event be monitored per single trigger rule.

```
<trigger definition> ::= CREATE TRIGGER <trigger name>
                         { BEFORE | AFTER } <trigger event> ON <table>
                         [ REFERENCING <tran_table or var_list> ]
                         <triggered action>
<trigger event>   ::= INSERT | DELETE | UPDATE [ OF < column list> ]
<triggered action> ::= [ FOR EACH { ROW | STATEMENT } ]
                       [ WHEN ( <condition> ) ] <triggered SQL statement>
```

Fig. 1 Definition of SQL triggers

The WHEN clause specifies an additional condition to be checked once the trigger rule is fired and before the action is executed. Conditions are predicates over the database state. If the WHEN clause is missing, the condition is supposed to be true and the trigger action is executed as soon as the trigger event occurs. The action is executed when the rule is triggered and its condition is true. Actions are

stored procedures and may include SQL statements, control constructs, and calls to user-defined functions. The following example shows a trigger rule to impose a constraint on the database that the age of any person may never decrease.

Example 1: Trigger rule to guarantee no decrease on age value.

```
CREATE TRIGGER age-no-decrease BEFORE UPDATE OF Patient
FOR EACH ROW
  WHEN (new.Age < old.Age) begin log the event; signal error condition; end
```

The potential applications of triggers are significant [9, 19, 23]: signal integrity constraint violation and force rollbacks of the violating transactions, maintain consistency across system catalogs or other metadata, notify users in the form of messages, implement business rules or workflow management, and many more.

The behavior of triggers is defined as the "execution model." It specifies how trigger rules are evaluated and treated at runtime. Figure 2 illustrates the steps in processing triggers [22]. The signaling phase detects and signals the occurrence of an event. The event activates the corresponding trigger rules in the triggering phase, and the condition parts of the triggered rules are evaluated in the evaluation phase. The trigger conflict problem occurs when the conditions of more than one trigger rules are evaluated to be true. The scheduling phase indicates the order to process conflict triggers. The execution phase processes the scheduled trigger rules. On processing rule's action, the change in a database state may trigger another or even the same set of rules.

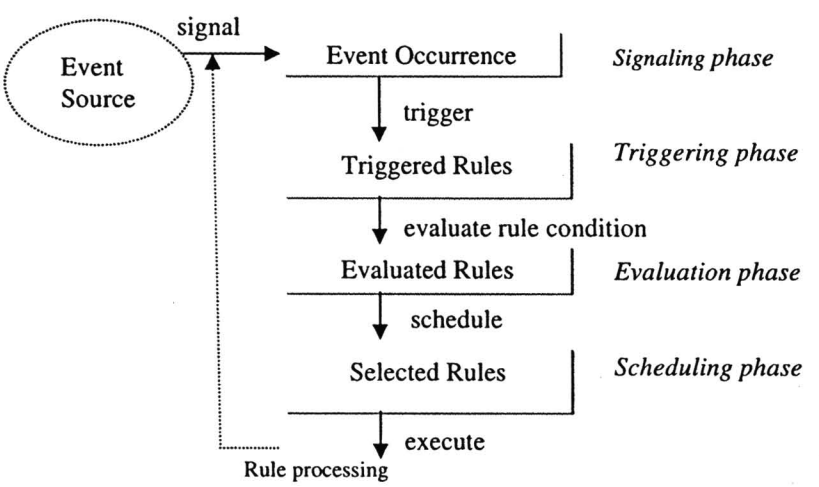


Fig. 2 Trigger rule execution steps

After the evaluation phase, more than one rule may be eligible for execution. This problem is known as *trigger rule conflict* [4, 22]. To solve the problem, the database management system must provide a *conflict resolution policy* to select a

trigger rule for execution. The common conflict resolution policy adopted by most systems is assigning rule priority [21, 26]. The rule prioritization is either assigning a high priority to the rule that is most recently fired, or setting priority on the specificity of the rule. The two approaches are dynamic, thus less practical in the system with large and complex trigger rule set. When deterministic behavior is highly desirable, the scheme to associate rules with priority statically is more appropriate. Static priority mechanism determines order of trigger rules either by the system (e.g., based on rule creation time) or by the user (e.g., explicitly associate each rule with a numeric value). We propose a conflict resolution mechanism (in section 4) to incorporate derived knowledge (i.e., the knowledge obtained from the database content) into the rule prioritization scheme.

3 The Trigger Induction Framework and Its Application

We design the framework to add active behavior to the medical database through the induced trigger rules and rule processing module as shown in figure 3. There are three major components in our model: mining, trigger generation and conflict resolution components. Mining component induces knowledge in form of rules, association and classification, from the database contents. The data repository contains both base data and trigger rules. Trigger generation component is responsible for converting induced classification/association rules into trigger format then stores generated triggers in the repository. In case of trigger rule application and rule conflict occurs, conflict resolution component will handle the situation. We demonstrate the application of our proposed framework towards database consistency enforcement through example 2.

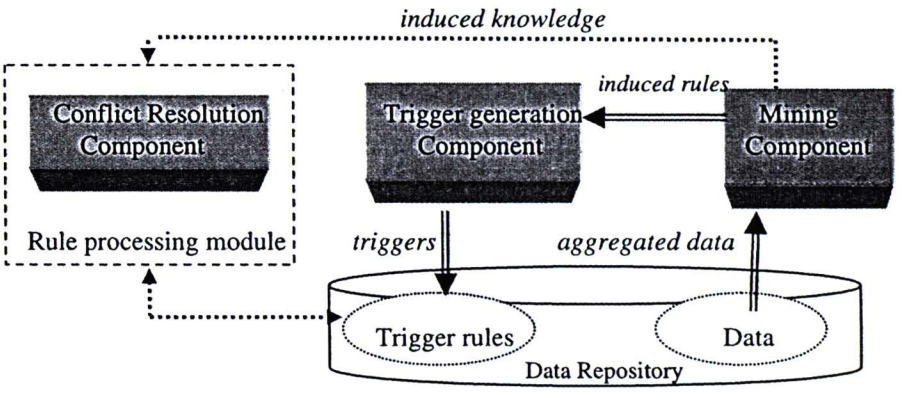


Fig. 3 A framework of active medical database containing trigger rule induction and trigger conflict resolution components

Example 2: Trigger induction on Diabetes database.

For a given medical database containing base tables related to patient personal information and treatment records, the aggregated information collected from related base tables with selected features suitable for mining component is shown schematically as follows.

Diabetes (Patient_ID, Name, Sex, Age, Temperature, Blood_Pressure_Upper, Blood_Pressure_Low, Diabetes_family, Weight, Height, BMI, Blood_Sugar, Diabetes, Insulin_level)

These data are input to the mining component to induce the following classification rules:

1. If Diabetes_family=yes and BMI>24.9 Then Diabetes=yes
2. If Diabetes_family=no and blood_sugar>128 Then Diabetes=yes

The trigger generation component then converts these rules into SQL triggers.

```
CREATE TRIGGER rule_1 ON diabetes FOR UPDATE, INSERT
    AS IF (SELECT COUNT(*) FROM diabetes
WHERE (diabetes_family = 'yes') and (BMI > 24.9) and (diabetes <> 'yes') ) > 0
BEGIN ROLLBACK TRAN; RAISERROR ('diagnose error'); END

CREATE TRIGGER rule_2 ON diabetes FOR UPDATE, INSERT
    AS IF (SELECT COUNT(*) FROM diabetes
WHERE (diabetes_family = 'no') and (blood_sugar > 128) and (diabetes <> 'yes')) > 0
BEGIN ROLLBACK TRAN; RAISERROR ('diagnose error'); END
```

Suppose there is an attempt to insert the following information into the database.

```
INSERT INTO Diabetics (Patient_ID, Name, Sex, Age, Temperature,
        Blood_Pressure_Upper, Blood_Pressure_Low, Diabetes_family, Weight,
        Height, BMI, Blood_Sugar, Diabetes, Insulin_level)

Values(P021, Amitta, Female, 33, 37.6, 130, 80, no, 72, 1.62, 27.4, 130, no, 0)
```

This new information violates Trigger rule_2 since *Diabetes_family = 'no'* and *Blood_Sugar = 130*, but the diagnosed *Diabetes = 'no'*. Therefore, this transaction has to be undone and the database remains in a consistent state.

The proposed framework is semi-automatic in that the trigger induction process has to be invoked by the database administrator. After database contents have been created and modified and numerous new contents might have been inserted into the database, the administrator may consider activating the trigger induction process. Upon activation the previous induced trigger rules are removed as they may not be relevant to the current database state. The mining component has been set to induce only rules with 100% accuracy rate since precision is critical criteria in medical domain. Induced rules are prioritized in descending order of

their support (or coverage) values. The *top-k* rules will be sent to trigger generation component; the *k* value is adjustable by the database administrator. At the moment we do not consider the issue of rule conflict since we assume that database integrity constraints are powerful mechanism sufficient to prevent conflicting cases inserted into database contents.

4 A Method for Trigger Conflict Resolution

After triggers have been created, the problem of trigger conflict might occur during trigger processing phase. In this section, we define an algorithm (in figure 4) to handle trigger conflict by reorganizing the trigger rules into different layers, or strata. Then, associate each stratum with a numeric priority. The major mechanism leading to priority assigning is the induced knowledge regarding the database state modification.

Input: an unordered stratum set S, a database R and its metadata
Output: an ordered stratum set O with assigned priority on each stratum
Steps:

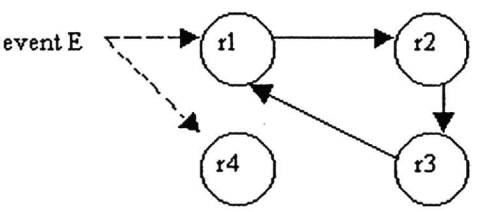
1. ***Active_Tuple_Set*$_i$ ← *Activate*(S_i , *E*)**
   /* Activate every stratum S_i , $S_i \in$ S, such that the occurrence of an event *E* can invoke its trigger rule(s), and record all affected tuples in the corresponding *Active_Tuple_Set*. */
2. **K_i ← *Induce*(*Active_Tuple_Set*$_i$)**
   /* The knowledge induction method (such as rule learning, decision-tree induction) is applied to induce knowledge from the content of each *Active_Tuple_Set*. The induced knowledge is stored in K_i. */
3. **q_i = *Degree_of_Constraint* (K_i , *metadata*)**
   /* Calculate the value *q*, or the *Degree_of_Constraint*, of each set of induced knowledge *K* comparing to the integrity constraints given as a metadata. */
4. ***sort*(*i*, *q*)**
   /* Apply any sorting algorithm on the *q*-value associated with each stratum S_i. */
5. ***return an ordered stratum set O = { S_i | S has been sorted by its index i }***

Fig. 4 Trigger conflict resolution algorithm

We have applied the concept of stratum [3] to guarantee that trigger rule execution eventually terminates. A *stratum* is an ordered set of trigger rules that locally converge. A stratum locally converges if after any transaction invoking trigger rules, rule processing terminates in a final state in which the set of triggered rules

is empty. *Stratum set* is an unordered set of strata in which each stratum is independent from other strata so that the trigger rule execution in one stratum does not affect rules in other strata. The example of non-terminate trigger execution due to the cycle in action-triggering events is shown in figure 5. Breaking the cycle into different layers, depicted in figure 6, and the local convergence within each stratum are two sufficient conditions for termination on trigger execution.

Fig. 5 The cycle among trigger rules



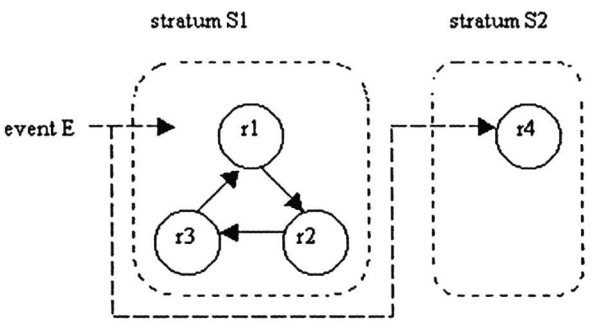


Fig. 6 Organizing cyclic rules into strata

Step 3 of the proposed algorithm is to set priorities among strata when several strata are activated by the occurrence of an event. We propose the trigger conflict resolution algorithm to solve the problem of multiple stratum activation on an event E. The key concept is to apply, in step 2, knowledge induced from the database content and the metadata (i.e., the set of integrity constraints) to guide the priority assigning scheme. The function to compute priority for each stratum is defined as:

$$Degree\_of\_Constraint(\mathrm{K}, \mathrm{IC}) = \frac{\#matched\_rules}{\#total\_rules} + \alpha$$

where *# matched_rule* is the number of induced knowledge, represented in the format of rule, completely matching with the integrity constraint rule (i.e., NOT (k) AND c yields the contradiction when $k \in K$ and $c \in IC$),
 # total_rules is the total number of induced knowledge represented as rules,

α is the average accuracy of the induced knowledge rules normalized in order to prevent the domination of the accuracy over the proportion of *matched_rules* and *total_rules*.

Step 4 of the algorithm groups triggers into several strata, then each stratum will be sorted according to the *Degree_of_Constraint* values. At a final step, each stratum has been returned with the priority value associated with it.

5 Related Work

The importance of integrating active behavior into the database systems has been recognized since the 1970s [12]. However, it was not until the late 1980s to early 1990s that the area of active databases has caught high interest among researchers [8, 10, 13, 22, 26]. At present most commercial database systems, such as Oracle, IBM DB2, Microsoft SQL Server, support the simple forms of triggers and also incorporate triggers into commercial object-oriented databases [7, 21]. The SQL standard [11, 14] has extensive coverage of triggers.

A simple concept of triggers has been successfully applied to solve problems in various domains [9]. For example, Sengupta et al [23] employ triggers to alert intrusion detection system administrator when a suspect event has been detected. In medical domain the employment of triggers to achieve active behavior is quite rare. Most of the proposed methods are for detecting static events such as the discovery of relationships that suggest risks of adverse events in patient records [20, 24], detection of dependency patterns of process sequences for curing brain stroke patients [18], the generation of rules to annotated protein data in medical database [16], or the exploration of environmental health data [6]. Our work differ from those appeared in the literature in that we propose a framework of employing knowledge discovery techniques to automatically create trigger rules and also prioritize rules in case of conflict. The utilization of our proposed method is to increase consistency in medical database. Any database modification events violating constraints will be alerted and undone. The system designed by Agrawal and Johnson [1] is also to support medical database but in a different aspect; they concentrate on security and privacy preservation of patients and other sensitive health data.

Although trigger is a powerful mechanism in active database systems, designing and writing correct trigger rules are not an easy and straightforward task. The difficulty is due to the complex and sometime unpredictable behavior of the triggers. Poorly designed triggers can activate each other indefinitely, which leads to the non-terminate execution. Several methods have been proposed [2, 3, 4, 5, 15, 25] to analyze trigger behavior at compile time and runtime. There exist some work on developing tools to aid trigger designing semantically [19] and visually [17]. Another problem regarding trigger behavior is the deterministic property of the triggers. Deterministic trigger processing guarantees the same order of execution when several trigger rules are activated simultaneously. We propose a mechanism to utilize domain-knowledge in choosing among activated triggered rules.

6 Conclusions

We present the design framework of active medical databases. Active behavior of the database system has been obtained through a set of trigger rules. Triggers are both created manually by database programmers and induced automatically via data mining techniques. We devise a method to induce trigger rules from existing database contents. We also propose a method to solve trigger conflict problem.

The trigger rule conflict occurs when an event activates several trigger rules simultaneously. To maintain the deterministic property of the active database processing, the database management system has to provide a conflict resolution policy. The common policy adopted by most systems is to assign rule priority. The rule prioritization is based on either the recent update or the complexity of the rule's condition. We propose a different scheme of prioritization by taking into account the knowledge regarding the database state. Moreover, we consider priority at the level of stratum, which may contain several related triggers. The concept of stratification preserves the termination property of trigger rule processing.

The design of medical active database framework and the trigger-conflict-resolution algorithm is the preliminary work toward the design and implementation of a set of tools to help database designer on designing and analyzing a complex set of triggers. A further investigation on a more practical active database with a larger trigger set is necessary.

Acknowledgments. The authors would like to thank all anonymous referees for their thorough reading and very helpful suggestion. This work has been fully supported by research fund from Suranaree University of Technology granted to the Data Engineering and Knowledge Discovery (DEKD) research unit. This research is also partly supported by grants from the National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF) under grant number RMU 5080026.

References

[1] Agrawal, R., Johnson, C.: Securing electronic health records without impeding the flow of information. Int. J. Medical Informatics 76, 471–479 (2007)

[2] Aiken, A., Hellerstein, J.M., Widom, J.: Static analysis techniques for predicting the behavior of active database rules. ACM Transactions on Database Systems 20(1), 3–41 (1995)

[3] Baralis, E., Ceri, S., Paraboschi, S.: Modularization techniques for active rule design. ACM Transactions on Database Systems 21(1), 1–29 (1996)

[4] Baralis, E., Ceri, S., Paraboschi, S.: Compile-time and runtime analysis of active behaviors. IEEE Transactions on Knowledge and Data Engineering 10(3), 353–370 (1998)

[5] Baralis, E., Widom, J.: An algebraic approach to rule analysis in expert database system. In: Proc. 20th VLDB, pp. 475–486 (1994)

[6] Bedard, Y., Gosselin, P., Rivest, S., et al.: Integrating GIS components with knowledge discovery technology for environmental health decision support. Int. J. Medical Informatics 70, 79–94 (2003)

[7] Bertino, E., Guerrini, G., Merlo, I.: Trigger inheritance and overriding in an active object database system. IEEE Transactions on Knowledge and Data Engineering 12(4), 588–608 (2000)
[8] Buchmann, A.: Current trends in active databases: Are we solving the right problems. In: Proc. Information Systems Design and Multimedia, pp. 121–133 (1994)
[9] Ceri, S., Cochrane, R.J., Widom, J.: Practical applications of triggers and constraints: Successes and lingering issues. In: Proc. 26th VLDB, pp. 254–262 (2000)
[10] Chakravarthy, S.: Rule management and evaluation: An active DBMS perspective. ACM SIGMOD Records 18(3), 20–28 (1989)
[11] Eisenberg, A., Melton, J.: SQL:1999, formerly known as SQL3. ACM SIGMOD Records 28(1), 131–138 (2000)
[12] Eswaran, K.P.: Specification, implementations and interactions of a trigger subsystem in an integrated database system. IBM Research Report RJ1820, San Jose, California (1976)
[13] Hanson, E.N., Widom, J.: An overview of production rules in database systems. Knowledge Engineering Review 8(2), 121–143 (1993)
[14] International Organization for Standardization, ISO/IEC 9075:2003 (2003)
[15] Karadimce, A.P., Urban, S.D.: Conditional term rewriting as a formal basis for analysis of active database rules. In: Proc. Research Issues in Data Engineering (RIDE), pp. 156–162 (1994)
[16] Kretschmann, E., Fleischmann, W., Apweiler, R.: Automatic rule generation for protein annotation with the C4.5 data mining algorithm applied on SWISS-PROT. Bioinformatics 17(10), 920–926 (2001)
[17] Lee, D., Mao, W., Chiu, H., et al.: Designing triggers with trigger-by-example. Knowledge and Information Systems 7, 110–134 (2005)
[18] Lin, F., Chou, S., Pan, S., et al.: Mining time dependency patterns in clinical pathways. Int. J. Medical Informatics 62(1), 11–25 (2001)
[19] Mota-Herranz, L., Celma-Gimenez, M.: Automatic generation of trigger rules for integrity enforcement in relational databases with view definition. In: Proc. 3rd Int. Conf. Flexible Query Answering Systems, pp. 286–297 (1998)
[20] Noren, G.N., Bate, A., Hopstadius, J., et al.: Temporal pattern discovery for trends and transient effects: Its application to patient records. In: Proc. KDD, pp. 963–971 (2008)
[21] Paton, N.: Active rules in database systems. Springer, Heidelberg (1999)
[22] Paton, N., Diaz, O.: Active database systems. ACM Computing Surveys 31(1), 63–103 (1999)
[23] Sengupta, S., Andriamanalimanana, B., Card, S.W., et al.: Towards data mining temporal patterns for anomaly intrusion detection systems. In: Proc. IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Tech. and Applications, pp. 205–209 (2003)
[24] Silva, A., Cortez, P., Santos, M.F., et al.: Rating organ failure via adverse events using data mining in the intensive care unit. Artificial Intelligence in Medicine 43(3), 179–193 (2008)
[25] van der Voort, L., Siebes, A.: Termination and confluence of rule execution. In: Proc. 2nd Int. Conf. Information and Knowledge Management, pp. 245–255 (1993)
[26] Widom, J., Ceri, S.: Active database systems: Triggers and rules for advanced database processing. Morgan Kaufmann, San Francisco (1996)

IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 2, No 3, March 2010
ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

Database Reverse Engineering based on Association Rule Mining

Nattapon Pannurat[1], Nittaya Kerdprasop[2] and Kittisak Kerdprasop[2,*]

[1] Faculty of Information Sciences, Nakhon Ratchasima College
290 Moo 2, Mitraphap Road, Nakhon Ratchasima, 30000, Thailand

[2] Data Engineering and Knowledge Discovery Research Unit, Suranaree University of Technology
111 University Avenue, Nakhon Ratchasima, 30000, Thailand

Abstract

Maintaining a legacy database is a difficult task especially when system documentation is poor written or even missing. Database reverse engineering is an attempt to recover high-level conceptual design from the existing database instances. In this paper, we propose a technique to discover conceptual schema using the association mining technique. The discovered schema corresponds to the normalization at the third normal form, which is a common practice in many business organizations. Our algorithm also includes the rule filtering heuristic to solve the problem of exponential growth of discovered rules inherited with the association mining technique.

Keywords: *Legacy Databases, Reverse Engineering, Database Design, Database Normalization, Association Mining.*

1. Introduction

Legacy databases are obviously valuable assets to many organizations. These databases were mostly developed with technologies in the 1970s [14] using old programming languages such as COBOL and RPG, and file systems of the mini-computer platforms. Some databases were even designed with the outdated concepts such as hierarchical data model, and thus made them difficult to be maintained and adjusted to serve current needs of modern companies.

One solution to modernize the legacy databases is to migrate and transform their structures and corresponding contents to the new systems. This approach is, however, hard to achieve if the design document of the system does no longer exist, which is the common situation in most enterprises. To solve the problems of recovering database structures and migrating legacy databases, we propose the database reverse engineering methodology.

The process of reverse engineering [7] originally aimed at discovering design and production procedure from devices, end products, or other hardware. This methodology often used in the Second World War for military advantage by copying opponents' technologies. Reverse engineering of software refers to the process of discovering source code and system design from the available software [7].

In database community, reverse engineering is an attempt to extract the domain semantics such as keys, functional dependencies and integrity constraints from the existing database structures [6, 13]. Typically, database reverse engineering is the process of extracting design specifications from legacy systems and making the reverse transformation from logical to conceptual schema [6, 15].

Our work deals with the reverse schema process by making a step further from logical schema to the lower level of database instances. We apply the machine learning technique, association rule mining in particular, to induce dependency relationships among data attributes. The major problem of applying association mining to real-life databases is that it always generates tremendous amount of association rules [11, 12]. We thus include the rule- filtering component in our design to select only promising association rules.

The structure of this paper is organized as follows. Section 2 presents the basic concept and the design framework of our methodology. Section 3 explains the system implementation by means of an example. Section 4

* Corresponding author

discusses related work. Finally, Section 5 concludes the paper.

2. Database Reverse Engineering with NoWARs

The objective of our system is to induce conceptual schema from the database instances with the basic assumption that database design documents are absent. We apply the normalization principles and the association mining technique to discover the missing database design.

Normalization [8] is the process to transform unstructured relation into separate relations, called normalized ones. The main purpose of this separation is to eliminate redundant data and reduce data anomaly (insert, update, and delete). There are many different levels of normalization depending on the purpose of database designer. Most database applications are designed to be in the third and the Boyce-Codd normal forms in which their dependency relations [3] are sufficient for most organizational requirements. Figure 1 [9] illustrates the refinement steps from un-normalized relations to the relations in fifth normal form.

The main condition to transform from one normal form to the next level is the dependency relationship, which is a constraint between two sets of attributes in a relation. Experienced database designers are able to elicit this kind of information. But in the reverse engineering process in which the business process and operational requirements are unknown, this task of dependency analysis is tough even for the experienced ones. We thus propose to use the machine learning technique called association mining.

Association mining searches for interesting relationships among a large set of data items [1, 2]. The discovery of interesting association relationships among huge amounts of business transaction records can help in many business decision making process, such as catalog design, cross-marketing, and loose-leader analysis [11]. An example of association rule mining is market basket analysis. This process analyzes customer buying habits by finding association the different items that customers place in their shopping baskets. For example, customers are buying milk also tend to buy bread and water drinking at the same time. These can represent in association rule as follows.

milk [support=5%] → bread, water drinking [support=5%] [confidence=100%]

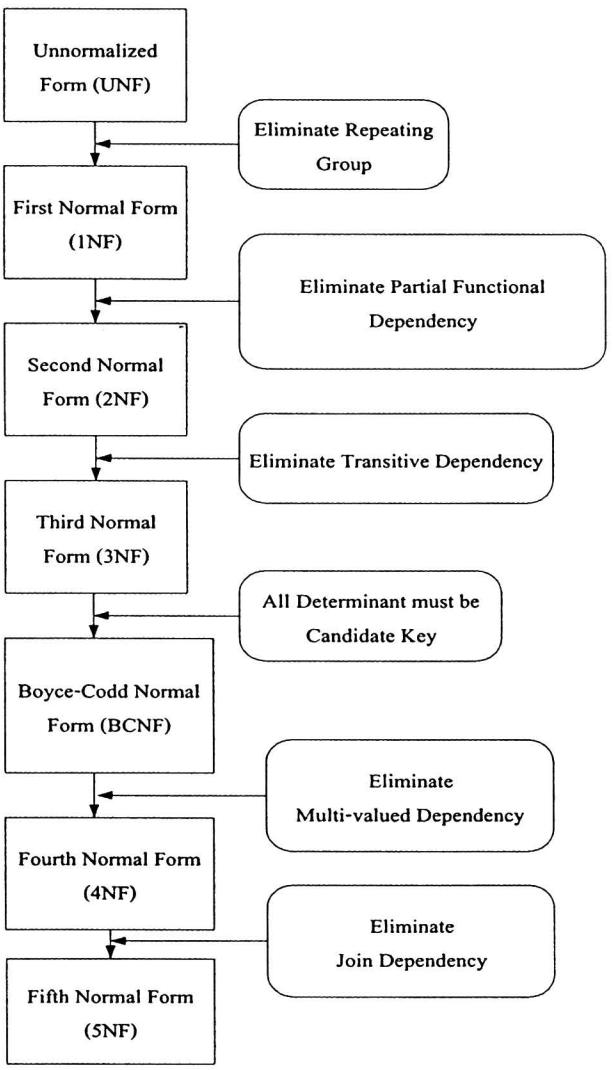


Fig. 1 Normalization steps.

A support of 5% for association rule means that 5% of all the transactions under analysis show that milk, bread, and water drinking are purchased together. A confidence of 100% means that 100% of the customers who purchased some milk also bought bread and water drinking.

Our methodology of database reverse engineering composes of designing and improving the process of normalization with association analysis technique. We use the normalization concept and association analysis technique to create a new algorithm called NoWARs (Normalization With Association Rules).

NoWARs is an algorithm that combines normalization process and association mining technique together. We can find association rules by taking the dataset on a database and feeding into a data mining process. We use

Apriori algorithm to find association rules. NoWARs has two important steps, first finding association rules and second normalization with rules obtained from the first step. The details of NoWARs algorithm are shown in Figure 2 and its workflow are shown in Figure 3.

Algorithm: NoWARs

Input: Database instances in table format
Output: Relational schema

(1) Set Minimum Confidence value = 1.0
(2) Set Minimum Support value = 0.0 and Maximum Support value = 1.0
(3) Call Apriori to obtain association rules
(4) For every rule from i to n do
(5) Select one cause rule with max support value
(6) Select primary key of relation (2NF)
(7) Find corresponding attribute
(8) Select primary key of relation (3NF)
(9) Compare non-key attribute in (8) with all attributes in (5) then cut it off and we get one 3NF relation
(10) Compare all non -key attributes with 1NF relation and cut it off
(11) Combine all primary keys of 2NF with the remaining attribute in (10) to become the 3NF

Fig. 2 NoWARs algorithm.

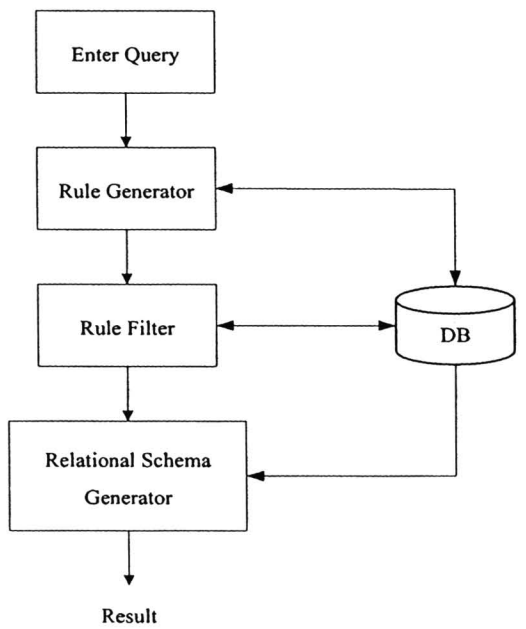


Fig. 3 The work flow of NoWARs algorihm.

3. Implementation

The algorithm NoWARs starts when user enter query to define dataset to normalize. Then NoWARs will find the association rules by calling Apriori algorithm and save resulting a form of association rules in the database. Then NoWARs will select some rules to use in normalization process. Finally, use the selected rules to generate the 3NF table in relational schema form. The input of NoWARs is the un-normalize table. The example of input data format is shown in Table 1.

Table 1: Example of input data.

INV	*DATE*	*C_ID*	*P_ID*	*P_Name*	*QTY*
001	9/1/2010	C01	P01	Printer	3
001	9/1/2010	C01	P02	Phone	5
002	9/1/2010	C03	P05	TV	6
002	9/1/2010	C03	P04	Lamp	2
:	:	:	:	:	:

The un-normalized data as shown in Table 1 will be analyzed by the algorithm, and then its schema in 3NF is generated. We perform experimentation with five datasets as shown in Table 2. We use Oracle Database 10g XE Edition, tested on Pentium IV 3.0 GHz with RAM 512 MB machine.

Table 2: Number of records and attributes in experimental datasets.

Dataset Name	*Number of Records*	*Number of attributes*
Register	12438	157
Video_Rental	483478	523
Data_Org	91845	334
Invoice	119795	123
Car_Color	199337	312

We take the Register dataset as a running example. This dataset is originally un-normalized and its structure is as follows.

Register (STUDENT_CODE, STUDENT_NAME,
TEACHER_CODE, TEACHER_NAME,
UNIT, SUBJECT_CODE, SUBJECT_NAME)

After execution, its conceptual schema is recovered as shown in Figure 4. The performance of rule-filtering also analyzed and shown in Figure 5.

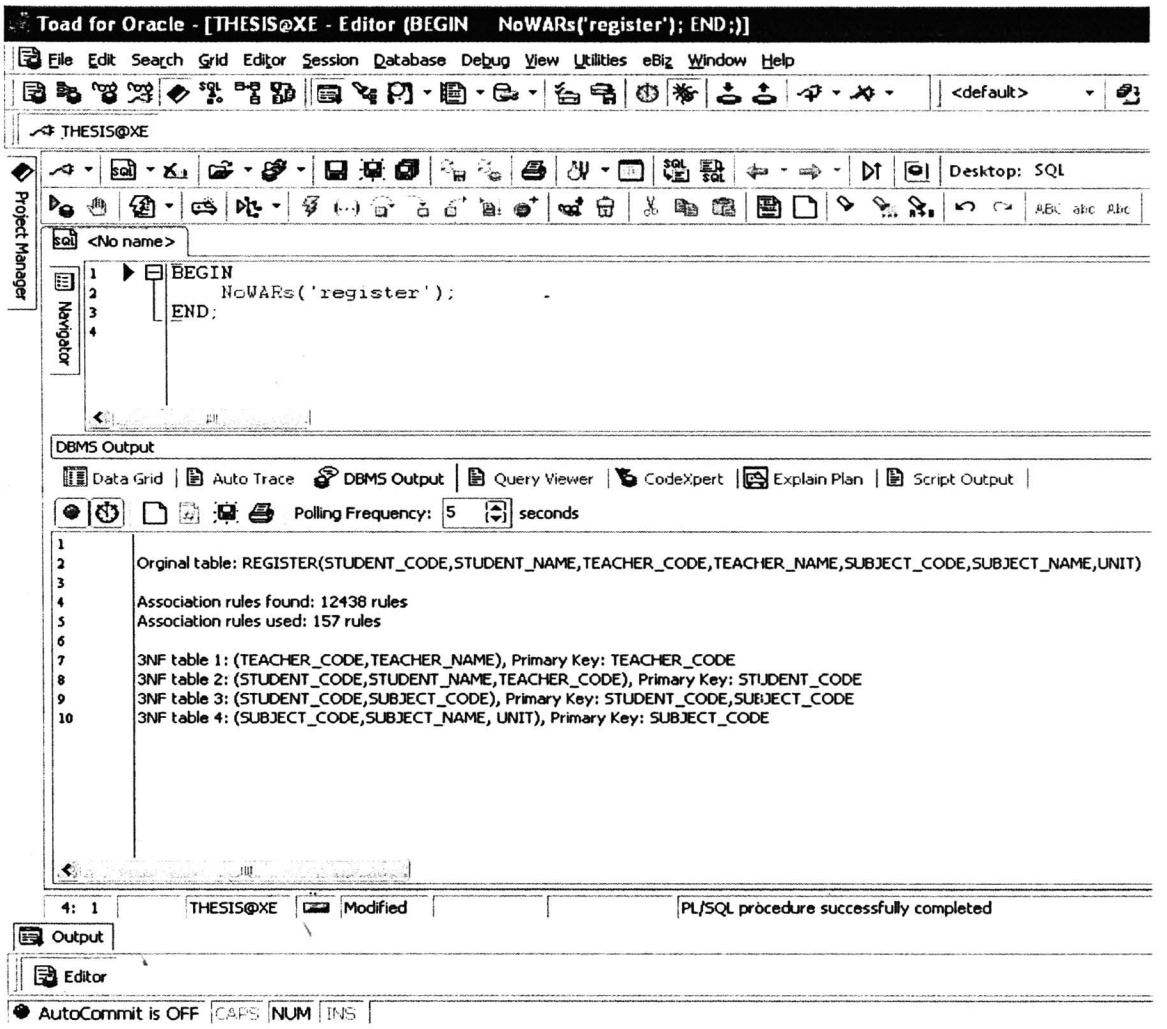


Fig. 4 The result of running NoWARs algorihm on Register dataset

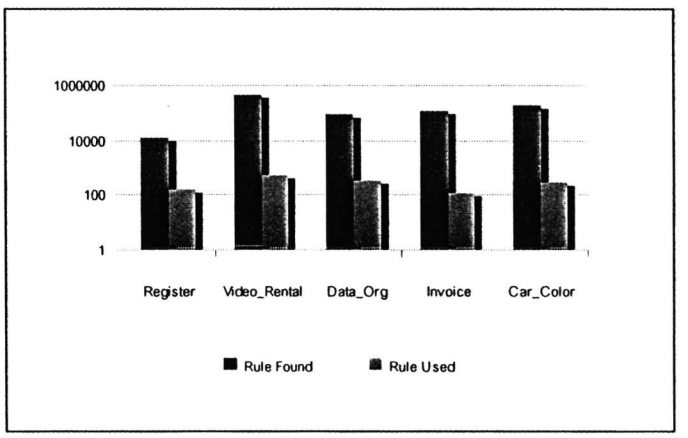


Fig. 5 Performance of rule-filtering component of NoWARs algorithm in reducing number of association rules

4. Related Work

Since the introduction of a famous association technique known as Apriori algorithm [1, 2], there have long been immense attempts to integrate this technique to improve database design, consistency checking, and querying. Han et al. [10] improved the DBMiner system to work with relational databases and data warehouses. DBMiner can do many data mining tasks such as classification, prediction and association. Sreenath, Bodagala, Alsabti, and Ranka [16] adopted Apriori algorithm to work with relational database system. They created Fast UPdate algorithm to search association data when the system has new transaction. Tsechansky, Pliskin, Rabinowitz and Porath [17] applied Apriori to find association data from many relations in the database. Berzal, Cubero, Marín and Serrano [4] used Tree-Based Association Rule mining (TBAR) to find association data in relational database. They kept large item set in tree structure format to reduce time cost in association process. Hipp, Güntzer and Grimmer [12] implemented Apriori algorithm with C++ programming language to work on DB2 database system. They used the program to find association data in Daimler-Chrysler Company database.

In parallel to the attempts of applying learning techniques to existing large databases, researchers in the area of database reverse engineering have proposed some means of extracting conceptual schema. Lee and Yoo [14] proposed a method to derive a conceptual model from object-oriented databases. The derivation process is based on forms including business forms and forms for database interaction in the user interface. The final products of their method are the object model and the scenario diagram describing a sequence of operations. The work of Perez et al. [15] emphasized on relational object-oriented conceptual schema extraction. Their reverse engineering technique is based on a formal method of term rewriting. They use terms to represent relational and object-oriented schemas. Term rewriting rules are then generated to represent the correspondences between relational and object-oriented elements. Output of the system is the source code to migrate legacy database to the new system.

Recent work in database reverse engineering has not concentrated on a broad objective of system migration. Researchers rather focus their study on a particular issue of semantic understanding. Lammari et al. [13] proposed a reverse engineering method to discover inter-relational constraints and inheritances embedded in a relational database. Chen et al. [5] also based their study on entity-relationship model. They proposed to apply association rule mining to discover new concepts leading to a proper design of relational database schema. They employed the concept of fuzziness to deal with uncertainty inherited with the association mining process. Our work is also in the line of association mining technique application to the database design. But our main purpose is for the understanding of legacy databases and our method deals with uncertainty by means of heuristic in the step of rule filtering.

5. Conclusions and Future Work

A forward engineering approach to the design of a complete database starts from the high-level conceptual design to capture detail requirements of the enterprise. Common tool normally used to represent these requirements is the entity-relationship, or ER, diagram and the product of this design phase is a conceptual schema. Typically, the schema at this level needs some adjustments based on the procedure known as normalization in order to reach a proper database design. Then, the database implementation moves to the lower abstraction level of logical design in which logical schema is constructed in a form of relations, or database tables.

In legacy systems that design documents are incomplete or even missing, the system maintenance or modification is a difficult task due to the lack of knowledge regarding high-level design of the system. To tackle this problem, a database reverse engineering approach is essential.

In this paper, we propose a method to discover conceptual schema from the database instances, or relations. The discovering technique is based on the association mining incorporated with some heuristic to produce a minimal set of association rules. Transformation rules are then applied to convert association rules to database dependencies. Normalization is the principal concept of our heuristic and transformation. To deduce repeating group, insert anomaly, delete anomaly and update anomaly. We introduce the novel algorithm, called NoWARs, to normalize the database tables. In the normalization process, NoWARs uses only 100% confidence association rules with any support values. The results from the NoWARs algorithm are the same as the design schema obtained from the database designer. But NoWARs cannot normalize data model to the level higher than third normal form, which might be the desired level of a highly secured database. We thus plan to improve our methodology to discover a conceptual schema up to the level of fifth normal form.

Acknowledgments

This research has been conducted at the Data Engineering and Knowledge Discovery (DEKD) research unit, fully

supported by Suranaree University of Technology. The work of first and second authors has been funded by grants from Suranaree University of Technology and the National Research Council of Thailand (NRCT), respectively. The third author has been supported by a grant from the Thailand Research Fund (TRF, grant number RMU5080026).

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between set of items in large databases", in Proceedings of ACM SIGMOD International Conference on Management of Data, 1993, pp. 207-216.

[2] R. Agrawal, and R. Srikant, "Fast algorithms for mining association rules in large database", in Proceedings of 20th International Conference on Very Large Data Base, 1994, pp. 487-499.

[3] W. W. Armstrong, "Dependency structures of database relationships", Information Processing, Vol.74, 1974, pp. 580-583.

[4] F. Berzal, J. Cubero, N. Marin, and J. Serrano, "TBAR: An efficient method for association rule mining in relational databases", Data & Knowledge Engineering, Vol.37, No.1, 2001, pp. 47-64.

[5] G. Chen, M. Ren, P. Yan, and X. Guo, "Enriching the ER model based on discovered association rules", Information Sciences, Vol.177, 2007, pp. 1558-1556.

[6] R. Chiang, T. M. Barron, and V. C. Storey, "A framework for the design and evaluation of reverse engineering methods for relational databases", Data & Knowledge Engineering, Vol.21, 1997, pp. 57-77.

[7] E.J. Chikofsky, and J. H. Cross, "Reverse engineering and design recovery: A taxonomy", IEEE Software, Vol.7, No.1, 1990, pp. 13-17.

[8] E. F. Codd, "A relational model of data for large shared data banks", Communications of the ACM, Vol.13, No.6, 1970, pp. 377-387.

[9] C. J. Date, and R. Fagin, "Simple conditions for guaranteeing higher normal forms in relational databases", ACM Transactions on Database Systems, Vol.17, No.3, 1992, pp. 465-476.

[10] J. Han, et al., "DBMiner: System for data mining in relational databases and data warehouses", in Proceedings of CASCON'97: Meeting of Minds, 1997, pp. 249-260.

[11] J. Han, and M. Kamber, Data Mining: Concepts and Techniques, San Diego: Academic Press, 2001.

[12] J. Hipp, U. Guntzer, and U. Grimmer, "Integrating association rule mining algorithms with relational database systems", in Proceedings of 3rd International Conference on Enterprise Information Systems, 2001, pp. 130-137.

[13] N. Lammari, I. Comyn-Wattiau, and J. Akoka, "Extracting generalization hierarchies from relational databases: A reverse engineering approach", Data & Knowledge Engineering, Vol.63, 2007, pp. 568-589.

[14] H. Lee, and C. Yoo, "A form driven object-oriented reverse engineering methodology", Information Systems, Vol.25, No.3, 2000, pp. 235-259.

[15] J. Perez, I. Ramos, V. Anaya, J. M. Cubel, F. Dominguez, A. Boronat, and J. A. Carsi, "Data reverse engineering for legacy databases to object oriented conceptual schemas", Electronic Notes in Theoretical Computer Science, Vol.72, No.4, 2003, pp. 7-19.

[16] S. T. Sreenath, S. Bodogala, K. Alsabti, and S. Ranka, "An efficient algorithm for the incremental updation of association rules in large databases", in Proceedings of 3rd International Conference on KDD and Data Mining, 1997, pp. 263-266.

[17] S. Tsechansky, N. Pliskin, G. Rabinowitz, and A. Porath, "Mining relational patterns from multiple relational tables", Decision Support Systems, Vol.27, No.1-2, 1999, pp. 179-195.

Natthapon Pannurat received his bachelor and master degrees in computer engineering in 2007 and 2009, respectively, from Suranaree University of Technology. He is currently a faculty member of Information Sciences, Nakhon Ratchasima College. His research interests are database management systems, data mining and machine learning.

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, Artificial Intelligence, Logic and Constraint Programming, Deductive and Active Databases.

Kittisak Kerdprasop is an associate professor and the director of DEKD research unit 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, Machine Learning, Artificial Intelligence, Logic and Functional Programming, Probabilistic Databases and Knowledge Bases.

Knowledge Induction from Medical Databases with Higher-Order Programming

Nittaya Kerdprasop and Kittisak Kerdprasop
Data Engineering and Knowledge Discovery (DEKD) Research Unit
School of Computer Engineering, Suranaree University of Technology
111 University Avenue, Nakhon Ratchasima 30000
THAILAND
nittaya@sut.ac.th, kittisakThailand@gmail.com

Abstract: - Medical data mining is an emerging area of computational intelligence applied to automatically analyze patients' records aiming at the discovery of new knowledge potentially useful for medical decision making. Induced knowledge is anticipated not only to increase accurate diagnosis and successful disease treatment, but also to enhance safety by reducing medication-related errors. Modern healthcare organizations regularly generate huge amount of electronic data that could be used as a valuable resource for knowledge induction to support decision-making of medical practitioners. Unfortunately, a domain-specific decision support system that provides a suite of customized and flexible tools to efficiently induce knowledge from medical databases with representational heterogeneity does not currently exist. We, thus, design and develop a medical decision support system based on a powerful logic programming framework. The proposed system includes a knowledge induction component to induce knowledge from clinical data repositories and the induced knowledge can also be deployed to pre-treatment data from other sources. The implementation of knowledge induction engine has been presented to express the power of higher-order programming of logic-based language. The flexibility of our mining engine is obtained through the pattern matching and meta-programming facilities provided by logic-based language.

Key-Words: - Medical decision making, Medical informatics, Logic-based knowledge induction, Higher-order programming

1 Introduction

Knowledge is a valuable asset to most organizations as a substantial source to enhance understanding of data relationships and support better decisions to increase organizational competency. Automatic knowledge acquisition can be achieved through the availability of the knowledge induction component. The induced knowledge can facilitate various knowledge-related activities ranging from expert decision support, data exploration and explanation, estimation of future trends, and prediction of future outcomes based on present data.

In this paper, we present the knowledge induction system specifically designed to facilitate knowledge discovery from medical data. Various data mining and machine learning methods had been proposed to learn useful knowledge from medical data [5], [6], [7], [8], [17], [19]. Major techniques adopted by many researchers are rule induction and classification tree generation with the main purpose to support medical diagnosis [3], [9], [13]. Some researchers had even extended the knowledge discovery aspect to the larger scale of medical decision support system and data warehouse [2], [4], [10], [11], [20].

Our work is also in the main stream of medical decision support system development, but our methodology is different from those appeared in the literature. The system proposed in this paper is based on logic and higher-order programming paradigms. The justification of our logic-based system is that the closed form of Horn clauses that treats program in the same way as data facilitates fusion of knowledge learned from different sources; this situation is a normal setting in medical domain. Knowledge reuse can also easily practice in this framework. We design the system as an integrated environment storing a repertoire of tools for discovering various kinds of knowledge.

The outline of this paper is as follows. Section 2 briefly discusses knowledge induction methods implemented in our system. Section 3 reviews the basics of logic and higher-order programming. Sections 4 and 5 present the conceptual design and implementation, respectively, of our system. Section 6 concludes the paper.

2 Knowledge Induction Methods

This section briefly reviews the three main data mining methods extensively applied to induce knowledge from varieties of data domains. These methods are implemented in our medical decision support system.

2.1 Tree-Based Knowledge Induction

Decision tree induction [18] is a popular method for inducing knowledge from data. Popularity is due to the fact that mining result in a form of decision tree is interpretability, which is more concern among medical practitioners than a sophisticated method but lack of understandability. A decision tree is a hierarchical structure with each node contains decision attribute and node branches corresponding to different attribute values of the decision node. The goal of building decision tree is to partition data with mixing classes down the tree until the leaf nodes contain pure class.

In order to build a decision tree, we need to choose the best attribute that contributes the most towards partitioning data to the purity groups. The metric to measure attribute's ability to partition data into pure class is *Info*, which is the number of bits required to encode a data mixture. To choose the best attribute, we have to calculate information gain, which is the yield we obtained from choosing that attribute. The information gain calculates yield on data set before splitting and after choosing attribute with two or more splits. The gain value of each candidate attribute is calculated. Then choose the maximum one to be the decision node. The process of data partitioning continues until the data subset has the same class label.

2.2 Association Mining

Association mining is the discovery of 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 $DB = \{ C_1, C_2, C_3, ..., C_n\}$ be a database of n cases or observations and each case contains items in I. A *pattern* is a set of items that occur in a case. The number of items in a pattern is called the length of the pattern. To search for all valid patterns of length 1 up to m in large database is computational expensive. For a set I of m different items, the search space for all distinct patterns can be as huge as 2^m-1. To reduce the size of the search space, the *support* measurement has been introduced [1]. The function *support*(P) of a pattern P is defined as a number of cases in DB containing P. Thus, *support*(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. The *minS* metric has an anti-monotone property and is applied as a basis for reducing search space of mining frequent patterns in algorithm Apriori [1].

2.3 Data Clustering

Clustering refers to the iterative process of automatic grouping of data based on their similarity. There exist a large number of clustering techniques, but the most classical and popular one is the k-means algorithm [12]. Given a data set containing n objects, k-means partitions these objects into k groups. Each group is represented by the centroid, or central point, of the cluster. Once cluster means or representatives are selected, data objects are assigned to the nearest centers. The algorithm iteratively selects new better representatives and reassigns data objects until the stable condition has been reached. The stable condition can be observed from cluster assigning that each data object does not change its cluster.

3 Programming Based on Logic

In logic programming, a clause is a disjunction of literals (atomic symbols or their negations) such as $p \vee q$ and $\neg p \vee r$. A statement is in clausal form if it is a conjunction of clauses such as $(p \vee q) \wedge (\neg p \vee r)$. Logic programming is a subset of first order logic in which clauses are restricted to Horn clauses.

A Horn clause, named after the logician Alfred Horn [16], is a clause that contains at most one positive literal such as $\neg p \vee \neg q \vee r$. Horn clauses are widely used in logic programming because their satisfiability property can be solved by resolution algorithm (an inference method for checking whether the formula can be evaluated to true).

A Horn clause with no positive literal, such as $\neg p \vee \neg q$, which is equivalent to $\neg (p \wedge q)$, is called *query* in Prolog and can be interpreted as ':- *p, q*' in which its value (true/false) to be proven by resolution method. A clause that contains exactly one positive literal such as r is called a *fact* representing a true statement, written in clausal form as '*r* :-' in which the condition part is empty and that means r is unconditionally true. Therefore, facts are used to represent data. A Horn clause that contains one positive literal and one or more

negative literals such as $\neg p \vee \neg q \vee r$ is called a *definite clause* and such clause can equivalently written as $(p \wedge q) \rightarrow r$ which in turn can be represented as a Prolog *rule* as *r :- p, q.* The symbol ':-' is intended to mean '$\leftarrow$', which is implication in first-order logic (it stands for 'if'), and the symbol ',' represents the operator $\wedge$ (or 'AND').

In Prolog, rules are used to define procedures and a Prolog program is normally composed of facts and rules. Running a Prolog program is nothing more than posing queries to obtain true/false answers. The advantages of using logic programming are the flexible form of query posing and the additional information regarding variable instantiation obtained from the Prolog system once the query is evaluated to be true.

The symbols *p, q, r* are called *predicates* in first-order logic programming and they can be quantified over variables such as *r(X) :- p(X,Y), q(Y).* This clause has the same meaning as $\forall X (p(X,Y) \wedge q(Y) \rightarrow r(X))$.The scope of variables is within a clause (delimit the end of clause with a period). Horn clauses are thus the fundamental concept of logic programming.

Higher-order predicate is a predicate in a clause that can quantify over other predicate symbols [14], [15]. As an example, besides the rule *r(X):- p(X,Y), q(Y),* if we are also given the following five Horn clauses (or facts): *p(1, 2). p(1, 3). p(5, 4). q(2). q(4).*

By asking the query: *?- r(X),* we will get the response as '*true*' and also the first instantiation information as *X=1.* If we want to know all instantiations that make *r(X)* to be true, we may ask the query: *?- findall(X, r(X), Answer).*We will get the response: *Answer = [1,5],* which is a set of all answers obtained from the predicate *r(X)* according to the given facts. The predicate symbol *findall* quantifies over the variables *X, Answer,* and the predicate *r.* The predicate *findall* is thus called a higher-order predicate.

Meta-level programming is also another powerful feature of Prolog. Meta-programs treat other programs as their input data. Data and program in Prolog take the same representational format, i.e. clausal form. Therefore, it is very natural to write meta-program in Prolog.

The following example illustrates the procedure *map* that takes a list of integers [1,2,3,4,5] and another procedure *square* as its input arguments and produce a list of square values as its output. If we pose the query: *?- map(square, [1,2,3,4,5], L),* then we will get the answer: *L = [1,4,9,16,25].*

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

map(ProcedureName, [H|T], [NewH|NewT]) :-
        Procedure=.. [ProcedureName,H,NewH],
        call(Procedure),
        map(ProcedureName, T, NewT).

map(_, [], []).
```

4 Medical Decision Support System

Health information is normally distributive and heterogeneous. Hence, we design the medical decision support system (Figure 1) to include data integration component at the top level to collect data from distributed databases and also from documents in text format. Data at this stage are to be stored in a warehouse to support direct querying as well as analysis with knowledge induction engine.

Knowledge base in our design stores both induced knowledge, in which its significance has to be evaluated by the domain expert, and background knowledge encoded from consultation with human experts. Knowledge inferring and reasoning is the module interfacing with medical practitioners and physicians at the front-end and accessing knowledge base at the back-end.

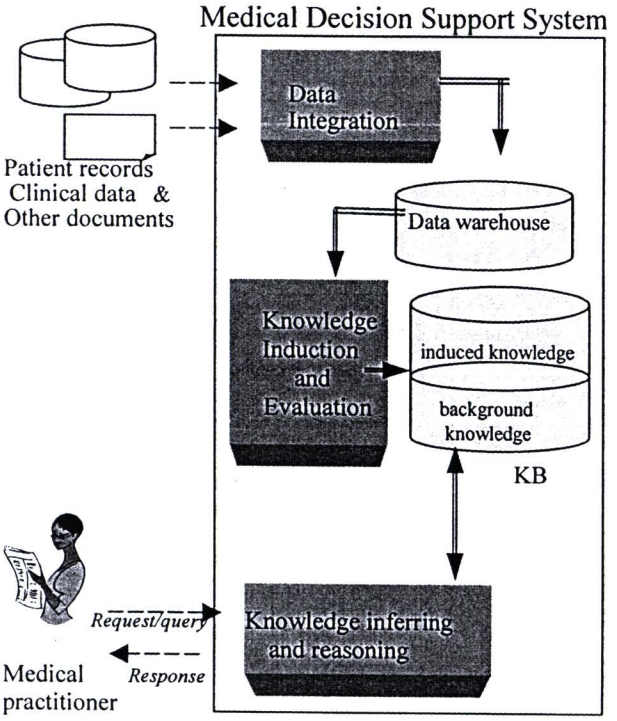


Figure 1. Knowledge induction component in the medical decision support system

The process of knowledge discovery is complex and iterative in its nature. We design the system to be composed of two phases: knowledge induction and knowledge inferring.

Knowledge induction is the back-end of the system responsible for acquiring and discovering new and useful knowledge. Usefulness is to be validated at the final step by human experts. Discovered knowledge is stored in the knowledge base to be applied to solve new cases or create new knowledge in the knowledge inferring phase, which is the front-end of the proposed system.

The proposed system obtains input from heterogeneous data sources. Such data can be redundant, incomplete, and noisy. Therefore, the knowledge-induction-and-evaluation component (Figure 2) has been designed to clean, transform, and select only relevant data sample.

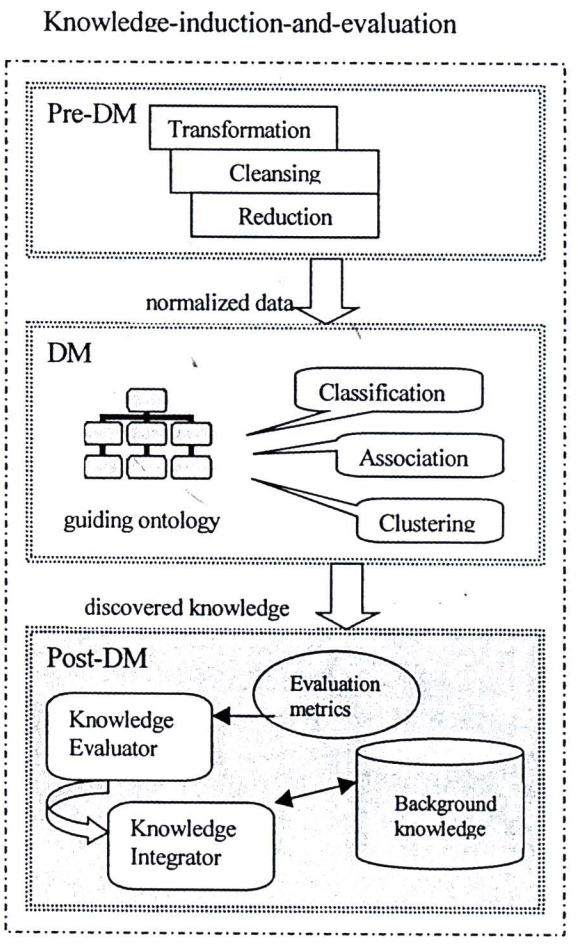


Figure 2. Architecture of the knowledge-induction-and-evaluation component

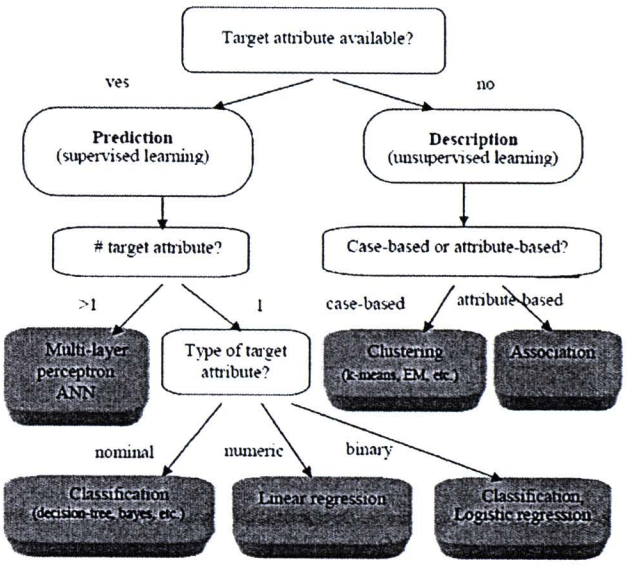


Figure 3. Ontology for guiding mining-method selection at the DM step

The DM component is for performing various mining tasks. Currently, we design and implement three different mining modules, i.e. classification, association, and clustering. We adopt the ontology concept at this step to guide the mining methodology selection. A simple form of ontology to select appropriate mining method is shown in Figure 3.

The Post-DM component composed of two main features: knowledge evaluator and knowledge integrator. These features perform functionalities aiming at a feasible knowledge deployment. Knowledge evaluator involves evaluation, based on corresponding measurement metrics, of the mining results. Knowledge integrator examines the induced patterns to remove redundant knowledge.

5 System Implementation

In this section, we present the Prolog coding of DM, a major module for mining different kinds of knowledge in the knowledge-induction-and evaluation component. Prolog code is based on the syntax of SWI Prolog (www.swi-prolog.org).

Data format. The data to be used by any mining method of the DM module take the same format, that is, as a Prolog file. As an illustration, we use the allergy data of ten patients. The following data show information of ten patients suffering from allergy (class = yes). The possible indicative symptoms are sore throat, fever, swollen glands, congestion, and

headache. Some patients had some of these symptoms but are not suffering from allergy (class = no). To induce the common symptoms (or model) of allergy patients from this data, we have to save this data set as a Prolog file (data.pl).

```
%% Data: Allergy diagnosis
 % Patients' symptoms and their possible values
attribute( soreThroat, [yes, no]).
attribute( fever, [yes, no]).
attribute( swollenGlands, [yes, no]).
attribute( congestion, [yes, no]).
attribute( headache, [yes, no]).
attribute( class, [yes, no]).
   % data instances
instance(1, class=no, [soreThroat=yes, fever=yes,
            swollenGlands=yes, congestion=yes,
            headache=yes]).
instance(2, class=yes, [soreThroat=no, fever=no,
            swollenGlands=no, congestion=yes,
            headache=yes]).
instance(3, class=no, [soreThroat=yes, fever=yes,
            swollenGlands=no, congestion=yes,
            headache=no]).
instance(4, class=no, [soreThroat=yes, fever=no,
            swollenGlands=yes, congestion=no,
            headache=no]).
instance(5, class=no, [soreThroat=no,
            fever=yes, swollenGlands=no,
            congestion=yes, headache=no]).
instance(6, class=yes, [soreThroat=no, fever=no,
            swollenGlands=no, congestion=yes,
            headache=no]).
instance(7, class=no, [soreThroat=no,
            fever=no, swollenGlands=yes,
            congestion=no, headache=no]).
instance(8, class=yes, [soreThroat=yes, fever=no,
            swollenGlands=no,
            congestion=yes, headache=yes]).
instance(9, class=no, [soreThroat=no,
            fever=yes, swollenGlands=no,
            congestion=yes, headache=yes]).
instance(10, class=no, [soreThroat=yes,
            fever=yes, swollenGlands=no,
            congestion=yes, headache=yes]).
```

Classification. The objective of classification is to induce data model of two classes: positive (class = yes) and negative (class=no). Binary classification is a typical task in medical data mining. The code, however, can be easily modified to classify data with more than two classes. To induce the common symptoms (or model) of patients suffering from allergy, we use the decision-tree induction method [18]. The process starts when the following main module is invoked. Note that clauses containing higher-order predicates are highlighted throughout the given program code.

```
:-include('data.pl').
:-dynamic current_node/1,node/2,edge/3.

main :-
    init(AllAttr,EdgeList),
    getNode(N), % get node number
    create_edge(N,AllAttr,EdgeList),
    print_model.

init(AllAttr,[root-nil/PB-NB]):-
    retractall(node(_,_)),
    retractall(current_node(_)),
    retractall(edge(_,_,_)),
    assert(current_node(0)),
    findall(X, attribute(X,_), AllAttr1),
    delete(AllAttr1, class, AllAttr),
    findall(X2,instance(X2,class=yes,_),PB),
    findall(X3,instance(X3,class=no,_),NB).

getNode(X) :-
    current_node(X),
    X1 is X+1,
    retractall( current_node(_)),
    assert( current_node(X1)).
```

The main module calls the *init* procedure (or precidate) to initialize the temporary knowledge base by removing all information that might be remained in the knowledge base and asserting the root node of the tree. The node and edge structures of our decision tree have the following formats:

node(nodeID, [PositiveCase]-[NegativeCase])

edge(ParentNode, EdgeLabel, ChildNode)

The node structure is composed of two parts: node-id and the mixture of positive and negative cases in that node. The edge is a link from parent node to child node. Each edge contains three pieces of information; that is, id of parent node, the edge label, and id of child node. Node id 0 is a special node representing a root node and it links to node number 1. The tree building starts with the *create_edge* and *create_nodes* procedures.

```
create_edge(_,_,[]) :- !.
create_edge(_,[],_) :- !.
create_edge(N, AllAttr, EdgeList) :-
        create_nodes(N, AllAttr, EdgeList).

create_nodes(_,_,[]) :- !.
create_nodes(_,[],_) :- !.
create_nodes(N, AllAttr, [H1-H2/PB-NB | T]) :-
      getNode(N1),    % get node number N1
       assert(edge(N,H1-H2,N1)),
       assert(node(N1,PB-NB)),
       append(PB, NB, AllInst),
       ( (PB \== [], NB \== []) ->
          (cand_node(AllAttr, AllInst, AllSplit),
           best_attribute(AllSplit,
                        [V, MinAttr, Split]),
           delete(AllAttr, MinAttr, Attr2),
           create_edge( N1, Attr2, Split))
          ;  true ),
         create_nodes(N, AllAttr, T).

best_attribute([], Min, Min).
best_attribute([H | T], Min) :-
      best_attribute(T, H, Min).
best_attribute([H | T], Min0, Min) :-
     H =[V,_, _ ],
      Min0 = [V0, _, _ ],
      ( V < V0 -> Min1 = H;
                  Min1 = Min0),
      best_attribute(T, Min1, Min).

  % generate candidate decision node
cand_node([],_,[]) :- !.
cand_node(_,[],[]).
cand_node([H | T],Ins,[[Val,H,SplitL] | Att]) :-
      info(H, Ins, Val, SplitL),
      cand_node(T,Ins,Att).

  % compute Info of each candidate node
concat3(A,B,C,R) :-
       atom_concat(A,B,R1),
       atom_concat(R1,C,R).
info(A, CurInstL, R, Split) :-
      attribute(A,L),
       maplist( concat3(A,=), L, L1),
       suminfo(L1, CurInstL, R, Split).

suminfo([],_,0,[]).
suminfo([H | T], CurInstL, R, [Split | ST]) :-
       AllBag=CurInstL,
       term_to_atom(H1,H),
       findall(X1, (instance(X1,_,L1),
                  member(X1, CurInstL),
                  member(H1,L1)), BagGro),
       findall(X2,(instance(X2,class=yes, L2),
                  member(X2, CurInstL),
                  member(H1,L2)), BagPos),
       findall(X3,(instance(X3,class=no, L3),
                  member(X3, CurInstL),
                  member(H1,L3)), BagNeg),
       (H11=H22) =H1,
       length(AllBag, Nall),
       length(BagGro, NGro),
       length(BagPos, NPos),
       length(BagNeg, NNeg),
       Split = H11-H22/BagPos-BagNeg,
        suminfo(T, CurInstL, R1,ST),
       ( NPos is 0 *->L1 = 0;
         L1 is (log(NPos/NGro)/log(2)) ),
       ( 0 is NNeg *->L2 = 0;
         L2 is (log(NNeg/NGro)/log(2)) ),
       ( NGro is 0 -> R= 999;
         R is (NGro/Nall)*(-(NPos/NGro)*L1-
               (NNeg/NGro)*L2)+R1 ).
```

The given source code does not provide detail for *print_model* procedure. Interested readers are suggested to simply add a rule *print_model :- true.*

Then run the program by calling predicate *main*. Prolog will respond *true* with no other information because we simply add the always-true condition in the *print_model* predicate. At this moment we can view the tree model by calling *listing(node)* and *listing(edge)* predicates. The results will be as follows:

```
1 ?- main.
true.

2 ?- listing(node).
:- dynamic user:node/2.
user:node(1, [2, 6, 8]-[1, 3, 4, 5, 7, 9, 10]).
user:node(2, []-[1, 3, 5, 9, 10]).
user:node(3, [2, 6, 8]-[4, 7]).
user:node(4, []-[4, 7]).
user:node(5, [2, 6, 8]-[]).
true.

3 ?- listing(edge).
:- dynamic user:edge/3.
user:edge(0, root-nil, 1).
user:edge(1, fever-yes, 2).
user:edge(1, fever-no, 3).
user:edge(3, swollenGlands-yes, 4).
user:edge(3, swollenGlands-no, 5).
true.
```

The running results convey the following information. From node number 1, the edge with label fever-yes (representing attribute fever with a value yes) links to node number 2. Node 1 contains all ten cases of patients suffering and not suffering from allergy, whereas node 2 contains the information []-[1,3,5,9,10] to infer none of positive cases and five negative cases. Therefore, the results in the above node and edge structures represent the following data model:

```
class(allergy) :- fever=no,
                  swollenGlands=no.
```

Association mining. We implement the association mining module based on the algorithm APRIORI [1]. The implementation shows only the first pass of the algorithm; that is, the generation of frequent itemsets. The second pass, which is the generation of association rules from frequent itemsets, can be easily extended from the given code.

Main predicate of this module is *association_mining*. Upon invocation, this predicate obtains input data from the predicate *input(Data)*, and get the minimum support value through the predicate *min_support(V)*. Then the main predicate starts the process by making candidate and large itemsets of length one, two, three, and so on (through the predicates *makeC1*, *makeL*, and *apriori_loop*, respectively). All highlighted terms are higher-order predicates. These predicates are *maplist*, *include*, and *setof*.

The predicate *maplist* takes three arguments; therefore, it may be written as *maplist/3*. This predicate applies its first argument, which is also a predicate, to each element of a list appeared in the second argument. The result is a list in the third argument.

The predicate *include/3* takes another predicate as its first argument and adds the result obtained from the first argument to the list in second argument. The result appears as a list in the third argument. The predicate *setof/3* also works with other predicate to collect each answer as a list in its third argument.

```
association_mining :-
        input(Data),
        min_support(V),
        makeC1(C),
        makeL(C,L),
        apriori_loop(L,1).

apriori_loop(L,N) :-
        length(L) is 1,!.
apriori_loop(L,N) :- N1 is N+1,
        makeC(N1,L,C),
        makeL(C, Res),
        apriori_loop(Res, N1).

makeC1(Ans) :- input(D),
        allComb(1, ItemSet, Ans2),
        maplist(countSS(D), Ans2, Ans).

makeC(N,ItemSet,Ans) :- input(D),
        allComb(2,ItemSet, Ans1),
        maplist(flatten, Ans1, Ans2),
```

```
        maplist(list_to_ord_set, Ans2, Ans3) ,
        list_to_set(Ans3,Ans4),
        include(len(N), Ans4, Ans5),
        maplist(countSS(D), Ans5, Ans).

              %scan database to find: List+N
makeL(C,Res) :- include(filter, C, Ans),
                maplist(head, Ans, Res).

filter(_+N) :-  input(A),
        length(A, I),
        min_support(V),
        N>=(V/100)*I.

head(H+_,H).

    % arbitrary subset of the set containing
    %  given number of elements
comb(0, _, []).
comb(N, [X|T], [X|Comb]) :-
        N > 0,
        N1 is N-1,
        comb(N1,T,Comb).
comb(N,[_|T],Comb) :-
        N > 0,
        comb(N,T,Comb).

allComb(N,I,Ans) :-
        setof( L, comb(N, I, L), Ans).

countSubset(A,[],0).
countSubset(A,[B|X],N) :-
        not(subset(A,B)),
        countSubset(A,X,N).
countSubset(A,[B|X],N) :-
        subset(A,B),
        countSubset(A,X,N1),
        N is N1+1.

countSS(SL,S,S+N) :-
        countSubset(S,SL,N).
len(N,X) :- length(X,N1), N is N1.
```

Clustering. We implement the data clustering based on k-means algorithm [12]. The main predicate is *clustering* in which the number of clusters (k) has to be specified and data are to be included. The predicate *makeInitCluster* creates initial k clusters with randomized k centroids, then assign each data to the closest centroid through the predicate *assignPoint*.

Note that the symbol '*', such as those appear in the predicate *cmax(Res, A*V)* and *freq(X, N*Y, N*F)*, refers to the data format to represent *Attribute*Value*; it does not mean multiplication. In Prolog, numerical computation will occur in a clause with the predicate *'is'*, such as *S1 is S + 1* in the *reComputeCenter* procedure.

The iteration step, *repeatCompute* predicate, re-computes the new k centroids and then re-assign each data point to the new closest centroid. Iteration stops when all data do not change their clusters. The source code presented in the following works with categorical data. For numerical or data with mixing types, the distance measurement has to be modified.

```
clustering(K) :-
        makeInitCluster(K, AllClust),
        assignPoint(AllClust, Data, Start, AllPt),
        OldClust=AllClust,
        repeatCompute(K, AllPt, OldClust).

makeInitCluster(K, AllClust):-
         initClust(K, 1, AllClust).

initClust(K, L0, []) :-
        L0 > K , ! .
initClust(K, L0, [L0*L|T]) :-
        instance(L0,_,L),
        L1 is L0+1,
        initClust(K, L1, T).

assignPoint(_, U, M, []) :-
        M > U, !.
assignPoint(AllClust, U, M, [M-V-A|T]) :-
         maplist(freq(M), AllClust, Res),
         cmax(Res, A*V),
         M1 is M+1,
         assignPoint(AllClust, U, M1, T).
```

```
freq(X, N*Y, N*F) :-
        instance(X, _, L1),
        intersection(L1, Y, I),
        length(I, F).

cmax(L, A*V) :-
       maplist(cvalue, L, L2),
       max_list(L2, V),
       member(A*V, L), !.

cvalue(_*V, V).

reComputeCenter(K, S, AllPoint, []) :-
       S > K, !.

reComputeCenter(K, S, AllPoint, [S*NewC | T]) :-
        findall(P, member(P-_-S, AllPoint), Z),
        allPointAtAllAttr(Z, NewC),
        S1 is S+1,
        reComputeCenter(K, S1, AllPoint, T).

allPointAtAllAttr(AllP, NewClusters) :-
        findall(AttName, (attribute(AttName,_),
                  AttName\==class), AttNameL),
        maplist(allPoint(AllP), AttNameL,
                  NewClusters).

allPoint(AllP, Att, A) :-
    findall(Att=V, (instance(X, _, K),
                    member(X, AllP),
                    member(Att=V, K)) , Z),
    maxFreq(Z, A*V).

maxFreq(L, A*V) :-
    findall(X*C, (member(X,L), count(X,L,C)), Z),
    cmax(Z,A*V).

repeatCompute(K, AllPt, OldClust) :-
   reComputeCenter(K,Start,AllPt,NewClus),
   ( OldClust==NewClus ->
     writeln('-No-cluster-changes***End*');
     ( writeln(newClust-NewClus),
       assignPoint(NewClus,Data,Start,AllPt2),
       writeln(allNewPoint-AllPt2),
       repeatCompute(K, AllPt2, NewClus) ) ).
```

6 Conclusion

Huge amount of data collected by hospitals and clinics are not yet turned into useful knowledge due to the lack of efficient analysis tools. We thus propose a rapid prototyping of an automatic data-mining tool to induce knowledge from medical data. The induced knowledge is to be evaluated and integrated into the knowledge base of a medical decision support system. Discovered knowledge facilitates the reuse of knowledge base among decision-support applications within organizations that own heterogeneous clinical and health databases. One obvious application of such knowledge is to pre-process other data sets by grouping it into focused subset containing only relevant data instances.

Our implementation of knowledge induction engines is based on the concept of higher-order Horn clauses using the logic-programming paradigm. Higher-order programming has been originally appeared in functional languages and soon be ubiquitous in several modern programming languages such as Java. Higher order style of programming has shown the outstanding benefits of code reuse and high level of abstraction.

This paper illustrates higher order programming techniques in Prolog by means of higher-order predicates such as *maplist*, *findall*, *setoff*, and *include*. These predicates take other predicates as its argument. With such expressive power of higher-order predicates, program coding of the designed system is very concise as demonstrated in the paper. Program conciseness contributes directly to program verification and validation, which are important issues in software engineering.

The powerful feature of meta-level programming in Prolog facilitates the reuse of data-mining results represented as rules to be flexibly applied as conditional clauses in other applications. The plausible extension of our current work is to add constraints into the knowledge induction method in order to limit the search space and therefore yield useful and timely knowledge. We also plan to extend our system to work with stream data that normally occur in modern medical institutions.

Acknowledgements

This research has been funded by grants from the National Research Council and the Thailand Research Fund (TRF, grant number RMU5080026). Data Engineering and Knowledge Discovery

(DEKD) Research Unit has been fully supported by Suranaree University of Technology.

References:

[1] R. Agrawal et al., Fast discovery of association rules, In U. Fayyad, G.Piatetsky-Shapiro, P. Smyth, and R.Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, AAAI Press, pp.307-328.

[2] Y. Bedard et al., Integrating GIS components with knowledge discovery technology for environmental health decision support, *Int. J Medical Informatics*, Vol.70, 2003, pp.79-94.

[3] C. Bojarczuk et al., A constrained-syntax genetic programming system for discovering classification rules: Application to medical data sets, *Artificial Intelligence in Medicine*, Vol.30, 2004, pp.27-48.

[4] E. German, A. Leibowitz, and Y. Shahar, An architecture for linking medical decision-support applications to clinical databases and its evaluation, *J. Biomedical Informatics*, Vol.42, 2009, pp.203-218.

[5] S. Ghazavi and T. Liao, Medical data mining by fuzzy modeling with selected features, *Artificial Intelligence in Medicine*, Vol.43, No.3, 2008, pp.195-206.

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

[7] N. Hulse et al., Towards an on-demand peer feedback system for a clinical knowledge base: A case study with order sets, *J Biomedical Informatics*, Vol.41, 2008, pp.152-164.

[8] C.-P. Hung, H.-J. Su, and S.-L. Yang, Melancholia diagnosis based on GDS evaluation and meridian energy measurement using CMAC neural network approach, *WSEAS Transactions on Information Science and Applications*, 6(3), March 2009, pp.500-509.

[9] 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*, Vol.17, No.10, 2001, pp.920-926.

[10] P.-J. Kwon, H. Kim, and U, Kim, A study on the web-based intelligent self-diagnosis medical system, *Advances in Engineering Software*, Vol.40, 2009, pp.402-406.

[11] C. Lin et al., A decision support system for improving doctors' prescribing behavior, *Expert Systems with Applications*, Vol.36, 2009, pp.7975-7984.

[12] J. MacQueen, Some methods for classification and analysis of multivariate observations, *Proceedings of the 5th Berkeley Symp. on Mathematical Statistics and Probability*, vol.1, pp.281-297.

[13] E. Mugambi et al., Polynomial-fuzzy decision tree structures for classifying medical data, *Knowledge-Based System*, Vol.17, No.2-4, 2004, pp.81-87.

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

[15] L. Naish, Higher-order logic programming in Prolog, *Technical Report* 96/2, Dept. Computer Science, Univ. Melbourne, Australia, 1996.

[16] S.-H. Nienhuys-Cheng and R.D. Wolf, *Foundations of Inductive Logic Programming*, Springer, 1997.

[17] B. Pandey and R.B. Mishra, Knowledge and intelligent computing system in medicine, *Computers in Biology and Medicine*, Vol.39, 2009, pp.215-230.

[18] J.R. Quinlan, Induction of decision trees, *Machine Learning*, Vol.1, 1986, pp.81-106.

[19] O. Rijal et al., A relook at logistic regression methods for the initial detection of lung ailments using clinical data and chest radiography, *WSEAS Transactions on Information Science and Applications*, 6(9), September 2009, pp.1503-1512.

[20] T. Wah and O. Sim, Development of a data warehouse for lymphoma cancer diagnosis and treatment decision support, *WSEAS Transactions on Information Science and Applications*, 6(3), March 2009, pp.530-543.

Approximate Frequent Pattern Discovery Over Data Stream

Kittisak Kerdprasop, and Nittaya Kerdprasop

Abstract—Frequent pattern discovery over data stream is a hard problem because a continuously generated nature of stream does not allow a revisit on each data element. Furthermore, pattern discovery process must be fast to produce timely results. Based on these requirements, we propose an approximate approach to tackle the problem of discovering frequent patterns over continuous stream. Our approximation algorithm is intended to be applied to process a stream prior to the pattern discovery process. The results of approximate frequent pattern discovery have been reported in the paper.

Keywords—Frequent pattern discovery, Approximate algorithm, Data stream analysis.

I. Introduction

DATA stream is defined as massive amounts of data continuously generated at a rapid rate, possibly time-varying and unpredictable [1], [4], [10]. Major characteristics of data streams are the continuously online arrival of data elements, uncontrolled order of such elements upon arrival, variable sizes, and a one-time processing of an element before it is discarded or archived due to the massive size of data that far exceeds the storage capacity. The requirements of timely analysis and efficient memory usage constrain most data stream mining algorithms to sacrifice accuracy of the analysis results for the fast and feasible processing.

Development of approximation algorithms [6], [13] is a direct solution to the problem of data stream analysis. However, the large volumes of data continuously arriving in a stream could eventually make the algorithms inefficient. A more practical solution is to apply a data reduction technique along with the approximation algorithms. Data summarization techniques, such as wavelet analysis [11] and histogram [4], have been proposed as synopsis data structures to provide a summary presentation of data. The issue of dynamic space allocation as the underlying data distribution changes over time is a fundamental problem of these approaches.

Data stream analysis by choosing a subset of the incoming stream is another class of techniques for producing approximate results. Sampling is a statistical-based technique widely used to scale up the algorithms [8]. Nevertheless, in the context of data stream in which the data size is unknown, simply applying a sampling method cannot give reliable approximation.

We, therefore, propose a novel approximation method to draw representatives from data stream. To produce a good approximation to the true value or quantity of underlying stream, we apply the expectation-maximization technique to get a good guess of data characteristics. Our algorithm has been designed to produce data elements from which the approximate analysis is close to the exact one. We then perform frequent pattern discovery over the sample data. Frequent pattern analyses on several data sets to verify the reliability of the method have been conducted.

The paper is organized as follows. Section 2 presents the theoretical background of our method. Section 3 is the proposed algorithm. Section 4 presents some of the experimental results from frequent pattern analyses over the reduced data stream. We conclude in Section 5 with a discussion for future work.

II. Data Stream Density Estimation

When the number of data is overwhelming and the exact data distribution is unknown, the characteristics of stream have to be estimated before data sampling can be performed. We concentrate on the sampling problem because the efficiency of frequent pattern discovery depends largely on the ability to draw samples effectively.

For a particular domain of stream data, we consider the rejection sampling method. Rejection sampling, or acceptance-rejection sampling, is a sampling method first introduce by Von Neumann [15]. This method is used in cases where a target distribution, $f(x)$, is too complicate for us to sample from it directly.

Suppose we have a simpler distribution, $g(x)$, which we can evaluate and generate samples from, then the difficult sampling problem can be avoided by sampling from $g(x)$ instead. By generating a uniform random variable u from the interval [0,1], we accept x if the condition $u \leq f(x) / Cg(x)$ holds; otherwise reject the value of x and repeat the sampling

Manuscript received October 15, 2007. This work was supported in part by the Thailand Research Fund under grant RMU-5080026 and research fund from the National Research Council of Thailand. DEKD research unit is fully supported by Suranaree University of Technology.

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).

step. Posing the restriction $Cg(x) \geq f(x)$ for some $C > 1$, we say that Cg envelopes f. The validation of this method is the envelope principle. When simulating the point (x, v) where $v = u^*Cg(x)$, we produce a uniform simulation over the subgraph of $Cg(x)$. Accepting only points such that $u \leq f(x) / Cg(x)$ then produces points (x, v) uniformly distributed over the subgraph of $f(x)$ and thus, marginally, a simulation from $f(x)$.

Rejection sampling will work best if g is a good approximation to f. However, in a high-dimensional problem the value of C needs to be chosen very large to ensure the requirement $Cg(x) > f(x)$, for all x. The result is an enormous rejection rate.

The difficulty of applying rejection sampling method directly to the problem of data stream analysis is that we do not know beforehand where the modes of f are located or how high they are. In other words, we do not know the exact characteristics of the target density. We thus propose to apply the Expectation-Maximization (EM) technique [7], [12], [14] to approximate the density $f(x)$.

We consider multi-dimensional stream data as mixtures of Gaussian, or normal, probability density functions (pdf). Gaussian mixtures [9], [12] are combinations of Gaussian distributions written as:

$$g(x) = \sum_{i=1}^{K} p_i f(x \mid \theta_i) \quad (1)$$

A random variable x denotes independent observation in K mixture components. The p_i's are the mixing proportions, $0 < p_i < 1$ for all $i = 1, ..., K$, and $p_1 + ... + p_K = 1$. The $f(x|\theta_i)$ denotes the density of a d-dimensional Gaussian distribution with mean vector μ and covariance matrix Σ, that is $\theta = (\mu,\Sigma)$, and the Gaussian pdf is given by [5], [14]:

$$g_{(\mu,\Sigma)}(x) = \frac{1}{(2\pi)^{d/2}\sqrt{\det(\Sigma)}} \exp\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\} \quad (2)$$

By varying the number of Gaussians K, the mixing proportions p_i, and the parameter θ_i of each Gaussian density function, Gaussian mixtures can be used to describe any complex pdfs.

In stream data a mixture density $p_i f(x|\theta_i)$ has been observed with unknown parameters θ_i and p_i. To find these parameters to optimally fit a mixture model for a given set of data, the EM algorithm [7], [12], [14] can be used. The EM algorithm is a broadly applicable approach to the iterative computation of maximum likelihood estimates. For a set of *iid* samples $X = \{ x_1, ..., x_N \}$ drawn from a data generation model $f_{(\mu,\Sigma)}(x_i)$, thus the resulting density for the samples is:

$$\prod_{i=1}^{N} f_{(\mu,\Sigma)}(x_i) = L(\theta \mid x) \quad (3)$$

The likelihood function $L(\theta \mid x)$ is the likelihood of the parameters given the data. In the maximum likelihood problem, the goal is to find θ that maximizes L, that is $\arg\max_\theta L(\theta \mid X)$. In the Gaussian case, the computation of the exponential can be avoided by maximizing $log\ (L(\theta \mid x))$ instead of $L(\theta \mid x)$.

The EM algorithm is an approach to find the maximum of likelihood functions in incomplete data problems. Let X be observed data, Z be unobserved data, and Y = X∪Z be full data set. The probability distribution of Z depends on X and the unknown parameter θ. Given an initial parameter $\theta^{(0)}$, The EM algorithm produces a sequence $\{\theta^{(0)}, \theta^{(1)}, \theta^{(2)}, ... \}$ that converges to a stationary point of the likelihood function

III. A Novel Algorithm on Stream Sampling

For the problem of frequent pattern discovery over data stream, we assume that the observed data distribute normally. The central idea of our approach is the bounded estimation of stream data characteristics. Given a specific number of models, the EM method is applied to estimate the mean value of each model. Then these means are scaled up to get an upper bound (E')for the underlying partially observed target density. The proposed idea can be graphically displayed as in Fig. 1.

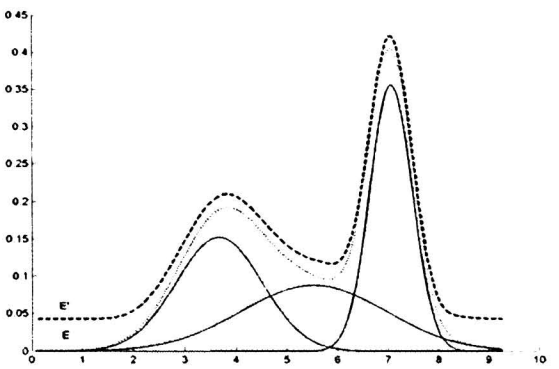


Fig. 1 Rejection sampling with an upper bound E'

The target function is represented as a one-dimensional 3-Gaussian mixtures (the three solid lines at the bottom of Fig. 1) from which we want to draw samples. The density $E(x)$ is estimated with the upper bound requirement that $E(x) > f(x)$ for all x. $E'(x)$ is the approximation (shown as a thick dash line in Fig. 1) of the unknown target density. A broad distance of E and E' (e.g., at $x = 1$) represents a rejecting area, whereas a narrow distance (e.g., at $x = 6.5$) is an acceptance one.

It should be noted that EM requires a pre-specified number of K components to be incorporated into the mixture models. According to our proposed method, a suitable number should be selected by a user. To cope with multi-dimensional problem, we propose to use a statistical method – principal component analysis (PCA) – to reduce the complicated problem to a simpler two-dimensional problem. That is, we take into account only the first and second major components of the data set. The two-dimensional data are used to train the EM algorithm to estimate parameters μ and Σ of the Gaussian mixture models. The estimated Gaussian pdf is a distribution E (as shown in Fig. 1). To sample from the estimated density

we scale up this distribution to obtain an approximate E', which is a simpler distribution that we can evaluate and generate samples from. The outline of our approximate sampling algorithm is illustrated in Fig. 2.

Input: a d-dimensional data set D with N points
an integer K to specify the number of models
a sample size SS
Output: a sample set S drawn from the mixture models

// Data preprocessing steps //
1. If $d > 0$ then
   Apply PCA to obtain 1st and 2nd components
2. Transform D to a two-dimensional data set X

// Density estimation with EM to get a rough pdf $E'(X)$ //
3. Set max_iteration = max{50, d *K}
4. Initialize parameter $\theta = (\mu, \Sigma)$ for each of K Gaussian models
5. Initialize the prior probabilities $P(m_k)$ of each model m to $1/K$, $k = 1,..., K$
6. Repeat
7. Compute the probability

$$P(m_k^{(i)} \mid x_n, \theta^{(i)}) = \frac{P(m_k^{(i)} \mid \theta^{(i)}) \cdot p(x_n \mid \mu_k^{(i)}, \Sigma_k^{(i)})}{\sum_j P(m_j^{(i)} \mid \theta^{(i)}) \cdot p(x_n \mid \mu_j^{(i)}, \Sigma_j^{(i)})}$$

8. Update means μ_k, variances Σ_k, and priors P

$$\mu_k^{(i+1)} = \frac{\sum_{n=1}^{N} x_n P(m_k^{(i)} \mid x_n, \theta^{(i)})}{\sum_{n=1}^{N} P(m_k^{(i)} \mid x_n, \theta^{(i)})}$$

$$\Sigma_k^{(i+1)} = \frac{\sum_{n=1}^{N} P(m_k^{(i)} \mid x_n, \theta^{(i)}) (x_n - \mu_k^{(i+1)})(x_n - \mu_k^{(i+1)})^T}{\sum_{n=1}^{N} P(m_k^{(i)} \mid x_n, \theta^{(i)})}$$

$$P(m_k^{(i+1)} \mid \theta^{(i+1)}) = \frac{1}{N} \sum_{n=1}^{N} P(m_k^{(i)} \mid x_n, \theta^{(i)})$$

9. Until the *max_iteration* has been reached or the joint likelihood of all data with respect to all the models is greater than the lower boundary criterion $CL(\theta)$

$$L(\theta) \geq CL(\theta) = \sum_{k=1}^{K} \sum_{n=1}^{N} P(m_k \mid x_n, \theta) \log p(x_n \mid \theta)$$

10. Get $E(X)$ as $\theta_i = (\mu_k, \Sigma_k)$ for $k = 1, ..., K$,
11. Get $E'(X)$ as a rough $\theta_i' = (\mu^r{}_k, \Sigma^r{}_k)$ from r iterations, $r < 10$

// Sampling steps //
12. Set count = 0
13. While count < SS
14. Sample x from $E(X)$
15. Generate u from U(0,1)
16. If $u \leq E(x)/(\sqrt{d} * E'(x))$
    then Accept x, add it to S, and increment count
17. Return S

Fig. 2 An algorithm to obtain approximate samples

IV. Experimentations

To verify the utility of the proposed method on the real-world data we test our algorithm on four data sets: Wisconsin diagnostic breast cancer, Chess, DNA, and Audiology. These data are taken from from UC Irvine Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). The details of selected datasets are summarized in Table I. After the sampling step, the sample data are tested for accuracy and efficiency on the discovery of frequent patterns. We adopt the Apriori algorithm [2], [3] as a method to discover frequent patterns.

TABLE I
Dataset Characteristics

Dataset	File size	# Transactions	# Items
Breast cancer	21.1 KB	191	10
Chess	237 KB	2130	37
DNA	252 KB	2,000	61
Audiology	41.1 KB	150	70

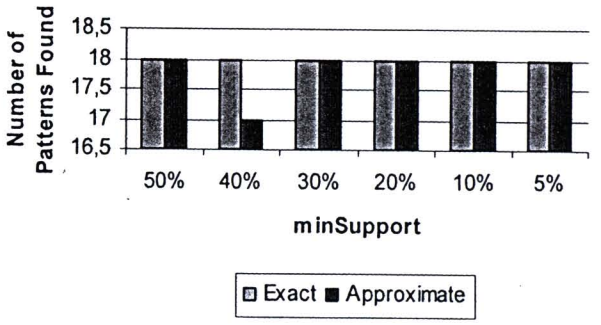
We comparatively study the performance of frequent pattern discovery on the whole dataset (and call it an exact analysis) against the sample data prepared as explained in the algorithm (Fig. 2). We compare the speed of discovery process, including sampling time for an approximate analysis, as well as the accuracy of patterns found. All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD.

The comparison results of accuracy and run time are shown in Figs. 3 and 4, respectively. An accuracy comparison has been done on the basis of number of frequent patterns discovered on each value of minimum support. The accuracy obtained from sampled data is almost as good as the accuracy of pattern discovery from the original dataset. It is also worth noticing that the lower minimum support value, the greater the number of patterns found.

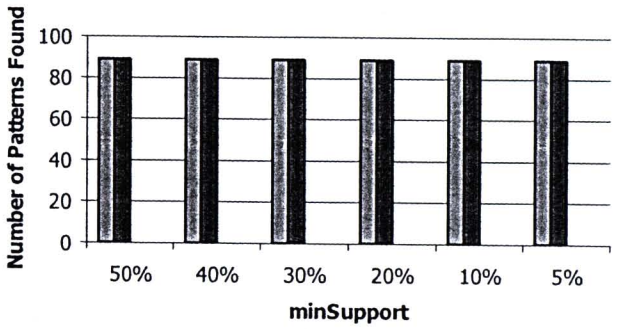
For the experimental results on speed comparison, our proposed sampling method can actually help reducing processing time of frequent pattern discovery. This gain is quite obvious in the case of low support value (minimum support threshold is less than 10%).

V. Conclusion and Discussion

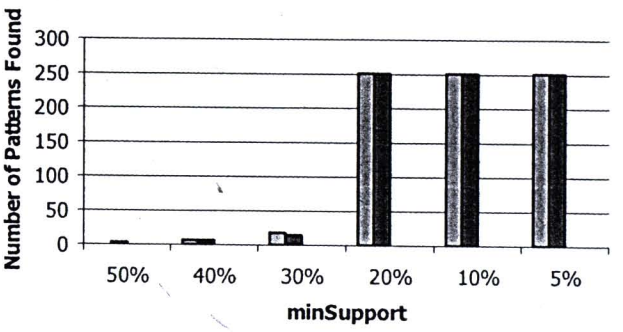
Frequent pattern discovery over data stream is a challenge problem due to the limitation on time and space. We propose to tackle the problem by means of sampling. Instead of applying simple random sampling, we argue that blindly taking sample from the stream in which we do not know the size of data in advance is incorrect. We, thus, propose a better solution by introducing the concept of guessing an upper bound E' and lower bound E of stream distribution. The distance of E and E' at each sampling point is a decision criteria for either sample acceptance or rejection. A narrow distance among the two estimated densities tends to the acceptance case if the distance ratio is greater than the generated uniform random variable from the interval [0, 1].



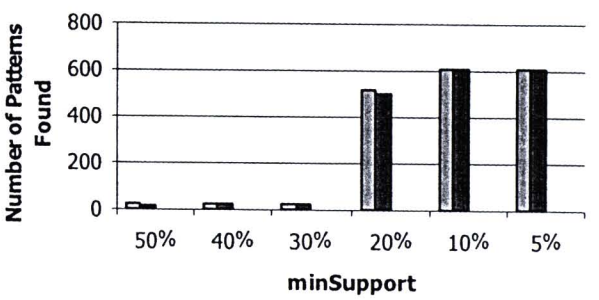
(a) Breast cancer data



(b) Chess data

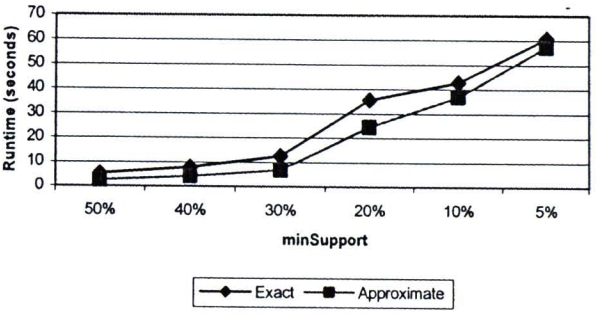


(c) DNA data

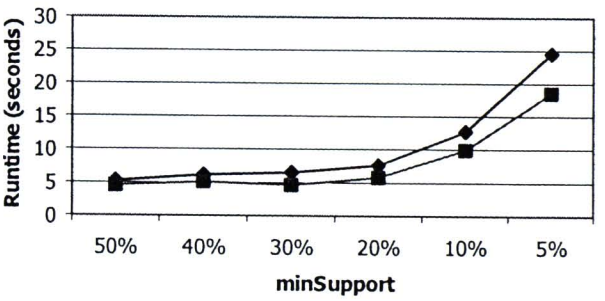


(d) Audiology data

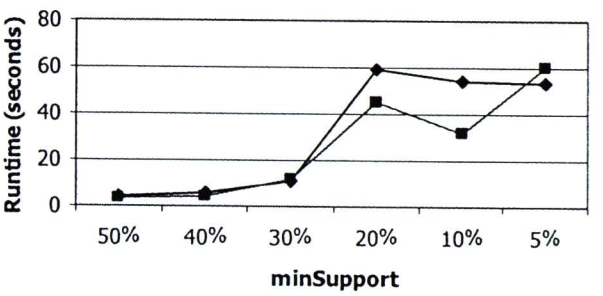
Fig. 3 an accuracy comparison of exact and approximate frequent pattern discovery



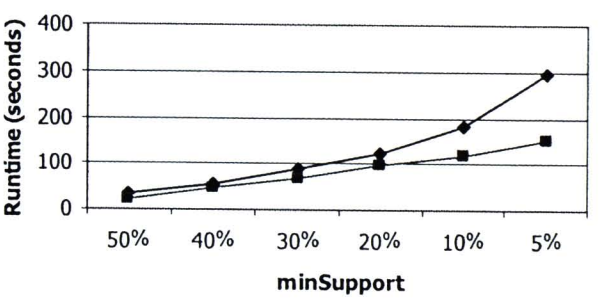
(a) Breast cancer data



(b) Chess data



(c) DNA data



(d) Audiology data

Fig. 4 a run-time comparison of exact and approximate frequent pattern discovery

The proposed idea of rejection sampling from the bounded density functions is intended to be a data preparation step prior to the frequent pattern discovery process. The experimental results confirm the accuracy and efficiency of our proposed method. We plan to investigate the problem of frequent pattern discovery from stream data further on the issues of data estimation. That is, we are interest in skewed data in which distributions are not uniformly distributed.

REFERENCES

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu, "A framework for clustering evolving data streams," in *Pro. Very Large Data Bases*, 2003.

[2] 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.

[3] R. Agrawal and R. Srikant, "Fast algorithm for mining association rules," in *Proc. Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Model and issues in data stream systems," in *Pro. ACM PODS*, 2002.

[5] J. Bilmes, "A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models," Dept. Electrical Engineering and Computer Science, University of California Berkeley, Technical Report TR-97-021, 1998.

[6] G. Coremode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," in *Pro. ACM PODS*, 2003.

[7] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society B*, vol. 39, pp. 1-22, 1977.

[8] P. Domingos and G. Hulten, "A general method to scaling up machine learning algorithms and its application to clustering," in *Pro. ICML*, 2001.

[9] M. A. T. Figueiredo and A. K. Jain, "Unsupervised learning of finite mixture models," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 24, pp. 381–396, 2002.

[10] M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data stream: A review," *SIGMOD Record*, vol. 34, pp. 18–26, 2005.

[11] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "One-pass wavelet decompositions of data streams," *IEEE Trans. Knowledge and Data Engineering*, vol. 15, pp. 541–554, 2003.

[12] J. M. Marin, K. Mengersen, and C. Robert, "Bayesian modelling and inference on mixtures of distributions," in *Handbook of Statistics*, vol. 25, Elsevier-Science, 2005.

[13] S. Muthukrishnan, "Data streams: Algorithms and applications," in *Proc. ACM-SIAM Symposium on Discrete Algorithm*, 2003.

[14] B. Resch, "A tutorial for the course computational intelligence," Available: http://www.igi.tugraz.at/lehre/CI

[15] J. von Neumann, "Various techniques used in connection with random digits," *Applied Mathematics Series*, vol. 12, National Bureau of Standards, Washington, D.C., 1951.

Kittisak Kerdprasop is an associate professor at the school of computer engineering, and a director of DEKD research unit, 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.

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.

Manuscript received October 15, 2007. This work was supported in part by the Thailand Research Fund under grant RMU-5080026 and research fund from the National Research Council of Thailand. DEKD research unit is fully supported by Suranaree University of Technology.

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.

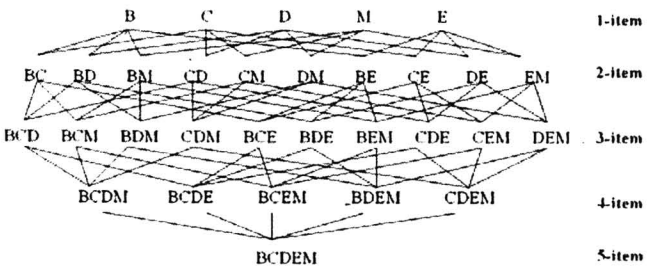


Fig. 2 A search space for finding patterns from the set of five items

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.

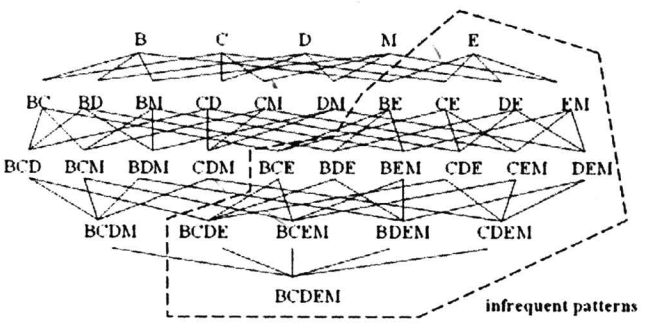


Fig. 3 A pruning of all patterns that contain an infrequent item E

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.

```
P1 = {x | x is an item in I and s(P) ≥ minS } //1-item pattern
For (k=1; Pk ≠ ∅; k++) do
    Ck+1 = Generate_candidate(Pk )
    For each Ti ∈ DB do
        Increment the count c of all candidates in Ck+1
                that are contained in Ti
    Pk = {c | c ∈ Ck and c.count ≥ minS}
Return ∪k Pk  //return all sets of frequent patterns

Generate_candidate (Pi-1 )
Ci = ∅ // initialize candidate frequent pattern set as empty
For each pattern J in Pi-1 do
    For each pattern K in Pi-1 and K ≠ J do
        If i-2 of the elements in J and K are equal then
            If all subsets of {K ∪ J} are in Pi-1 then
                Ci = Ci ∪ {K ∪ J}
Return Ci
```

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 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.

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 n^{th} 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

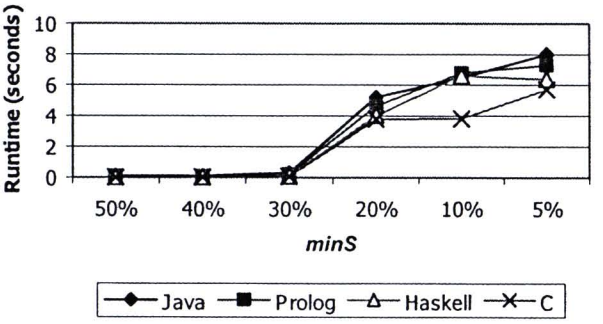
Dataset	File size	# Transactions	# Items
Vote	13.2 KB	300	17
Chess	237 KB	2,130	37
DNA	252 KB	2,000	61
Mushroom	916 KB	5,416	23

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.

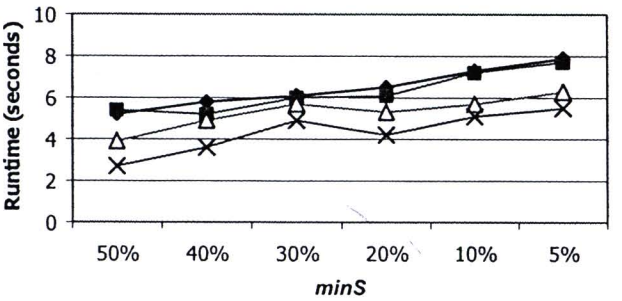
VII. CONCLUSIONS AND DISCUSSION

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.

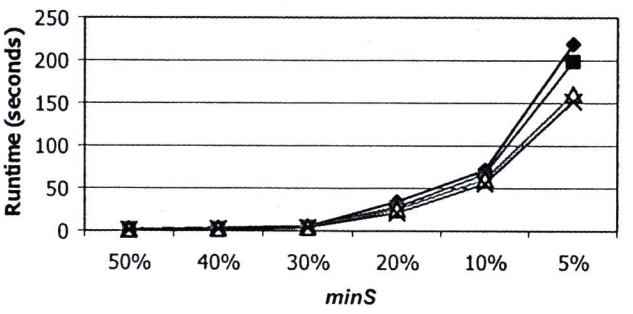
The idea to discover an association of items frequently co-occur was first proposed in 1993 by R. Agrawal, T. Imielinski, and A. Swami and the well known Apriori algorithm was proposed by R. Agrawal and A. Swami in 1994. Since then many variations of Apriori have been proposed. Most algorithms are implemented 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.



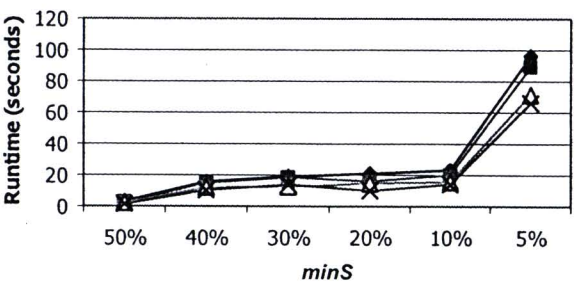
(a) Vote data



(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/.

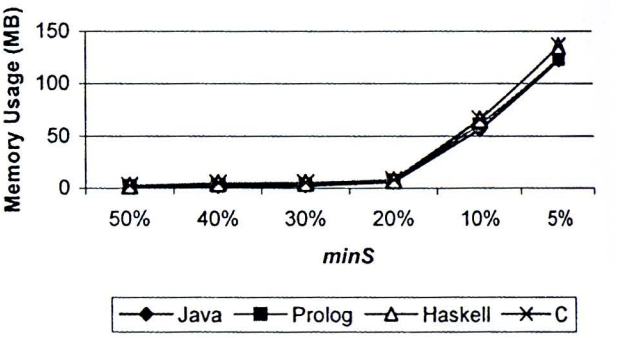
[9] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging vertical mining of large databases," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2000, pp. 22–33.

[10] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison Wesley, 2005.

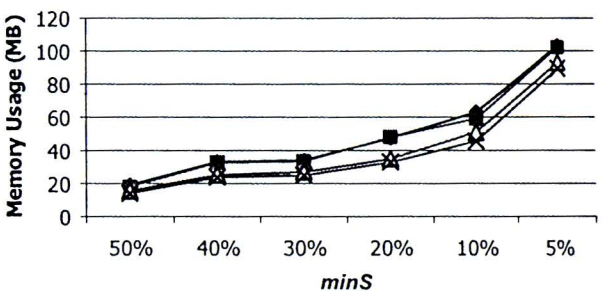
[11] S. Thompson, *Haskell: The Craft of Functional Programming* (2nd ed.), Addison Wesley, 1999.

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.

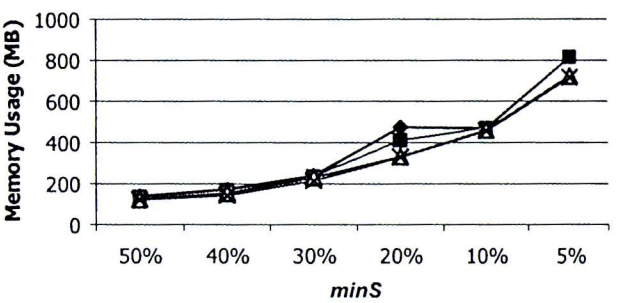
Nittaya Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science 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.



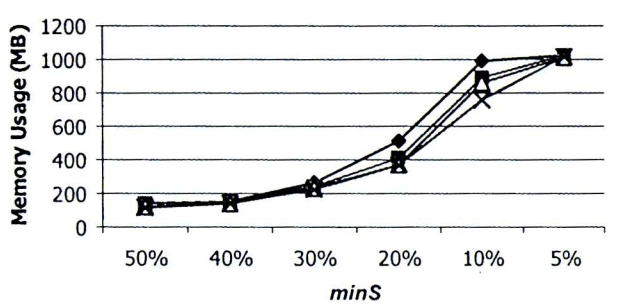
(a) Vote data



(b) Chess data



(c) DNA data



(d) Mushroom data

Fig. 8 Space comparison of different programming styles

Mining Frequent Patterns with Functional Programming

Nittaya Kerdprasop, and Kittisak Kerdprasop

***Abstract*—**Frequent patterns are 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 dataset. Most of the proposed frequent pattern mining algorithms have been implemented with imperative programming languages such as C, C++, Java. The imperative paradigm is significantly inefficient when itemset is large and the frequent pattern is long. We suggest a high-level declarative style of programming using a functional language. Our supposition is that the problem of frequent pattern discovery can be efficiently and concisely implemented via a functional paradigm since pattern matching is a fundamental feature supported by most functional languages. Our frequent pattern mining implementation using the Haskell language confirms our hypothesis about conciseness of the program. The performance studies on speed and memory usage support our intuition on efficiency of functional language.

***Keywords*—**Association, frequent pattern mining, functional programming, pattern matching.

I. Introduction

FREQUENT pattern mining is the discovery of relationships or correlations between items in a dataset. A set of market basket transactions [1], [2] is a common dataset used in frequent pattern analysis. A dataset is typically in a table format. Each row is a transaction, identified by a transaction identifier or a *TID*. A transaction contains a set of items bought by a customer. A set of transactions might be organized in either an enumerated (dense), or a sparse binary vector format [3], [7]. In either format a dataset can be processed horizontally or vertically. Fig. 1 illustrates the data organization formats of a simple market basket dataset.

In a horizontally enumerated data organization (fig. 1a), each transaction contains only items positively associated with a customer purchase. It is a simplistic representation of market basket data because it ignores other information such as the quantity of purchased items or the profit of item sold. A horizontally enumerated format is sometimes referred to as a *TidLists* dataset organization. In a vertical organization of items bought enumeration (Fig. 1b), each column stores an ordered list of *TIDs* of the transactions that contain an item. This format of a dataset occupies that same space as the horizontally enumerated format.

Figs. 1c and 1d represent a binary vector format. A value in each vector cell is 1 if the item is present in a transaction and 0 otherwise. A binary vector format is referred to as a *TidSets* dataset organization.

TI	Items
*D*1	{Cereal, Milk}
2	{Beer,Cereal,Diaper,Egg}
3	{Beer, Diaper, Milk}
4	{Beer,Cereal,Diaper,Milk}
5	{Diaper, Milk}

(a) Horizontally enumerated format

TID	Item IDs				
	B	C	D	E	M
2	1	2	2	1	
3	2	3		3	
4	4	4		4	
		5		5	

(b) Vertically enumerated format

TID	Items IDs				
	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(c) Horizontal binary vector

TID	Items IDs				
	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(d) Vertical binary vector

Fig. 1 Organization of a market basket dataset

Recent attention has been given to the influence of data organization on the performance of the process of frequent pattern discovery. Shenoy et al.[7] described the advantages of the vertical organization over the horizontal organization. They also introduced the VIPER algorithm that uses a combination of horizontal and vertical formats to reduce the space. Zaki and Gouda [10] presented a vertical data representation called *Diffset* that only keeps track of the differences in the *TidLists* of a candidate pattern from its generating frequent patterns. A vertical vector organization has been proven an efficient layout for the problem of frequent pattern discovery, but it suffers from the memory usage. We thus propose to switch the paradigm towards the algorithm implementation from conventional imperative to a declarative style of lazy functional programming. Our performance studies have confirmed the improvement on speed and memory usage.

Manuscript received November 29, 2006. This work was supported in part by the research fund from Suranaree University of Technology and the grant from the National Research Council of Thailand.

Nittaya Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224432; fax: +66-44-224602; e-mail: nittaya@ sut.ac.th, nittaya.k@gmail.com).

Kittisak Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (e-mail: kerdpras@sut.ac.th).

II. Search Space of Frequent Pattern Mining

In frequent pattern mining, we are interested in analyzing connections among items. A collection of zero or more items is called an itemset. For example, the first transaction in Fig. 1 contains the itemset {Cereal, Milk}. Since this set contains two items, it is called a 2-itemset. An itemset can be an empty set, a 1-itemset, a 2-itemset, and so on. Fig. 2 shows all combinations of distinct itemsets from the set of items {B, C, D, E, M}, where B = Beer, C = Cereal, D = Diaper, E = Egg, and M = Milk.

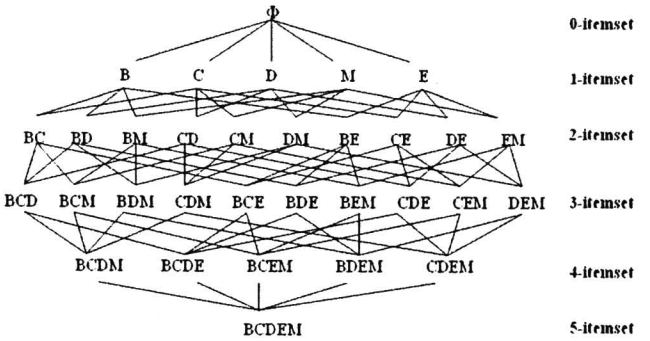


Fig. 2 A lattice of distinct itemsets

The discovery of interesting relationships hidden in large datasets is the objective of frequent pattern mining. The uncovered relationships can be represented in the form of association rules. An association rule is an inference of the form $X \rightarrow Y$, where X and Y are non-empty disjoint itemsets. To form association rules, we consider only valid itemsets. An itemset is valid if it really occurs in a transaction. For instance, from a dataset shown in Fig. 1 an itemset {Egg, Milk} is invalid because none of the customers buy both eggs and milk.

The identification of all valid itemsets is computational expensive. It can be seen from Fig. 2 that a dataset of I items has 2^I distinct itemsets. To reduce the search space, the measurements of *support* and *confidence* are used to constrain the mining process. The constraint *support* forces the mining process to discover only relationships that occur frequently, while *confidence* constrains the reliability of the inference made by a rule. The support count for an itemset Z, denoted as $\sigma(Z)$, is the number of transactions that contain a particular itemset Z. As an example, consider a dataset in Fig. 1, there are three transactions (*TID* 2, 3, 4) contain the item Beer, thus σ(Beer) = 3. Given the definition of support count, the metrics support and confidence of the association rule $X \rightarrow Y$ can be defined as follows [4], [8].

$$Support,\ s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N},$$

where N is the number of all transactions.

$$Confidence,\ c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}.$$

Given a dataset as shown in Fig. 1, an example of association rule is the statement that "customers who buy beer also buy diaper, with 60% supporting transactions and 100% confidence." An itemset is called a *frequent itemset* if its support is greater than or equal user-specified support threshold (called *minSup*). An association rule generated from frequent itemset with the confidence greater than or equal a confidence threshold (called *minConf*) is considered a valid association rule. With the pre-specified *minSup* and *minConf* metrics, the problem of association rule discovery can be stated as follows: Given a set of transactions, find all the rules having support ≥ *minSup* and confidence ≥ *minConf*. This problem can be decomposed into two subtasks:

(1) Frequent itemset generation: find all itemsets that satisfy the *minSup* threshold.

(2) Rule generation: generate from frequent itemsets all high confidence rules.

It is the *minSup* constraint that helps reducing the computational complexity of frequent itemset generation. Suppose we specify *minSup* = 2/5 = 40% on a set of transactions shown in Fig. 1; the item {Egg} is infrequent. As a result, all supersets of {Egg} are also infrequent. All infrequent itemsets can then be pruned to reduce the search space (see Fig. 3).

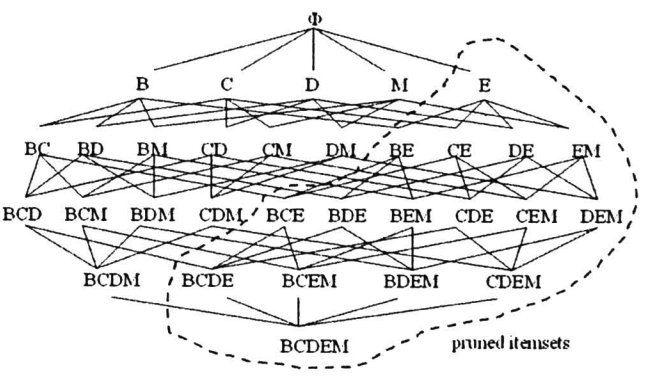


Fig. 3 A pruning of all itemsets that contain an infrequent item E

Frequent itemsets are actually patterns that appear in a dataset frequently. Finding frequent patterns has become an important data mining task. We propose that frequent patterns can be mined efficiently using a high-level programming language such as Haskell that provides a full support for pattern matching functionality.

III. Pattern Matching with Haskell

A problem in frequent pattern discovery is to determine how often a candidate pattern occurs. In association mining, a pattern is a set of items co-occurrence across a dataset. 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 itemsets 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 [5], [6], [9]. 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.

IV. Implementation

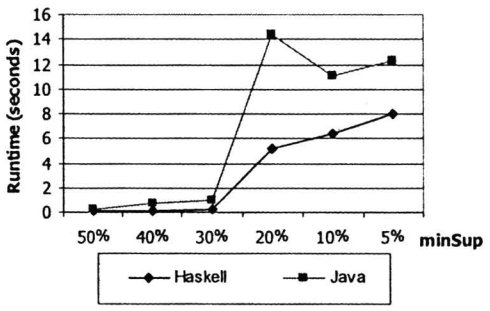
We implement Apriori algorithm [1], [2] using Haskell language as shown in Fig. 4. Each item is represented by the item identifier which is an integer. Thus, an itemset is denoted as a set of Int declared in the first line of our Haskell code. The function sumi is defined to count the number of occurrence 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 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]
```

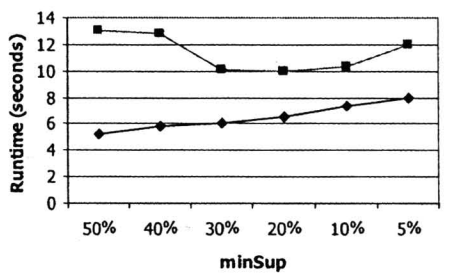
Fig. 4 Frequent itemsets discovery implemented with Haskell

V. Performance Studies

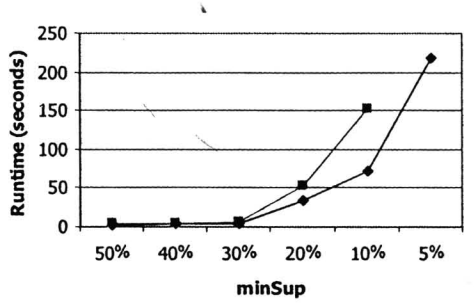
We comparatively study the performance of our implementations of frequent itemset discovery using Haskell versus Java. All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD. We select four datasets downloaded from UC Irvine Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html) to test the speed of Haskell and Java programs.



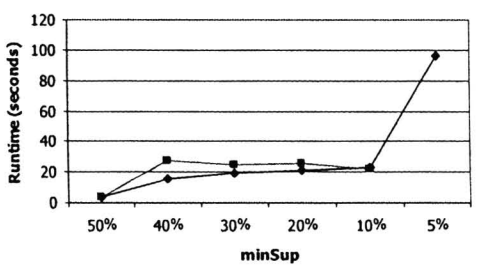
(a) Vote dataset



(b) Chess dataset



(c) DNA dataset



(d) Mushroom dataset

Fig. 5 Experimental results from two programming paradigms

TABLE I
DATASET CHARACTERISTICS

Dataset	File size	# Transactions	[a]# 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 details of selected datasets are summarized in table 1. The frequent itemset discovery pro-grams have been tested on each dataset with varied *minSup* values. Performance comparisons of Haskell and Java implementations on four datasets are graphically shown in Fig. 5.

It can be noticed from the experimental results that runtime increases as the minimum support (*minSup*) threshold gets lower. This is due to the fact that at a low level of *minSup* the number of frequent itemsets generated is significantly high. The implementation of frequent itemset discovery using Haskell outperforms that of Java on every dataset. A good performance can be clearly seen when *minSup* gets lower than 30%.

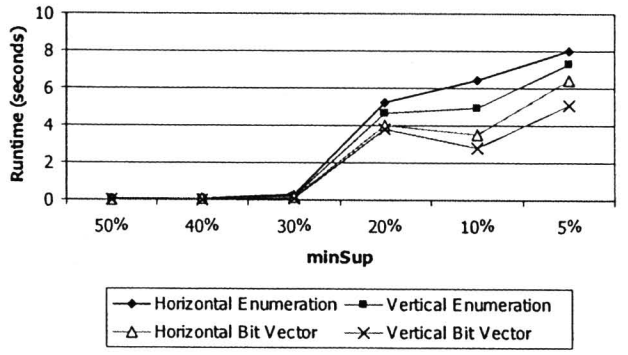
On large datasets with many items such as DNA and Mushroom, the program implemented with Java has a problem of insufficient memory and cannot run to completion at a 5% *minSup* threshold. This problem does not exist in the Haskell implementation.

The experimental results shown in Fig. 5 obtain from the datasets represented in a horizontally enumerated format. We also study the impact of different data formats on program running time and memory usage of a Haskell implementation. To observe the running time we implement the following code.

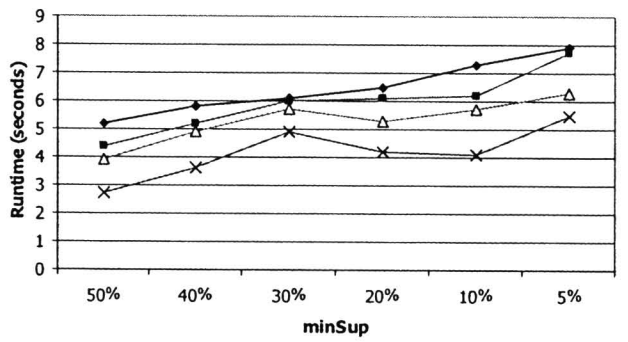
```
benchmark action = do
    prev <- getCPUTime
    action
    current <- getCPUTime
    let
      secs = fromIntegral (current-prev) / 1e12
    putStrln$ "Uses:  " ++ show secs
              ++ "  seconds "
```

The results of running time and memory usage using different styles of data representation are shown in Figs. 6 and 7, respectively. It can be noticed from the experimental results that on a speed comparison the vertical binary vector format is the fastest, the horizontal binary vector comes second following by the vertically enumerated organization. The horizontally enumerated format is the slowest one.

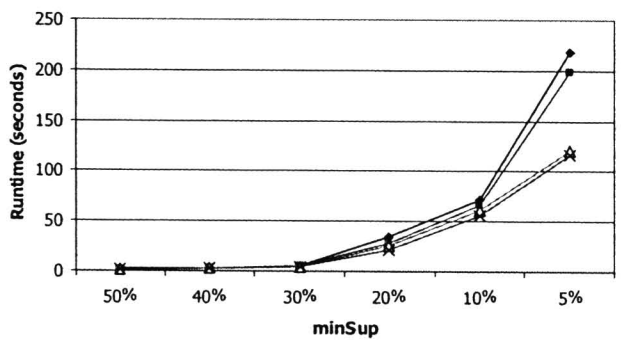
On the memory usage comparison the ordering is vice versa. Dataset represented with an enumeration format takes less storage area, while a binary vector format consumes more memory. The horizontal layout slightly outperforms the vertical layout in terms of memory usage during the process of finding frequent itemsets from the generated candidate sets.



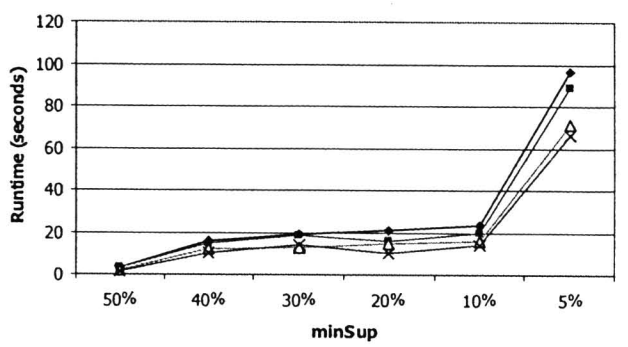
(a) Vote dataset



(b) Chess dataset

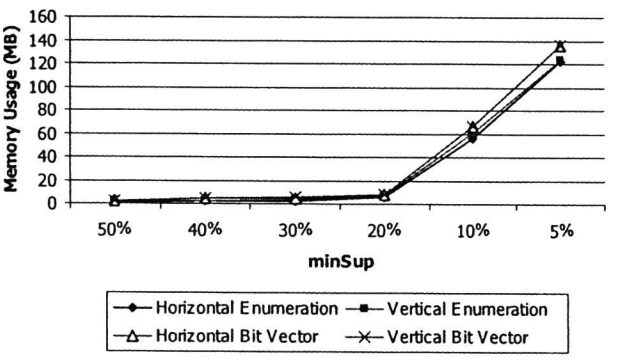


(c) DNA dataset

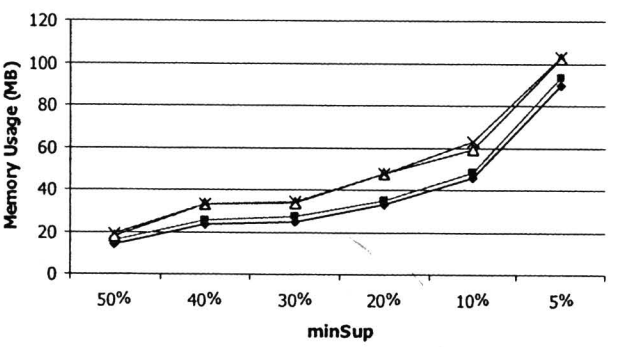


(d) Mushroom dataset

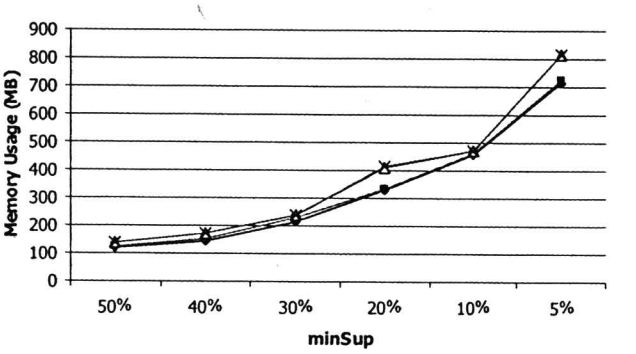
Fig. 6 The effect of data organization on speed



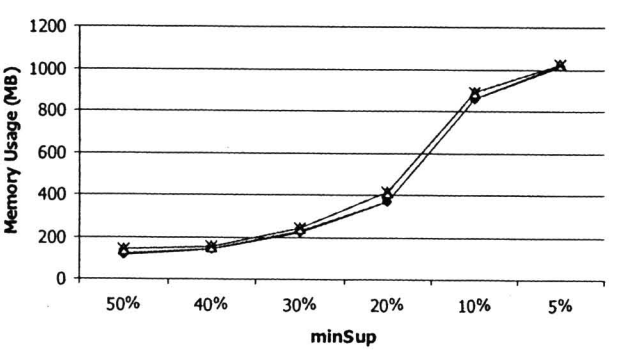
(a) Vote dataset



(b) Chess dataset



(c) DNA dataset



(d) Mushroom dataset

Fig. 7 The effect of data organization on memory usage

VI. Conclusion and Discussion

Association mining is one major problem in the area of data mining. The problem concerns finding frequent patterns hidden in a dataset. 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 dataset.

The idea to mine association rules 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 functional languages. Our supposition is that pattern matching is a fundamental feature supported by functional languages. The implementation of Apriori algorithm using Haskell confirms our hypothesis about conciseness of the program. The performance studies also support our intuition on efficiency because Haskell implementation outperforms the Java implementation in terms of speed and memory usage in every dataset.

This preliminary study supports our belief regarding functional programming paradigm towards frequent itemsets mining. 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. With the power of Haskell, this is a very promising extension

Acknowledgment

This work was supported in part by grants from National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF). Kittisak Kerdprasop is a director of Data Engineering and Knowledge Discovery (DEKD) research unit in which Nittaya Kerdprasop is also a member and a researcher of this research unit. DEKD is fully supported by Suranaree University of Technology.

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] A. Ceglar and J. Roddick, "Association mining," *ACM Computing Surveys*, vol. 38, no.2, 2006.

[4] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* (2nd ed.), Morgan Kaufmann, 2006.

[5] P. Hudak, J. Fasel, and J. Peterson, "A gentle introduction to Haskell," Yale University, *Technical Report Yale U/DCS/RR-901*, 1996.

[6] P. Jones and J. Hughes (eds.), *Standard Libraries for the Haskell 98 Programming Languages*. Available: http://www.haskell.org/library/.

[7] 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.

[8] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison Wesley, 2005.

[9] S. Thompson, *Haskell: The Craft of Functional Programming* (2nd ed.), Addison Wesley, 1999.

[10] M. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2003, pp. 326–335.

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.

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.

Moving Data Mining Tools toward a Business Intelligence System

Nittaya Kerdprasop, and Kittisak Kerdprasop

***Abstract*—**Data mining (DM) is the process of finding and extracting frequent patterns that can describe the data, or predict unknown or future values. These goals are achieved by using various learning algorithms. Each algorithm may produce a mining result completely different from the others. Some algorithms may find millions of patterns. It is thus the difficult job for data analysts to select appropriate models and interpret the discovered knowledge. In this paper, we describe a framework of an intelligent and complete data mining system called SUT-Miner. Our system is comprised of a full complement of major DM algorithms, pre-DM and post-DM functionalities. It is the post-DM packages that ease the DM deployment for business intelligence applications.

***Keywords*—**Business intelligence, data mining, functional programming, intelligent system.

I. Introduction

DATA mining (DM) or *Knowledge Discovery in Databases* (KDD) has been defined [3] as the automatic discovery of previously unknown patterns or relationships in large and complex datasets. Most DM algorithms have been drawn from the areas of Statistics and Machine Learning adapted to induce knowledge from data contained within a database. The main objective of DM is to use the discovered knowledge for the purposes of explaining current behavior, predicting future outcomes, or providing support for business decision. The DM techniques used in business-oriented applications are also known as *Business Intelligence* (BI). BI is a general term to mean all processes, techniques, and tools that gather and analyze data for the purpose of supporting enterprise users to make better decisions [1], [7].

Despites its high claims and expectations, DM technology requires a highly trained professional to do an iterative, multi-step process of accessing and preparing data, choosing an appropriate algorithm to mine the data, analyzing the learned knowledge, and presenting nontrivial, valuable knowledge to executives or decision makers. Owing to advancement in the machine leaning research, mining can be done efficiently on a dataset of large size. A hindrance of DM employment as an automatic knowledge acquisition tool in BI is the part of post-mining evaluation to obtain only a relevance and valuable knowledge.

The difficulty of discovering and deploying new knowledge in the BI context is due to the lack of intelligent and complete DM system. Most DM packages are comprised of learning algorithms integrated into a visual environment. Such graphical environment is a useful facility for experienced data analysts or data miners, but it provides limited functionalities for a novice to interpret and evaluate significance of the mining results.

As an example, consider Fig. 1 that shows the three different mining results obtained from three rule-induction algorithms: Ripple-Down Rule Learner, Ripper (Repeated Incremental Pruning to Produce Error Reduction), PART. The dataset is taken from the credit card promotion database [9]. The mining objective is to learn a profile for individuals likely to take advantage of a life insurance promotion advertised along with their credit card statement. A learned profile can help the credit card company to send the promotion materials only to a select group of individuals who are likely to take advantage of a life insurance promotion.

Income Range	Magazine Promotion	Watch Promotion	Life Insurance Promotion	Credit Card Insurance	Sex	Age
40-50K	Yes	No	No	No	Male	45
30-40K	Yes	Yes	Yes	No	Female	40
40-50K	No	No	No	No	Male	42
30-40K	Yes	Yes	Yes	Yes	Male	43
50-60K	Yes	No	Yes	No	Female	38
20-30K	No	No	No	No	Female	55
30-40K	Yes	No	Yes	Yes	Male	35
20-30K	No	Yes	No	No	Male	27
30-40K	Yes	No	No	No	Male	43
30-40K	Yes	Yes	Yes	No	Female	41
40-50K	No	Yes	Yes	No	Female	43
20-30K	No	Yes	Yes	No	Male	29
50-60K	Yes	Yes	Yes	No	Female	39
40-50K	No	Yes	No	No	Male	55
20-30K	No	No	Yes	Yes	Female	19

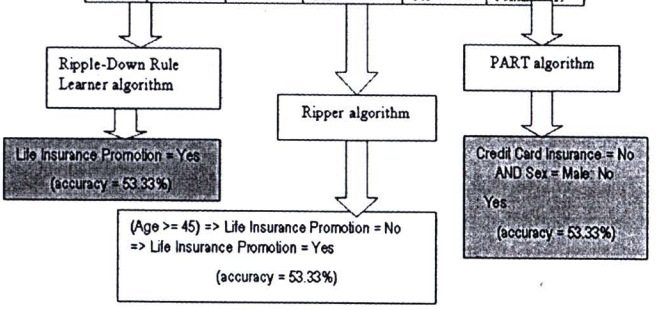


Fig. 1 Different mining results obtained from different algorithms

Manuscript received November 30, 2006. This work was supported in part by the research fund from Suranaree University of Technology and the grants from the National Research Council of Thailand, the Thailand Research Fund.

Nittaya Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224432; fax: +66-44-224602; e-mail: nittaya@ sut.ac.th, nittaya.k@gmail.com).

Kittisak Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (e-mail: kerdpras@sut.ac.th).

The mining results (*i.e.* classification models) shown in fig. 1 obtained from the three runs on WEKA system [10], [11]. The three classification models show the same accuracy when tested with 10-fold cross validation technique, but all three models produce different knowledge. The knowledge conveyed by each model can be explained in Fig. 2.

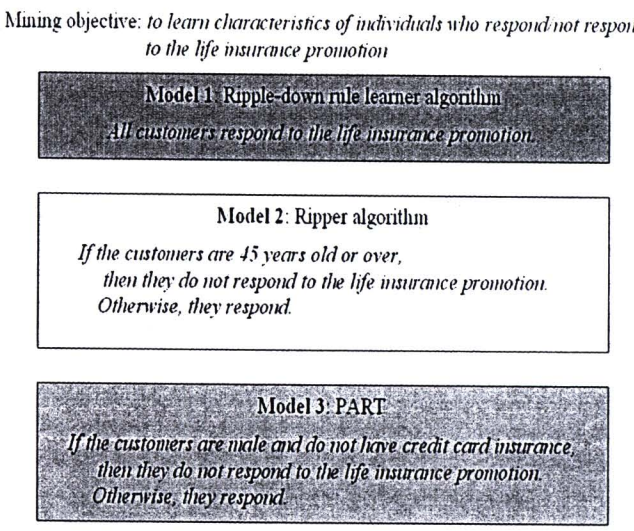


Fig. 2 Interpretation on mining results

It can be seen that even though model 1 is as accurate as models 2 and 3, it provides less knowledge on predicting the response of future customers. It is, however, the burden of the data analyst to choose either model 2 or 3, or even to combine both models as new knowledge for the task of customer segmentation.

To choose and use the mining results are not a trivial task because the model-selection facilities provided by most DM packages are very rare or limited. It is, therefore, the objective of this research to design and implement a complete framework of a knowledge mining system called the *SUT-Miner* (it is named after the sponsor of this project, *i.e.* Suranaree University of Technology). We design the system to be both intelligent and complete with the full functionalities of pre-DM, DM, and post-DM. The system is an extension of the work presented in [5], [6].

This paper presents work in progress of the development of SUT-Miner system. The implementation is based on the functional paradigm using the Haskell language. The organization of this paper is as follows. Section 2 provides a brief explanation of DM tasks that are included in the design of our system. Section 3 sketches the overall architecture of the system. Section 4 explains the implementation and the experimental results showing the advantage of functional programming over the object-oriented programming. Section 5 concludes the paper and suggests an extension of the system towards an approximate and progressive scheme.

II. Data Mining Conceptual Model

DM is about learning patterns. *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.

According to [3], DM involves fitting models to, or determining patterns from, observed data. Primary goals of DM are prediction and description.

Prediction uses supervised learning technique to predict values of data using known values found from different data. DM tasks for prediction include classification, regression, time-series analysis.

Description focuses on employing unsupervised learning technique to find human-interpretable patterns describing the data. DM tasks for description are clustering, summarization, association, and sequence discovery.

To start building a DM methodology, it is necessary to set up a conceptual framework into which data and its structures might be classified. The terminology to denote structures of the dataset is summarized in Fig. 3.

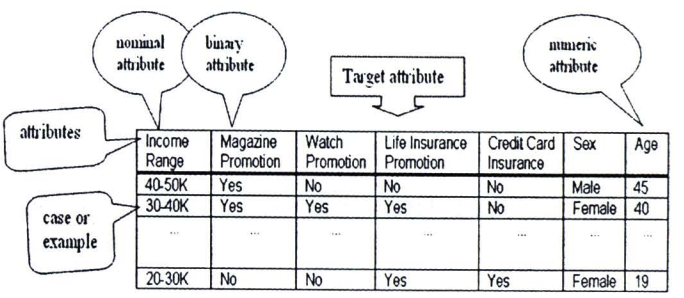


Fig. 3 Summarization of a dataset terminology

We adopt the ontology of DM methodology proposed by [8] to abstract data and their relationships for guiding the DM method selection. The simple ontology presented in Fig. 4 provides a means to explicitly guide the novice data miners towards DM task selection. This guideline is data-driven in that the data types and structures are used as a basis for selecting appropriate DM method.

Our data mining system consists of three main parts: pre-DM, DM, and post-DM. The ontology presented in Fig. 4 is for a method selection in the DM part. We also need complete ontology for the parts of pre-DM and post-DM as well as the full specification of each DM task. These will be our future work.

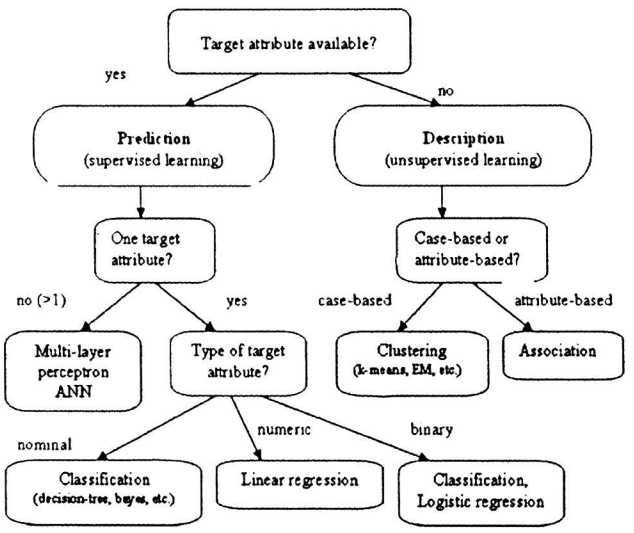


Fig. 4 A simple ontology for DM method selection

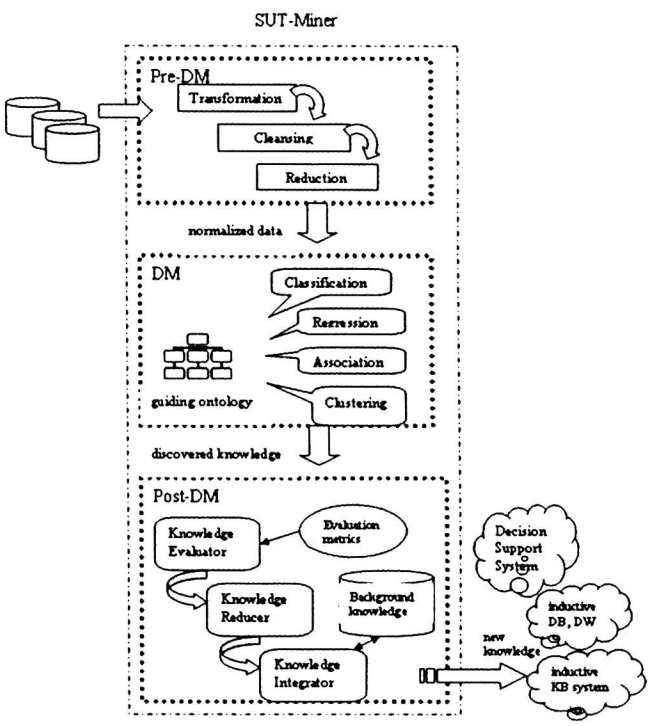


Fig. 5 The architecture of SUT-Miner system

III. An Overall Architecture of SUT-Miner

At a high level of our framework, we design the SUT-Miner system to be comprised of three main phases: pre-DM, DM, 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, prediction, clustering, and association. The post-DM phase involves 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 which is important for the applications in BI. The overall architecture of our SUT-Miner system is presented in Fig. 5.

IV. Rapid Prototyping with Haskell

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 following example shows C versus Haskell codes to compute a list of fibonacci numbers starting with zero.

C-code

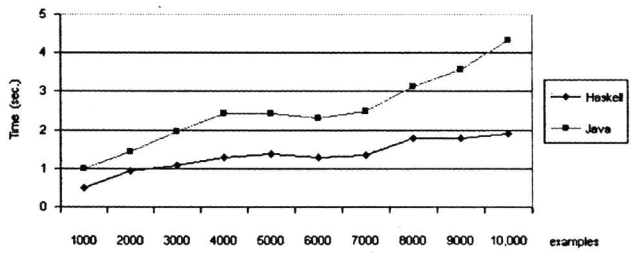
```
int * fib (int n)
  { int a = 0, b = 1, i, temp;
    int * fibsequence;
    fibsequence = (int *) malloc ((sizeof int) *n);
    for (i = 0; i<n; i++)
      { fibsequence[i] = a;
        temp = a + b;
        a = b;
        b = temp;
      }
    return fibsequence;
  }
```

Haskell-code

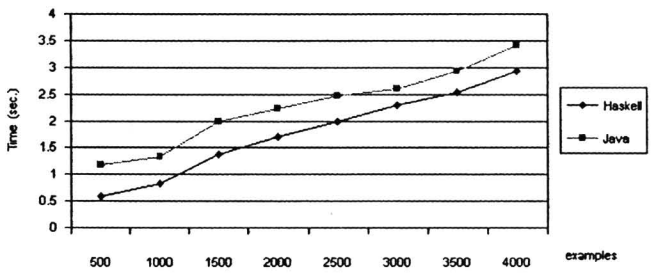
```
fib :: [Int]
fib = 0: 1: [ a+b | (a, b) <- zip fib (tail fib) ]
```

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. According to our experimentation, the speed of running Haskell program on a moderate-size dataset is quite impressive. The experimental results shown in Fig. 6 compare the running time of a Haskell program against a Java program for mining a linear regression model. The first experiment

computes a regression model of two variables. The number of variables is increased to six in the second experiment. The experimentations are performs on a computer notebook with CPU speed 1.8 GHz and main memory 512 MB.



(a) mining regression model of two variables



(b) mining regression model of six variables

Fig. 6 Mining regression model with Haskell and Java

V. Conclusion and Future Work

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.

Appendix

The Haskell code to mine two-variable and six-variable regression models is provided here.

```
-- ------------------------------------------------------
main = do
   hSetBuffering stdin LineBuffering
   ex <- readArff  --ex is example
   let exs = read ex::[[Float]]
   let n = length (head (read ex::[[Float]]))
   if (n==2)   then solution_2 exs
      else if (n==3)    then solution_3 exs
         else if (n==4)then solution_4 exs
            else if (n==5)then solution_5 exs
               else if (n==6)then solution_6 exs
write x = do
   hdl <- openFile "matrix.txt" WriteMode
   hPutStr hdl x
   hClose hdl

-- Funtion for Solutions
-- Input/Output function

multi :: [Float]->[Float]->[Float]
my_length :: [[a]]->Float
sumsq :: [Float]->[Float]

my_length [] = 0
my_length (x:xs) = 1 + my_length xs

sumsq [] = []
sumsq (x:xs) = [x*x] ++ sumsq xs

multi [][]= []
multi (x:xs)(y:ys) = [x*y] ++ multi (xs)(ys)

-- s2_Solution for Exponential Regression
--
solution_2 ex = do
   let aa = 2
   let ab = s2_sum_x ex
   let ba = s2_sum_x ex
   let bb = s2_sum_x2 ex
   let ya = s2_sum_y ex
   let yb = s2_sum_xy ex

   let line_1 = "2\n"
   let line_2 = show aa ++ "," ++ show ab ++ "\n"
   let line_3 = show ba++","++ show bb ++"\n"
   let line_4 = "\n"
   let line_5 = show ya ++","++ show yb
   write (line_1++line_2++line_3++line_4++line_5)

s2_sum_x :: [[Float]] -> Float
s2_sum_y :: [[Float]] -> Float
s2_sum_x2 :: [[Float]] -> Float
s2_sum_multi :: [[Float]] -> Float
s2_form_x :: [[Float]]->[Float]
s2_form_y :: [[Float]]->[Float]

s2_form_x [] = []
s2_form_x [[a,b]] = [a]
s2_form_x (x:xs) = s2_form_x [x] ++ s2_form_x xs

s2_form_y [] = []
s2_form_y [[a,b]] = [b]
s2_form_y (y:ys) = s2_form_y [y] ++ s2_form_y ys
```



```
s2_sum_x [] = 0
s2_sum_x xs = sum(s2_form_x xs)

s2_sum_y [] = 0
s2_sum_y ys = sum(s2_form_y ys)

s2_sum_x2 [] = 0
s2_sum_x2 xs = sum(sumsq(s2_form_x xs))

s2_sum_xy []= 0
s2_sum_xy  s = sum(multi (s2_form_x s)(s2_form_y s))

s2_sum_multi [] = 0
s2_sum_multi s = sum(multi (s2_form_x s)(s2_form_y s))

-- s5_Solution for Exponential Regression
--
solution_6 ex = do
   let aa = 6
   let ab = s6_sum_x1i ex
   let ac = s6_sum_x2i ex
   let ad = s6_sum_x3i ex
   let ae = s6_sum_x4i ex
   let af = s6_sum_x5i ex

   let ba = s6_sum_x1i ex
   let bb = s6_sum_x1i2 ex
   let bc = s6_sum_x1x2 ex
   let bd = s6_sum_x1x3 ex
   let be = s6_sum_x1x4 ex
   let bf = s6_sum_x1x5 ex

   let ca = s6_sum_x2i ex
   let cb = s6_sum_x1x2 ex
   let cc = s6_sum_x2i2 ex
   let cd = s6_sum_x2x3 ex
   let ce = s6_sum_x2x4 ex
   let cf = s6_sum_x2x5 ex

   let da = s6_sum_x3i ex
   let db = s6_sum_x1x3 ex
   let dc = s6_sum_x2x3 ex
   let dd = s6_sum_x3i2 ex
   let de = s6_sum_x3x4 ex
   let df = s6_sum_x3x5 ex

   let ea = s6_sum_x4i ex
   let eb = s6_sum_x1x4 ex
   let ec = s6_sum_x2x4 ex
   let ed = s6_sum_x3x4 ex
   let ee = s6_sum_x4i2 ex
   let ef = s6_sum_x4x5 ex

   let fa = s6_sum_x5i ex
   let fb = s6_sum_x1x5 ex
   let fc = s6_sum_x2x5 ex
   let fd = s6_sum_x3x5 ex
   let fe = s6_sum_x4x5 ex
   let ff = s6_sum_x5i2 ex

   let ya = s6_sum_yi ex
   let yb = s6_sum_x1iyi ex
   let yc = s6_sum_x2iyi ex
   let yd = s6_sum_x3iyi ex
   let ye = s6_sum_x4iyi ex
   let yf = s6_sum_x5iyi ex

   let line_1 = "3\n"
   let line_2 = show aa++","++ show ab ++ ","++ show ac ++
        show ad ++ "," ++ show ae ++ "," ++ show af ++"\n"
   let line_3 = show ba++","++ show bb ++ ","++ show bc ++
        show bd ++ "," ++ show be ++ "," ++ show bf ++"\n"
   let line_4 = show ca++","++ show cb ++ ","++ show cc ++
        show cd ++ "," ++ show ce ++ "," ++ show cf ++"\n"
   let line_5 = show da++","++ show db ++ ","++ show dc ++
        show dd ++ "," ++ show de ++ "," ++ show df ++"\n"
   let line_6 = show ea++","++ show eb ++ ","++ show ec ++
        show ed ++ "," ++ show ee ++ "," ++ show ef ++"\n"
   let line_7 = show fa++","++ show fb ++ ","++ show fc ++
        show fd ++ "," ++ show fe ++ "," ++ show ff ++"\n"
   let line_8 = "\n"
   let line_9 = show ya++","++ show yb ++ ","++ show yc ++
        ","++ show yd ++ ","++ show ye ++ "," ++ show yf
   write (line_1++line_2++line_3++line_4++line_5++line_6++
        line_7++line_8++line_9)

s6_form_x1i [] = []
s6_form_x1i [[a,b,c,d,e,f]] = [a]
s6_form_x1i (x:xs) = s6_form_x1i [x] ++ s6_form_x1i xs

s6_form_x2i [] = []
s6_form_x2i [[a,b,c,d,e,f]] = [b]
s6_form_x2i (x:xs) = s6_form_x2i [x] ++ s6_form_x2i xs

s6_form_x3i [] = []
s6_form_x3i [[a,b,c,d,e,f]] = [c]
s6_form_x3i (x:xs) = s6_form_x3i [x] ++ s6_form_x3i xs

s6_form_x4i [] = []
s6_form_x4i [[a,b,c,d,e,f]] = [d]
s6_form_x4i (x:xs) = s6_form_x4i [x] ++ s6_form_x4i xs

s6_form_x5i [] = []
s6_form_x5i [[a,b,c,d,e,f]] = [e]
s6_form_x5i (x:xs) = s6_form_x5i [x] ++ s6_form_x5i xs

s6_form_yi [] = []
s6_form_yi [[a,b,c,d,e,f]] = [f]
s6_form_yi (x:xs) = s6_form_yi [x] ++ s6_form_yi xs

s6_sum_x1i [] = 0
s6_sum_x1i s = sum(s6_form_x1i s)

s6_sum_x2i [] = 0
s6_sum_x2i s = sum(s6_form_x2i s)

s6_sum_x3i [] = 0
s6_sum_x3i s = sum(s6_form_x3i s)

s6_sum_x4i [] = 0
s6_sum_x4i s = sum(s6_form_x4i s)

s6_sum_x5i [] = 0
s6_sum_x5i s = sum(s6_form_x5i s)

s6_sum_yi [] = 0
s6_sum_yi s = sum(s6_form_yi s)

s6_sum_x1i2 [] = 0
s6_sum_x1i2 s = sum(sumsq(s6_form_x1i s))

s6_sum_x2i2 [] = 0
s6_sum_x2i2 s = sum(sumsq(s6_form_x2i s))
```



```
s6_sum_x3i2 [] = 0
s6_sum_x3i2 s = sum(sumsq(s6_form_x3i s))

s6_sum_x4i2 [] = 0
s6_sum_x4i2 s = sum(sumsq(s6_form_x4i s))

s6_sum_x5i2 [] = 0
s6_sum_x5i2 s = sum(sumsq(s6_form_x5i s))

s6_sum_x1iyi []= 0
s6_sum_x1iyi s = sum(multi (s6_form_x1i s)(s6_form_yi s))

s6_sum_x2iyi []= 0
s6_sum_x2iyi s = sum(multi (s6_form_x2i s)(s6_form_yi s))

s6_sum_x3iyi []= 0
s6_sum_x3iyi s = sum(multi (s6_form_x3i s)(s6_form_yi s))

s6_sum_x4iyi []= 0
s6_sum_x4iyi s = sum(multi (s6_form_x4i s)(s6_form_yi s))

s6_sum_x5iyi []= 0
s6_sum_x5iyi s = sum(multi (s6_form_x5i s)(s6_form_yi s))

s6_sum_x1x2 [] = 0
s6_sum_x1x2 s = sum(multi (s6_form_x1i s)(s6_form_x2i s))

s6_sum_x1x3 [] = 0
s6_sum_x1x3 s = sum(multi (s6_form_x1i s)(s6_form_x3i s))

s6_sum_x1x4 [] = 0
s6_sum_x1x4 s = sum(multi (s6_form_x1i s)(s6_form_x4i s))

s6_sum_x1x5 [] = 0
s6_sum_x1x5 s = sum(multi (s6_form_x1i s)(s6_form_x5i s))

s6_sum_x2x3 [] = 0
s6_sum_x2x3 s = sum(multi (s6_form_x2i s)(s6_form_x3i s))

s6_sum_x2x4 [] = 0
s6_sum_x2x4 s = sum(multi (s6_form_x2i s)(s6_form_x4i s))

s6_sum_x2x5 [] = 0
s6_sum_x2x5 s = sum(multi (s6_form_x2i s)(s6_form_x5i s))

s6_sum_x3x4 [] = 0
s6_sum_x3x4 s = sum(multi (s6_form_x3i s)(s6_form_x4i s))

s6_sum_x3x5 [] = 0
s6_sum_x3x5 s = sum(multi (s6_form_x3i s)(s6_form_x5i s))

s6_sum_x4x5 [] = 0
s6_sum_x4x5 s = sum(multi (s6_form_x4i s)(s6_form_x5i s))
--
-- ------------------------------------------------------------------
```

ACKNOWLEDGMENT

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 first author would like to thank all the research assistants who participated in the SUT-Miner project, especially Prapanpong Nopsuwan who coded linear regression in Haskell and Java.

REFERENCES

[1] E. Awad and H. Ghaziri, *Knowledge Management*, Pearson Prentice Hall, 2004.
[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] K. Kerdprasop and N. Kerdprasop, "Multi-agents in data filtering systems," in *Proc. 7th Int. Conf. on Software Engineering and Applications*, 2003, pp.471-475.
[6] N. Kerdprasop and K. Kerdprasop, "Enhancing the power of OLAP with knowledge discovery," in *Proc. 7th Int. Conf. on Software Engineering and Applications*, 2003, pp.43-47.
[7] M. Raisinghani (ed.), *Business Intelligence in the Digital Economy*, Idea Group Publishing, 2004.
[8] K. Rennolls, "An intelligent framework (O-SS-E) for data mining, knowledge discovery and business intelligence," in *Proc. 16th Int. Workshop on Database and Expert System Applications*, 2005, pp.715-719.
[9] R. Roiger and M. Geatz, *Data Mining: A Tutorial-Based Primer*, Addison Wesley, 2003.
[10] WEKA, available at http://www.cs.waikato.ac.nz/ml/weka
[11] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques* (2nd ed.), Morgan Kaufmann, 2005.

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.

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.

Automated induction of frequent patterns with knowledge-based software engineering

Kittisak Kerdprasop and Nittaya Kerdprasop

Data Engineering and Knowledge Discovery (DEKD) Research Unit
School of Computer Engineering, Suranaree University of Technology
Nakhon Ratchasima 30000 Thailand
kittisakThailand@gmail.com, nittaya@sut.ac.th

Abstract

Frequent patterns refer to the occurrences of some data items frequently found together in the database. Automated induction of frequent patterns, also known as frequent pattern mining, is among major research topics in the area of data mining for which the efficient and effective mining techniques have been sought. In this paper, we study the problem of frequent pattern mining within the context of knowledge-based software engineering. The term knowledge-based software engineering has emerged as a cross discipline of software engineering and artificial intelligence to solve a crisis of software productivity due to difficulties associated with the engineering of complex software systems. We propose a knowledge discovery tool to capture frequent patterns in the educational data sets. Our automated tool was implemented with a higher-order logic-programming scheme so that the knowledge-intensive tasks can be efficiently coded. The implementation demonstrated in this paper can also be extended without much effort to support knowledge caption and representation in order to automatically generate knowledge content in the knowledge-base system.

1 Introduction

Frequent-pattern mining is the discovery of 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 $DB = \{C_1, C_2, C_3, ..., C_n\}$ be a database of n instances and each data instance contains items in the set I. A *pattern* is a set of items that occur in a data instance. The number of items in a pattern is called the length of the pattern. To search for all valid patterns of length 1 up to m in large database is computational expensive. For a set I of m different items, the search space for all distinct patterns can be as huge as 2^m-1. To reduce the size of the search space, the *support* measurement has been introduced [1]. The function *support*(P) of a pattern P is defined as a number of instances in DB containing P. Thus, *support*(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. The *minS* metric has an anti-monotone property and is applied as a basis for reducing search space of mining frequent patterns in the well-known algorithm Apriori [1].

A logic-based approach to the development of knowledge discovery system has long been an interesting research topic among data mining and machine learning researchers. For the classification task, tree-based concept induction [4, 8] and rule induction [5] are major approaches normally adopted. Frequent-pattern mining task was mostly based on the well-known APRIORI algorithm [1]. WARMR system [2] upgraded APRIORI algorithm to discover frequent patterns. Its extension [3] was developed to discover frequent Datalog patterns and relational association rules. Our implementation approach is also based on the mathematical logic concepts, but we extend the predicate terms to the level of higher-order logic.

2 Higher-order logic programming

In logic programming, a clause is a disjunction of literals (atomic symbols or their negations) such as $p \vee q$ and $\neg p \vee r$. A statement is in clausal form if it is a conjunction of clauses such as $(p \vee q) \wedge (\neg p \vee r)$. Logic programming is a subset of first order logic in which clauses are restricted to Horn clauses. A Horn clause, named after the logician Alfred Horn [7], is a clause that contains at most one positive literal such as $\neg p \vee \neg q \vee r$. Horn clauses are widely used in logic programming because their satisfiability property can be solved by resolution algorithm (an inference method for checking whether the formula can be evaluated to true).

A Horn clause with no positive literal, such as $\neg p \vee \neg q$, which is equivalent to $\neg (p \wedge q)$, is called *query* in Prolog and can be interpreted as '*:- p, q*' in which its value (true/false) to be proven by resolution method. A clause that contains exactly one positive literal such as *r* is called a *fact* representing a true statement, written in clausal form as '*r :-*' in which the condition part is empty and that means *r* is unconditionally true. Therefore, facts are used to represent data. A Horn clause that contains one positive literal and one or more negative literals such as $\neg p \vee \neg q \vee r$ is called a *definite clause* and such clause can equivalently written as $(p \wedge q) \rightarrow r$ which in turn can be represented as a Prolog *rule* as *r :- p, q.* The symbol '*:-*' is intended to mean '$\leftarrow$', which is implication in first-order logic, and the symbol ',' represents the operator $\wedge$ (or 'AND'). In Prolog, rules are used to define procedures and a Prolog program is normally composed of facts and rules. Running a Prolog program is nothing more than posing queries to obtain true/false answers. The symbols *p*, *q*, *r* are called *predicates* in first-order logic programming and they can be quantified over variables such as *r(X) :- p(X,Y), q(Y).* This clause has the same meaning as $\forall X (p(X,Y) \wedge q(Y) \rightarrow r(X))$.The scope of variables is within a clause. Horn clauses are thus the fundamental concept of logic programming.

Higher-order predicate is a predicate that can quantify over other predicate symbols [6]. As an example, besides the rule *r(X):- p(X,Y), q(Y)*, if we are also given the following five Horn clauses (or facts): *p(1, 2). p(1, 3). p(5, 4). q(2). q(4).* Then by asking the query: *?- r(X)*, we will get the response as '*true*' and also the first instantiation as *X=1*. If we want to know all instantiations that make *r(X)* true, we may ask the query: *?- findall(X, r(X), Answer).* We will get the response: *Answer = [1, 5]*, which is a set of all answers obtained from the predicate *r(X)* according to the given facts. The predicate symbol *findall* quantifies over the variables *X*, *Answer*, and the predicate *r*. The predicate *findall* is thus called a higher-order predicate.

3 Frequent pattern mining with higher-order logic

We implemented the frequent-pattern mining program (Figure 1) based on the APRIORI algorithm [1]. Main predicate of this program is *frequent_pattern_mining*. Upon invocation, this predicate will obtain input data, also in the format of Prolog program, from the predicate *input(Data)*. The predicate *minS(V)* specifies the minimum value of support metric. Then the main predicate starts the automated induction process by searching candidate item sets and large item sets of length one, two, three, and so forth (through the predicates *makeC1*, *makeL*, and *apriori_loop*, respectively. All highlighted terms in Figure 1 are higher-order predicates. These predicates are *maplist*, *include*, and *setof*.

The program execution results on educational data sets are shown in Figure 2 (only the five patterns with highest support values are shown). These data sets are the dropout data at the secondary level (7^{th} grade – 12^{th} grade) in the school year 2002-04 reported by California state education agencies [9]. Each data instance is a report from each school district containing 15 attributes: Locale (location of the school), LO-offered (the lowest grade of the school), HI-offered (the highest grade offered by the school), the total number of student enrollments in grade 7^{th} through 12^{th}, and the total number of dropouts at grade 7^{th} through 12^{th}.

```
frequent_pattern_mining :- input(Data), minS(V), makeC1(C), makeL(C,L),
apriori_loop(L,1).
apriori_loop(L,N) :- length(L) is 1,!.
apriori_loop(L,N) :- N1 is N+1, makeC(N1,L,C), makeL(C, Res), apriori_loop(Res, N1).
makeC1(Ans) :- input(D), allComb(1, ItemSet, Ans2), maplist( countSS(D), Ans2, Ans).
makeC(N, ItemSet, Ans) :- input(D), allComb(2, ItemSet, Ans1),
        maplist(flatten, Ans1, Ans2), maplist(list_to_ord_set, Ans2, Ans3) ,
        list_to_set(Ans3,Ans4), include(len(N),Ans4,Ans5), maplist(countSS(D),Ans5,Ans).
  %scan database to find: List+N
makeL(C, Res) :- include(filter, C, Ans), maplist(head, Ans, Res).
filter(_+N) :- input(A), length(A, I), min_support(V), N>=(V/100)*I.
head(H+_, H).
   % arbitrary subset of the set containing given number of elements
comb(0, _, []).
comb(N, [X|T], [X|Comb]) :- N> 0, N1 is N-1, comb(N1, T, Comb).
comb(N, [_|T], Comb) :- N> 0, comb(N, T, Comb).
allComb(N, I, Ans) :- setof( L, comb(N, I, L), Ans).
countSubset(A, [], 0).
countSubset(A, [B|X], N) :- not(subset(A, B)), countSubset(A, X, N).
countSubset(A, [B|X], N) :- subset(A, B), countSubset(A, X, N1), N is N1+1.
countSS(SL, S, S+N) :- countSubset(S, SL, N).
len(N, X) :- length(X, N1), N is N1.
```

Figure 1. Frequent-pattern mining logic program implemented with higher-order predicates

School year 2002 (390 data instances)	LO-offered=KG & HI-offered=12	support=0.84
	HI-offered=12 & Grade-7-dropouts=0	support=0.76
	HI-offered=12 & Grade-8-dropouts=0	support=0.74
	LO-offered=KG & Grade-7-dropouts=0	support=0.74
	Grade-7-dropouts=0 & Grade-8-dropouts=0	support=0.71
School year 2003 (565 data instances)	LO-offered=KG & Grade-7-enrollment=[1-1000]	support=0.67
	LO-offered=KG & Grade-8-enrollment=[1-1000]	support=0.67
	Grade-7-enrollment=[1-1000] & Grade-8-enrollment=[1-1000]	support=0.66
	Grade-7-dropouts=0 & Grade-8-dropouts=0	support=0.65
	LO-offered=KG & Grade-7-dropouts=0	support=0.63
School year 2004 (555 data instances)	LO-offered=KG & Grade-7-enrollment=[1-1000]	support=0.67
	Grade-7-enrollment=[1-1000] & Grade-8-enrollment=[1-1000]	support=0.67
	LO-offered=KG & Grade-8-enrollment=[1-1000]	support=0.66
	LO-offered=KG & HI-offered=12	support=0.59
	Grade-7-dropouts=0 & Grade-8-dropouts=0	support=0.59

Figure 2. Running results of frequent-pattern mining program on data sets of California school dropouts during the school years 2002-04

4 Conclusions

Logic programming is a declarative style of writing programs. It is a very high-level language in the sense that it focuses on the computation's logic rather than the mechanics. Logic programming languages such as Prolog are admittedly suitable for rapid prototyping of new ideas or new program architecture. This programming paradigm is based on a solid foundation of first-order logic in which the logical properties of a computation and the coverage of predicates and quantification are emphasized. First-order logic, however, poses a restriction on the type of variables appearing in quantification not to include predicates. Higher-order logic, on the contrary, allows variables to quantify over predicates. With such relaxation, higher-order logic facilitates the implementation of a knowledge-intensive application such as frequent pattern mining, which is the discovery of frequent recurring correlations between data items in the database.

In this paper, we demonstrated a technique to implement frequent-pattern mining algorithm with the higher-order logic concept. The running examples were also illustrated through the educational data sets in which strong patterns of California state school dropout students during the year 2002-04 had been discovered. The pattern-mining program presented in this paper can also be applied to data sets in other domains.

Acknowledgements

This research has been funded by grants from the National Research Council of Thailand (NRCT) and the first author is supported by the Thailand Research Fund (TRF, grant number RMU5080026). DEKD has been fully supported by Suranaree University of Technology.

References

[1] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., & Verkamo, A. (1996). Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp.307-328. AAAI Press.

[2] Dehaspe, L. & Toivonen, H. (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1): 7-36.

[3] Dehaspe, L. & Toivonen, H. (2001). Discovery of relational association rules. In S. Dzeroski and N. Lavrac (Eds.), *Relational Data Mining*, pp.189-212. Springer.

[4] Kramer, S. & Widmer, G. (2001). Inducing classification and regression trees in first order logic. In S. Dzeroski and N. Lavrac (Eds.), *Relational Data Mining*, pp.140-159. Springer.

[5] Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, 13:245-286.

[6] Naish, L. (1996). Higher-order logic programming in Prolog. *Technical Report 96/2*, Department of Computer Science, University of Melbourne, Australia.

[7] Nienhuys-Cheng, S.-H. & Wolf, R. (1997). *Foundations of Inductive Logic Programming*. Springer.

[8] Quinlan, J. & Cameron-Jones, R. (1993). FOIL: A midterm report. *Proceedings of the 6th European Conference on Machine Learning*, LNAI 667, pp.3-20. Springer.

[9] Sable, J, & Gaviola, N. (2007). *NCES Common Core of Data Local Education Agency-Level Public-Use Data File on Public School Dropouts: School Year 2002-04*. National Center for Education Statistics, Institue of Education Sciences, U.S. Department of Education, Washington, D.C. (http://nces.ed.gov/pubsearch/)

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.

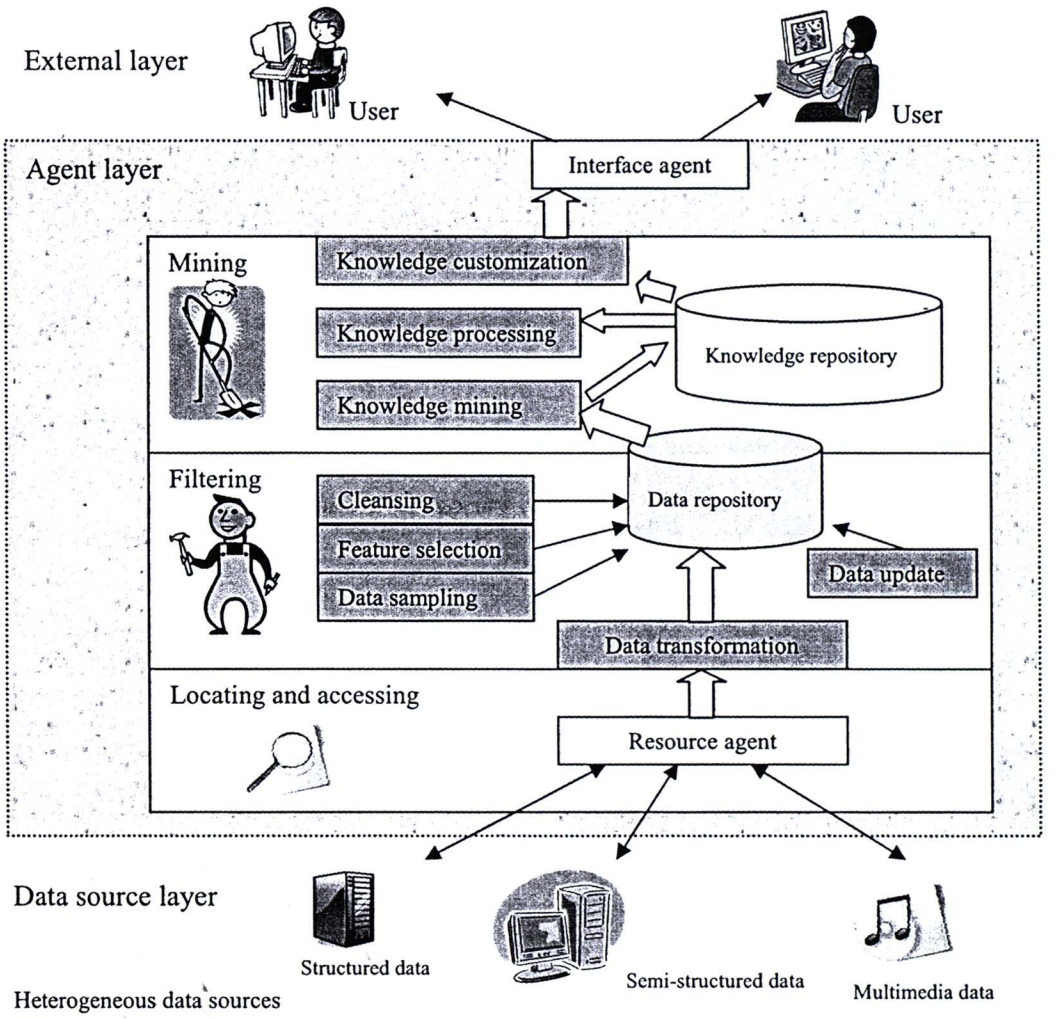


Figure 2. Agent-based model in a knowledge discovery system.

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

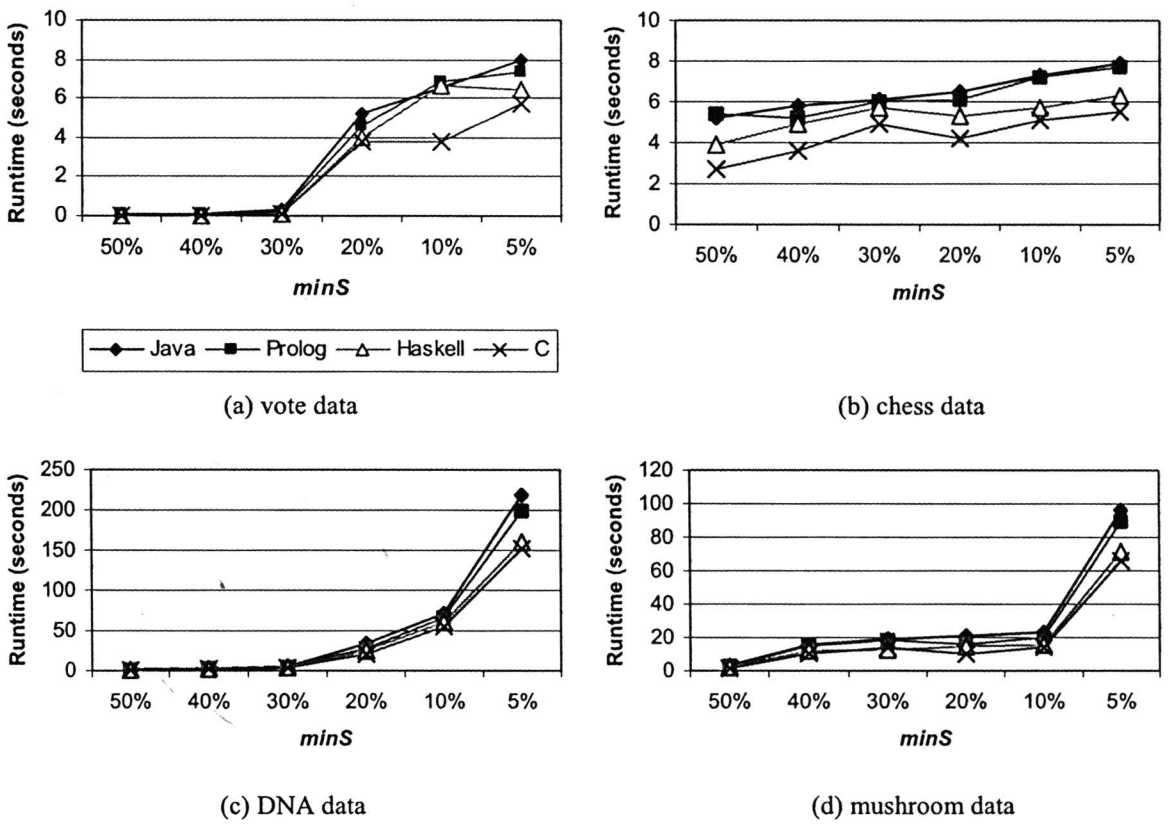
```
r1:- n(X), cL1(X).
r2(X):- cC2(X).
clear:-retractall(l1(_)),  retractall(c1(_)),
      retractall(c2(_)) , retractall(l2(_)).
% Create L1
cL1([]).
cL1([H|T]) :-  findall(X,f([H],X),L), length(L, Len),
               Len >= 2 , !,
               cL1(T), assert(l1(([H], Len)))
               ;
   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

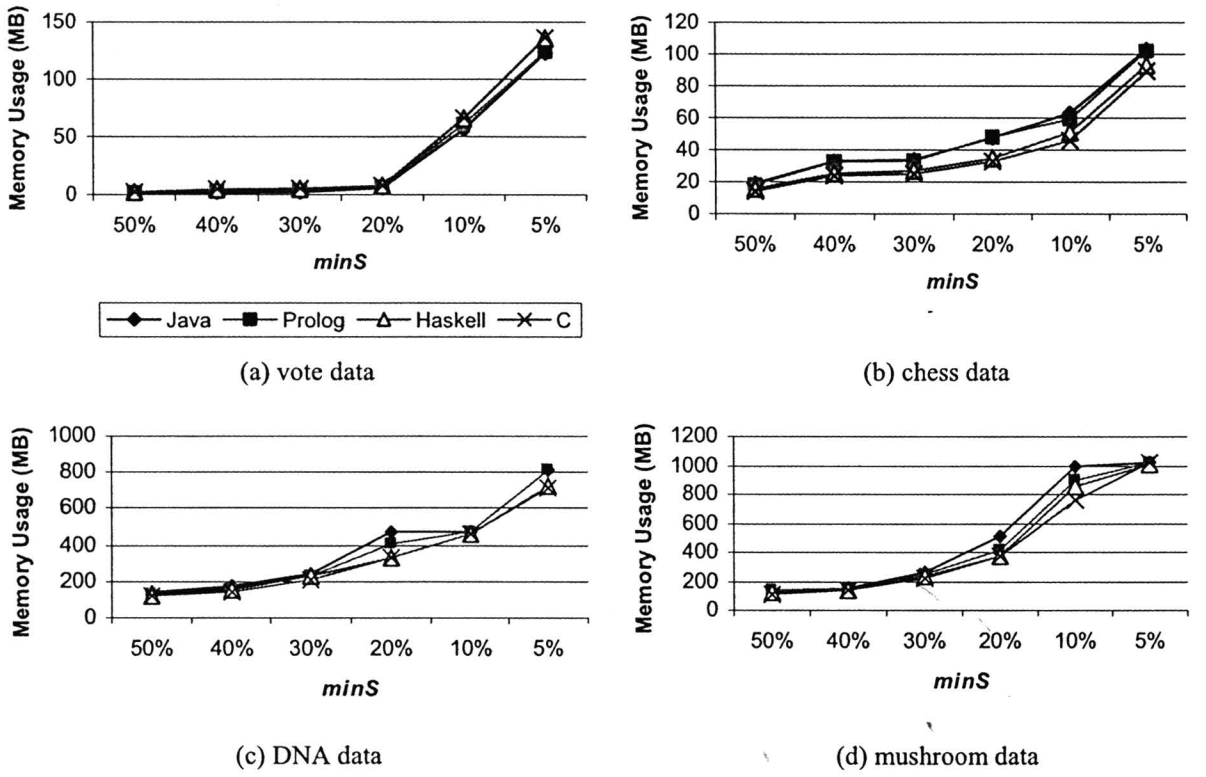


Figure 5. Memory usage comparison of declarative versus imperative programming.

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.

ภาคผนวก ข

บทความสำหรับการเผยแพร่

การค้นพบรูปแบบที่ปรากฏบ่อยด้วยวิธีการโปรแกรมเชิงประกาศ

Frequent pattern discovery with declarative programming

กิตติศักดิ์ เกิดประสพ[1] และ นิตยา เกิดประสพ

Kittisak Kerdprasop and Nittaya Kerdprasop

บทคัดย่อ

การค้นพบรูปแบบที่ปรากฏบ่อยเป็นกระบวนการค้นหาแพทเทิร์นที่ปรากฏซ้ำๆ ในข้อมูล แพทเทิร์นเหล่านี้มีประโยชน์ในการระบุความสัมพันธ์ที่ซ่อนอยู่ในกลุ่มข้อมูล อัลกอริทึมค้นหารูปแบบที่ปรากฏบ่อยในข้อมูลมักจะได้รับการพัฒนาเป็นโปรแกรมในลักษณะของการโปรแกรมเชิงกระบวนคำสั่ง ซึ่งถ้าแพทเทิร์นมีความซับซ้อนหรือมีขนาดที่ยาวมาก วิธีการโปรแกรมเชิงกระบวนคำสั่งจะมีประสิทธิภาพด้อยลง ผู้วิจัยจึงได้เสนอแนวทางของการโปรแกรมเชิงประกาศซึ่งคาดว่าจะมีประสิทธิภาพสูงกว่า โดยได้ใช้ภาษาฮาสเกลและภาษาโปรล็อกในการพัฒนาโปรแกรมเพื่อค้นหารูปแบบที่ปรากฏบ่อย การทดสอบผลใช้วิธีการเปรียบเทียบจำนวนบรรทัดในซอร์สโค้ด เวลาที่ใช้ในการรันโปรแกรม และเนื้อที่หน่วยความจำที่ใช้

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.

Key Word: Frequent pattern discovery, declarative programming, Haskell, Prolog

E-mail address: kittisakThailand@gmail.com

[1] หน่วยวิจัยวิศวกรรมข้อมูลและการค้นหาความรู้ สาขาวิชาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยเทคโนโลยีสุรนารี

Data Engineering and Knowledge Discovery Research Unit, School of Computer Engineering, Suranaree Univ. of Tech.

บทนำ

การค้นพบรูปแบบที่ปรากฏบ่อย หรือ frequent pattern discovery เป็นการค้นหารูปแบบที่เกิดขึ้นซ้ำๆ ในข้อมูลขนาดใหญ่ เช่น การค้นพบจากฐานข้อมูลทรานแซคชันของห้างสรรพสินค้าสาขาบางแค พบว่าลูกค้านิยมซื้อนมผงและชาเขียว ในขณะที่การค้นพบจากทรานแซคชันของสาขานครราชสีมาระบุว่าลูกค้าซื้อนมผงพร้อมกับไข่ไก่ รูปแบบการซื้อสินค้าดังตัวอย่างนี้จะช่วยสะท้อนถึงพฤติกรรมของผู้บริโภคซึ่งอาจจะแตกต่างกันในแต่ละภูมิภาค ความรู้เช่นนี้จะเป็นประโยชน์ต่อผู้บริหารในการวางแผนจัดวางชั้นสินค้า รวมไปถึงการวางแผนจัดรายการกระตุ้นยอดขายสินค้า เช่นจากพฤติกรรมการบริโภคของลูกค้าในย่านบางแค ถ้าผู้จัดการห้างสรรพสินค้าทราบว่ากำไรต่อหน่วยของชาเขียวสูงกว่านมผงมาก การตัดสินใจลดราคานมผงให้ต่ำกว่าราคาจำหน่ายของร้านค้าทั่วไปจึงคาดได้ว่าน่าจะดึงดูดลูกค้าที่ต้องการซื้อนมผงเข้ามาซื้อของในห้างสรรพสินค้า และคาดหมายต่อไปได้ว่าลูกค้าที่ซื้อนมผงจะซื้อชาเขียวร่วมด้วย ซึ่งจะส่งผลให้ยอดขายและกำไรโดยรวมของการจำหน่ายนมผงและชาเขียวสูงขึ้น

การค้นพบความสัมพันธ์หรือความเชื่อมโยงของสินค้าที่ถูกซื้อร่วมกันบ่อยนี้ เรียกว่า การทำเหมืองข้อมูลเพื่อค้นหาความสัมพันธ์ (association mining) ซึ่งอาศัยขั้นตอนพื้นฐานที่สำคัญคือ การค้นหารูปแบบที่ปรากฏบ่อย ลักษณะของรูปแบบที่ปรากฏบ่อย (frequent pattern) มีได้หลากหลายประเภท เช่น รูปแบบการซื้อสินค้าของลูกค้าส่วนใหญ่ในห้างสรรพสินค้า, รูปแบบการค้นหาข้อมูลจากเว็บเพจต่างๆ ของผู้ใช้อินเทอร์เน็ต, รูปแบบการจัดโครงสร้างโมเลกุลในสารประกอบโปรตีน เป็นต้น

นับตั้งแต่ปีค.ศ.1993 ที่ได้มีการเริ่มเสนอแนวคิดและเทคนิคของการทำเหมืองข้อมูลประเภทการค้นพบรูปแบบที่ปรากฏบ่อยโดย Rakesh Agrawal, Tomasz Imielinski และ Arun Swami [2] งานวิจัยด้านนี้ก็ได้รับความนิยมอย่างกว้างขวางเนื่องจากใช้ประโยชน์ได้กับการวิเคราะห์ข้อมูลในหลากหลายสาขา มีการพัฒนาเทคนิคต่างๆของการค้นหารูปแบบที่ปรากฏบ่อยให้มีประสิทธิภาพสูง แต่จากความก้าวหน้าของเทคโนโลยีอินเทอร์เนต ทำให้รูปแบบของข้อมูลมีลักษณะเปลี่ยนแปลงไปจากข้อมูลที่มีขนาดใหญ่แต่คงตัว (static) กลายเป็นข้อมูลที่มีลักษณะพลวัต (dynamic) เปลี่ยนแปลงได้ทั้งการเพิ่มปริมาณอย่างต่อเนื่องและการเปลี่ยนรูปแบบการกระจายของข้อมูล ข้อมูลที่มีปริมาณเพิ่มได้ไม่จำกัดเช่นนี้เรียกว่า ข้อมูลสตรีม ในช่วงเวลาห้าปีที่ผ่านมานักวิจัยจำนวนหนึ่งในสาขาการทำเหมืองข้อมูล เริ่มให้ความสนใจกับลักษณะของข้อมูลสตรีม และพยายามปรับปรุงเทคนิคการค้นหารูปแบบที่ปรากฏบ่อยที่ใช้งานได้ดีกับข้อมูลคงตัวให้ทำงานกับข้อมูลสตรีมได้ แต่เทคนิคเหล่านั้นยังมีข้อจำกัดและใช้ได้กับเพียงบางแอพพลิเคชัน ผู้วิจัยจึงได้เสนอแนวทางการโปรแกรมเชิงฟังก์ชันด้วยภาษาฮาสเกล (Haskell) และการโปรแกรมเชิงตรรกะด้วยภาษาโปรล็อก (Prolog) เพื่อพัฒนาเทคนิคการค้นหารูปแบบที่ปรากฏบ่อยให้ทำงานได้กับหลากหลายแอพพลิเคชัน

การโปรแกรมเชิงฟังก์ชันด้วยภาษา Haskell และการโปรแกรมเชิงตรรกะด้วยภาษา Prolog มีลักษณะเด่นคือโครงสร้างของภาษาสนับสนุนการทำงานกับแพทเทิร์นในลักษณะของ pattern matching นอกจากนี้ภาษาฮาสเกลยังใช้การประมวลผลด้วยเทคนิค lazy evaluation ที่ช่วยให้สามารถทำงานกับข้อมูลที่ปริมาณเพิ่มขึ้นอย่างไม่มีขีดจำกัด การพัฒนาเทคนิคการค้นพบรูปแบบที่ปรากฏบ่อยในข้อมูลสตรีมโดยใช้วิธีการโปรแกรมเชิงฟังก์ชันและเชิงตรรกะ ซึ่งทั้งสองวิธีการเป็นการโปรแกรมเชิงประกาศจะช่วยพัฒนาความก้าวหน้าของงานวิจัยในสาขาการทำเหมืองข้อมูลประเภทการค้นพบรูปแบบที่ปรากฏบ่อย และการค้นพบความสัมพันธ์

การค้นพบรูปแบบที่ปรากฏบ่อย

การทำเหมืองจากฐานข้อมูลทรานแซคชันขนาดใหญ่ เพื่อค้นหาความสัมพันธ์หรือความเกี่ยวข้องของสินค้าที่ถูกซื้อร่วมกันบ่อย ได้รับการเสนอแนวคิดในปี 1993 โดยทีมนักวิจัยของ IBM Almaden Research Center ที่ประกอบด้วย Rakesh Agrawal, Tomasz Imielinski และ Arun Swami [2] โดยได้เสนออัลกอริทึมที่ต่อมาภายหลังนิยมเรียกว่า อัลกอริทึม AIS เพื่อค้นหาความสัมพันธ์และความเชื่อมโยงของสินค้าแล้วแสดงในลักษณะของกฎ ซึ่งก็คือข้อความในรูปแบบถ้า..แล้ว เรียกข้อความหรือกฎนี้ว่า *กฎความสัมพันธ์* (association rule) และมักจะเขียนอยู่ในรูปแบบ $X \rightarrow Y$ (s, c) แทนความหมายว่าเมื่อลูกค้าซื้อสินค้า X แล้วลูกค้าจะซื้อสินค้า Y ร่วมด้วย โดย X และ Y คือเซตของสินค้า หรือไอเท็มเซต ที่ไม่ใช่เซตว่างและเป็นเซตที่สมาชิกไม่ซ้ำกัน สัญลักษณ์ s แทนค่า support หรือค่าสนับสนุน ซึ่งหมายถึงสัดส่วนของจำนวนเรคคอร์ดที่ปรากฏทั้ง X และ Y ต่อจำนวนเรคคอร์ดทั้งหมดในฐานข้อมูลทรานแซคชัน ส่วนสัญลักษณ์ c แทนค่า confidence หรือค่าความเชื่อมั่นของกฎที่คำนวณได้จากสัดส่วนของจำนวนเรคคอร์ดที่ปรากฏทั้ง X และ Y ต่อจำนวนเรคคอร์ดที่ปรากฏ X เมื่อมีการกำหนด support และ confidence เป็นเกณฑ์ หรือมาตรวัด วิธีการค้นหากฎความสัมพันธ์จึงประกอบด้วยสองขั้นตอนหลัก คือ

1. ค้นหาไอเท็มเซตที่ปรากฏบ่อยทั้งหมด โดยไอเท็มเซตที่จัดเป็นเซตที่ปรากฏบ่อยจะต้องมีค่าสนับสนุนเท่ากับหรือสูงกว่าค่าสนับสนุนที่กำหนดไว้เป็นเกณฑ์ขั้นต่ำ (เรียกเกณฑ์ขั้นต่ำนี้ว่า minimum support หรือ min_sup)
2. สร้างกฎความสัมพันธ์จากไอเท็มเซตที่ปรากฏบ่อย โดยกฎนี้จะต้องมีค่าความเชื่อมั่นเท่ากับหรือสูงกว่าค่าความเชื่อมั่นที่กำหนดไว้เป็นเกณฑ์ขั้นต่ำ (เรียกว่า minimum confidence หรือ min_conf)

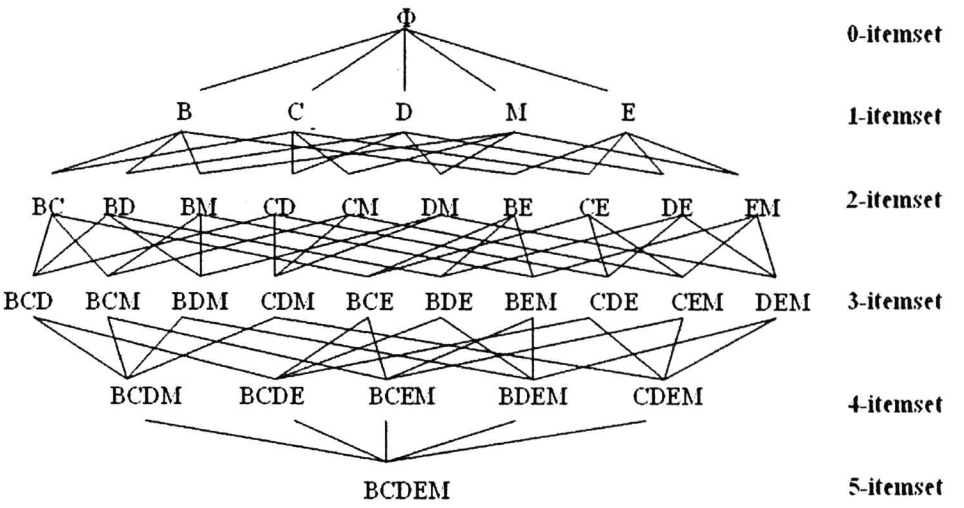
ตัวอย่างเช่น ถ้าฐานข้อมูลทรานแซคชันประกอบด้วย ข้อมูลการซื้อสินค้าของลูกค้า 5 ราย (รายละเอียดดังรูปที่ 1) การซื้อสินค้าของลูกค้าแต่ละรายจะเป็นแต่ละทรานแซคชันที่ระบุด้วยหมายเลขทรานแซคชัน หรือ TID (transaction identifier)

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

รูปที่ 1 ข้อมูลตัวอย่างแสดงรายการสินค้าที่ลูกค้าซื้อจากร้านค้า

ในฐานข้อมูลตัวอย่างมีสินค้า 5 ชนิดได้แก่ Beer, Cereal, Diaper, Egg, Milk สินค้าเหล่านี้เรียกว่าไอเท็ม กำหนดให้ในการค้นหาสินค้าที่ถูกซื้อร่วมกันบ่อย มีค่า min_sup เป็น 3/5 หรือ 60% และค่า min_conf เป็น 90%

ขั้นตอนแรกของการค้นหากฎความสัมพันธ์ จะเป็นการค้นหาไอเท็มเซตที่ปรากฏบ่อยทั้งหมด โดยจะต้องค้นหาจากไอเท็มเซตที่ไม่ใช่เซตว่างทั้งหมด 31 เซต แสดงไอเท็มเซตทั้งหมดได้ดังรูปที่ 2 (แต่ละไอเท็มในรูปใช้อักษรย่อ B, C, D, E, M แทน Beer, Cereal, Diaper, Egg, Milk ตามลำดับและละเว้นการใช้สัญลักษณ์เซตเพื่อให้อ่านง่าย)



รูปที่ 2 โครงข่ายของไอเท็มเซตทั้งหมดที่ต้องพิจารณาเพื่อค้นหาไอเท็มเซตที่ปรากฏบ่อย

เมื่อนับค่า support ของแต่ละไอเท็มเซตทั้ง 31 เซตเพื่อคัดเลือกเฉพาะไอเท็มเซตที่ผ่านเกณฑ์ min_sup 60% จะประกอบด้วยไอเท็มเซต 6 เซต ต่อไปนี้

1-itemset: {Beer}, {Cereal}, {Diaper}, {Milk}

2-itemset: {Beer and Diaper}, {Diaper and Milk}

3-itemset: {}

4-itemset: {}

5-itemset: {}

ขั้นตอนที่สองของการสร้างกฎความสัมพันธ์ จะเป็นการนำข้อมูลในไอเท็มเซตที่ปรากฏบ่อยมาสร้างเป็นกฎ โดยเริ่มจาก 2-itemset, 3-itemset, ... ไปตามลำดับ แต่เนื่องจากในตัวอย่างนี้ไอเท็มเซตที่ปรากฏบ่อยตั้งแต่ 3-itemset เป็นต้นไปเป็นเซตว่าง จึงพิจารณาเฉพาะข้อมูลใน 2-itemset ที่ประกอบด้วยสองเซตคือ {Beer and Diaper} และ {Diaper and Milk} นำไอเท็มใน 2-itemset มาสร้างกฎความสัมพันธ์ในเบื้องต้นได้ 4 กฎ ดังนี้

Beer $\longrightarrow$ Diaper (confidence = 3/3 = 100%)

Diaper $\longrightarrow$ Beer (confidence = 3/4 = 75%)

Diaper $\longrightarrow$ Milk (confidence = 3/4 = 75%)

Milk $\longrightarrow$ Diaper (confidence = 3/4 = 75%)

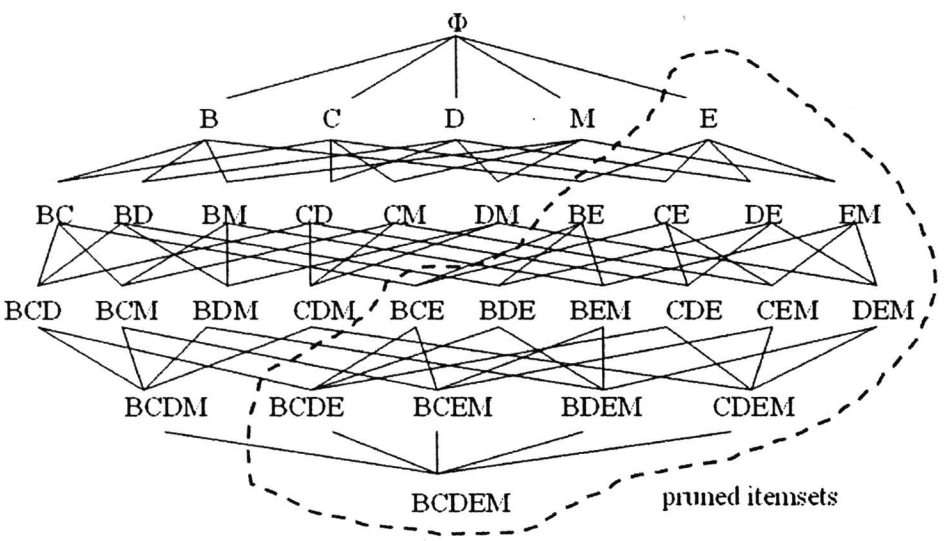
แต่จากเกณฑ์ min_conf ที่กำหนดไว้ 90% จึงทำให้ในผลลัพธ์สุดท้ายได้กฎความสัมพันธ์เพียงข้อเดียวคือ

Beer $\longrightarrow$ Diaper (confidence = 3/3 = 100%)

กฎความสัมพันธ์นี้ระบุด้วยความเชื่อมั่น 100% ว่าลูกค้าของร้านค้านี้จำนวนมากถึง 60% ที่ตั้งใจมาซื้อ Beer แล้วจะซื้อ Diaper ด้วย

จากจุดเริ่มต้นของการเสนออัลกอริทึม AIS [2] ในปีค.ศ. 1993 ทำให้แนวคิดของการค้นหาความรู้จากฐานข้อมูลหรือการทำเหมืองข้อมูล ประเภทการค้นหาความสัมพันธ์ ได้รับความสนใจอย่างมากจากนักวิจัยในสาขาการทำเหมืองข้อมูลและการวิเคราะห์ข้อมูล

ในปีต่อมา Rakesh Agrawal และ Ramakrishnan Srikant [3], [4] ได้ปรับปรุงอัลกอริทึม AIS ให้ทำงานได้เร็วขึ้นโดยอาศัยคุณสมบัติที่เรียกว่า Apriori property ที่ระบุว่า "เซตย่อยของไอเท็มเซตที่ปรากฏบ่อย จะต้องเป็นเซตที่ปรากฏบ่อยด้วยเช่นกัน" และเรียกอัลกอริทึมที่พัฒนาขึ้นใหม่นี้ว่า อัลกอริทึม APRIORI จุดเด่นของอัลกอริทึมนี้อยู่ที่ความสามารถในการพัฒนาความเร็วในการค้นหาไอเท็มเซตที่ปรากฏบ่อย ด้วยการละเว้นการพิจารณาไอเท็มเซตที่ปรากฏซ้ำด้วยความถี่ต่ำกว่าเกณฑ์ min_sup ดังตัวอย่างข้อมูลทรานแซคชันในรูปที่ 1 ถ้ากำหนด min_sup = 2/5 หรือ 40% จะพบว่าไอเท็ม Egg (หรือ E) ปรากฏในทรานแซคชันที่ 2 เพียงทรานแซคชันเดียว จึงไม่จัดเป็นไอเท็มเซตที่ปรากฏบ่อย ด้วยคุณสมบัติ Apriori ทำให้เราสามารถตัดการพิจารณาไอเท็มเซตทุกเซตที่มี E เป็นสมาชิก จึงลดจำนวนไอเท็มเซตที่ต้องพิจารณาจาก 31 เซต (ดังรูปที่ 2) ลงเหลือเพียง 15 เซต ดังแสดงในรูปที่ 3 (ส่วนที่ล้อมรอบด้วยเส้นประ คือไอเท็มเซตที่ไม่ต้องพิจารณาเนื่องจากปรากฏบ่อยไม่ถึงเกณฑ์ min_sup)



รูปที่ 3 โครงข่ายของไอเท็มเซตที่ขนาดเล็กลงจากการไม่ต้องพิจารณาไอเท็ม E

นับจากความสำเร็จของการคิดค้นอัลกอริทึม APRIORI [3],[4] ได้มีนักวิจัยจำนวนมากใช้แนวคิดนี้เป็นพื้นฐานและปรับปรุงประสิทธิภาพให้ดีขึ้นด้วยวิธีต่างๆ กัน งานวิจัยเด่นในกลุ่มนี้ประกอบด้วยงานวิจัยในปี 1995 ของ Park, Chen และ Yu [28] ที่เสนอแนวทางของการใช้เทคนิคแฮชชิงเพื่อเพิ่มความเร็วของการค้นหาไอเท็มเซตที่ปรากฏบ่อย Han และ Fu [19] ได้เสนอให้สร้างกฎความสัมพันธ์ที่มีโครงสร้างหลายระดับชั้นเพื่อให้มีความละเอียดมากขึ้น เช่น จากค้นพบกฎความสัมพันธ์ Beer → Diaper สามารถสืบค้นให้ละเอียดขึ้นได้ว่าเบียร์ที่ลูกค้านิยมซื้อร่วมกับผ้าอ้อมเด็กนั้นเป็นเบียร์ประเภทใด

ในกรณีของการค้นหาความสัมพันธ์จากฐานข้อมูลทรานแซคชัน ที่มีขนาดใหญ่มากจนกระทั่งไม่สามารถบรรจุข้อมูลทั้งหมดในหน่วยความจำหลักได้ Savasere, Omiecinski และ Navathe [31] เสนอให้ใช้เทคนิคการแบ่งข้อมูลเป็นส่วนย่อยแล้วทยอยค้นหาความสัมพันธ์ที่ปรากฏในแต่ละส่วนย่อยนั้น Toivonen [33] ได้ใช้แนวทางที่ต่างออกไปโดยเสนอการใช้เทคนิคการสุ่มเพื่อค้นหาความสัมพันธ์จากข้อมูลตัวแทน Cheung และคณะ [11] ได้เสนอให้ใช้เทคนิค incremental หรือการทำงานกับข้อมูลครั้งละไม่มาก และเมื่ออ่านข้อมูลใหม่เพิ่มเติมจะต้องสามารถปรับกฎความสัมพันธ์ให้ถูกต้องสอดคล้องกับข้อมูลใหม่ได้ นอกจากนี้ยังมีนักวิจัยจำนวนมาก ได้แก่ Park, Chen และ Yu [29], Agrawal และ Shafer [5], Cheung และคณะ [10], Zaki และคณะ [35] ได้เสนอแนวทางการพัฒนาความเร็วในการค้นหาไอเท็มเซตที่ปรากฏบ่อย ด้วยการประมวลผลแบบขนาน

งานวิจัยที่กล่าวถึงข้างต้นล้วนแต่ใช้แนวทาง APRIORI แต่เสริมประสิทธิภาพความเร็วด้วยเทคนิคต่างๆกัน แต่งานวิจัยของ Jiawei Han, Jian Pei และ Yiwen Yin ในปี 2000 [20] ได้เสนอแนวทางที่แตกต่างออกไป ด้วยการอ่านข้อมูลแล้วสร้างโครงสร้างต้นไม้เรียกว่า frequent pattern tree หรือ FP-tree จากนั้นใช้อัลกอริทึม FP-growth เพื่อค้นหาไอเท็มเซตที่ปรากฏบ่อย โดยไม่ต้องสร้างไอเท็มเซตชั่วคราว แล้วตัดทิ้งภายหลังเมื่อค่าสนับสนุนของเซตต่ำกว่าเกณฑ์ค่าสนับสนุนขั้นต่ำ วิธีนี้เมื่อเปรียบเทียบกับวิธี ARIORI แล้วสามารถเพิ่มความเร็วในการค้นหาไอเท็มเซตที่ปรากฏบ่อย แนวทางการค้นหาไอท็มเซตที่ปรากฏบ่อยด้วยโครงสร้าง FP-tree ได้รับความสนใจและพัฒนาต่อเนื่องมาโดยลำดับดังปรากฏในงานวิจัยต่างๆในช่วงปี 2000 ถึงปัจจุบัน [1], [16], [26], [30]

ในช่วงระยะเวลาตั้งแต่ปี 1993 ถึง 2000 เทคนิคการค้นพบกฎความสัมพันธ์และการค้นหารูปแบบที่ปรากฏบ่อยได้รับการพัฒนาอย่างต่อเนื่องให้มีประสิทธิภาพสูง และสามารถรองรับข้อมูลขนาดใหญ่ได้ แต่เมื่อเทคโนโลยีอินเทอร์เน็ตได้รับความนิยมสูงขึ้น ลักษณะของข้อมูลเปลี่ยนจาก offline เป็น online และปริมาณของข้อมูลเพิ่มขึ้นอย่างไม่มีขีดจำกัดเกิดเป็นลักษณะข้อมูลสตรีม หรือ data stream ข้อมูลสตรีมเริ่มได้รับการนิยาม [6], [14], [17], [21] เมื่อปี 2001 ว่าหมายถึง ข้อมูลที่เกิดขึ้นอย่างต่อเนื่อง ไม่มีขีดจำกัดในเรื่องของปริมาณและจุดสิ้นสุด ข้อมูลถูกส่งออกจากแหล่งผลิตด้วยความเร็วสูงและอาจจะมีการกระจายของข้อมูลที่ไม่คงที่ นักวิจัยได้พยายามปรับปรุงเทคนิคการค้นพบกฎความสัมพันธ์ให้ทำงานได้กับข้อมูลสตรีม เช่น ในปี 2005 Halatchev และ Gruenwald [18] ได้ปรับปรุงเทคนิคการค้นพบกฎความสัมพันธ์ให้สามารถประเมินข้อมูลที่หายไปในข้อมูลสตรีมที่รับมาจากเซนเซอร์ที่ส่งผ่านเครือข่าย Kargupta และคณะ [22] ได้พัฒนาระบบ VEDAS เมื่อปี 2004 ให้สามารถตรวจจับยานพาหนะในเวลาจริง นอกจากนี้ยังมีงานวิจัยจำนวนมากในช่วงระยะเวลาปี 2004-2005 [7], [8], [9], [12], [13], [15], [24], [25], [27], [32], [34] ที่มุ่งพัฒนาเทคนิคในการค้นหารูปแบบความสัมพันธ์และไอเท็มเซตที่ปรากฏบ่อยในข้อมูลสตรีม อัลกอริทึมต่างๆที่ถูกเสนอนี้ บางอัลกอริทึมพัฒนาขึ้นเพื่อรองรับเฉพาะบางแอพพลิเคชัน บางอัลกอริทึมใช้ค้นหาความสัมพันธ์เฉพาะข้อมูลในบางช่วง เช่น เฉพาะข้อมูลที่เกิดขึ้นล่าสุดในสตรีม และบางอัลกอริทึมทำงานกับข้อมูลสตรีมในลักษณะ offline

ทีมผู้วิจัยต้องการพัฒนาเทคนิคการค้นหารูปแบบที่ปรากฏบ่อยในข้อมูลสตรีมในลักษณะ online ให้ทำงานกับข้อมูลสตรีมได้หลากหลายแอพพลิเคชัน และเพื่อให้ได้รูปแบบความสัมพันธ์ในเวลาที่รวดเร็วทันต่อความต้องการใช้งาน การค้นหาความสัมพันธ์และรูปแบบที่ปรากฏบ่อยจะมุ่งเน้นที่การค้นพบโดยประมาณ การประมาณ

ค่าความถี่ของการปรากฏรูปแบบจะปรับปรุงจากเทคนิค Monte Carlo approximation ที่ผู้วิจัยและคณะได้นำเสนอไว้ใน [23] ในการพัฒนาโปรแกรมต้นแบบจะใช้แนวทางการโปรแกรมเชิงฟังก์ชันด้วยภาษา Haskell เนื่องจากมีความเหมาะสมในการทำ pattern matching และมีรูปแบบการประมวลผลในแบบ lazy evaluation จึงสามารถรองรับข้อมูลสตรีมที่เป็น infinite data ได้ดี นอกจากนี้ยังได้ทดสอบแนวทางการโปรแกรมเชิงตรรกะเพื่อใช้เปรียบเทียบประสิทธิภาพของการทำโปรแกรมระดับสูง

การโปรแกรมเชิงประกาศ

ลักษณะการโปรแกรมเชิงประกาศจะแตกต่างจากการโปรแกรมเชิงกระบวนคำสั่ง ตรงที่รูปแบบคำสั่งจะสั้นกว่าและใช้วิธีการประกาศแพทเทิร์นของอินพุทและเอาท์พุท โดยไม่ต้องระบุรายละเอียดขั้นตอน ข้อแตกต่างนี้จะเห็นได้ชัดเจน ดังตัวอย่างต่อไปนี้ที่แสดงการเปรียบเทียบวิธีการเขียนโปรแกรมเรียงลำดับข้อมูลแบบ quick sort ด้วยวิธีการโปรแกรมเชิงประกาศ (ด้วยภาษาฮาสเกล และ โปรล็อก) เทียบกับวิธีการโปรแกรมเชิงกระบวนคำสั่ง (ด้วยภาษาซี)

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, cc, 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;     }
```

การโปรแกรมเชิงประกาศจะใช้เทคนิค pattern matching เพื่อระบุรูปแบบอินพุทและเอาท์พุท และใช้วิธีการเรียกตัวเองซ้ำในลักษณะของ recursive ดังตัวอย่างในภาษาฮาสเกลและภาษาโปรล็อกต่อไปนี้

```
-- .....
-- Fibonacci function in Haskell
-- .....
   fib :: Int -> Int          -- a function takes one Integer and returns an Integer
   fib 0 = 0                  -- pattern 1: input is 0, output = 0
   fib 1 = 1                  -- pattern 2: input is 1, output = 1
   fib n = fib (n-2) + fib (n-1)   -- pattern 3: input is an integer other than 0 and 1,
                              --          then output = fib(n-2) + fib(n-1)
% .....
% Fibonacci function in Prolog
% .....
fib(0, 0).                 % pattern 1: input is 0, output is 0.
fib(1, 1).                 % pattern 2: input is 1, output is 1.
fib(N, F) :- N > 1,        % pattern 3: input is N and N>1,
             N1 is N-1,                % then declare N1 = N-1
             N2 is N-2,                %          N2 = N-2
             fib(N1, F1), fib(N2, F2), %  recursive calls fib() to get F1 and F2
             F is F1 + F2.             %  compute result F = F1+F2
```

การโปรแกรมเชิงประกาศเพื่อการค้นพบรูปแบบที่ปรากฏบ่อย

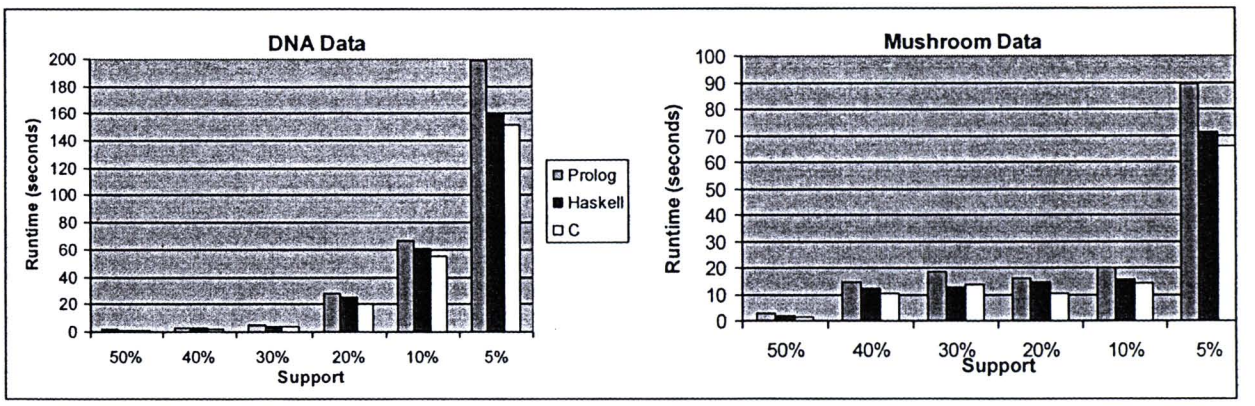
การค้นพบรูปแบบที่ปรากฏบ่อยด้วยวิธีการโปรแกรมเชิงฟังก์ชันในภาษาฮาสเกล ซอร์สโค้ดมีความยาวประมาณ 37 บรรทัด แสดงบางส่วนของโปรแกรมได้ดังรูปที่ 4 ในขณะที่การเขียนโปรแกรมเชิงตรรกะด้วยภาษาโปรล็อก ซอร์สโค้ดมีความยาวประมาณ 58 บรรทัด และแสดงบางส่วนของโปรแกรมได้ดังรูปที่ 5 อัลกอริทึมเดียวกันนี้เมื่อเขียนโปรแกรมด้วยภาษาซี ซอร์สโค้ดมีความยาวมากกว่า 300 บรรทัด

```
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 is a function to find candidate itemset
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 is a function to find large (or frequent) itemset
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]
```

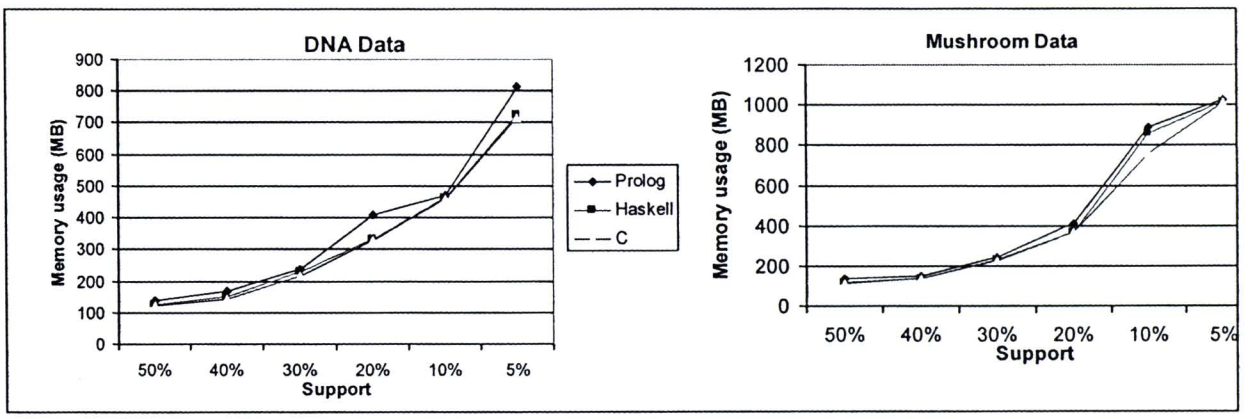
รูปที่ 4 โปรแกรมภาษาฮาสเกลเพื่อการค้นพบรูปแบบที่ปรากฏบ่อย

```
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).
```

รูปที่ 5 โปรแกรมโปรล็อกเพื่อการค้นพบรูปแบบที่ปรากฏบ่อย



รูปที่ 6 การเปรียบเทียบความเร็วของโปรแกรมด้วยข้อมูล DNA (ภาพซ้าย) และ Mushroom (ภาพขวา)



รูปที่ 7 การเปรียบเทียบหน่วยความจำที่ใช้เมื่อรันด้วยข้อมูล DNA (ภาพซ้าย) และ Mushroom (ภาพขวา)

การทดสอบความเร็วของการประมวลผล (รูปที่ 6) และหน่วยความจำที่ใช้ในระหว่างการประมวลผล (รูปที่ 7) ของโปรแกรมค้นพบรูปแบบที่ปรากฏบ่อยที่เขียนด้วยภาษาโปรล็อก ฮาสเกล และ ซี ทดสอบด้วยเครื่องคอมพิวเตอร์โน้ตบุค ความเร็วซีพียู 796 MHz หน่วยความจำหลัก 512 MB ฮาร์ดดิสก์ความจุ 40 GB ข้อมูล DNA ที่ใช้ในการทดสอบมีขนาด 252 KB จำนวนทรานแซคชัน (หรือเรคคอร์ด) 2,000 ทรานแซคชัน และมีจำนวนไอเท็ม (หรือฟิลด์) 61 ไอเท็ม ข้อมูล Mushroom มีขนาด 916 KB จำนวนทรานแซคชัน 5,416 ทรานแซคชัน และมีจำนวนไอเท็ม 23 ไอเท็ม

การค้นพบรูปแบบที่ปรากฏบ่อยจะต้องมีการกำหนดค่าสนับสนุน (support) ถ้ากำหนดค่าสนับสนุนต่ำจะมีผลให้ต้องค้นหารูปแบบในข้อมูลปริมาณมาก จากผลการทดสอบจึงเห็นได้ว่าเมื่อลดค่าสนับสนุนจาก 10% เป็น 5% โปรแกรมจะใช้เวลาประมวลผลนานมากขึ้น และใช้หน่วยความจำมาก และที่ค่าสนับสนุน 5% โปรแกรมที่เขียนด้วยภาษาโปรล็อกจะใช้เวลาในการประมวลผลนานกว่าโปรแกรมที่เขียนด้วยภาษาฮาสเกล และภาษาซี เมื่อเปรียบเทียบหน่วยความจำที่ใช้ในระหว่างการประมวลผล จะเห็นได้ว่าโปรแกรมในทั้งสามภาษาใช้เนื้อที่หน่วยความจำไม่ต่างจากกันอย่างมีนัยสำคัญ ดังนั้นเมื่อนำปัจจัยด้านขนาดของซอร์สโค้ดมาพิจารณาร่วมด้วย จะสามารถสรุปผลการทดสอบเบื้องต้นนี้ได้ว่าภาษาฮาสเกลซึ่งเป็นภาษาเชิงประกาศ ที่ใช้ลักษณะของฟังก์ชันในทางคณิตศาสตร์เป็นแนวคิด มีความเหมาะสมในการพัฒนาโปรแกรมค้นพบรูปแบบที่ปรากฏบ่อย โดยเฉพาะเมื่อระบุค่าสนับสนุนต่ำซึ่งต้องใช้หน่วยความจำในระหว่างการประมวลผลสูง

สรุป

งานวิจัยที่ได้นำเสนอนี้เป็นส่วนหนึ่งของโครงการวิจัยการค้นพบรูปแบบที่ปรากฏบ่อยในข้อมูลสตรีม ซึ่งเป็นข้อมูลที่มีขนาดไม่จำกัดและเกิดขึ้นอย่างต่อเนื่อง การพัฒนาโปรแกรมเพื่อประมวลผลกับข้อมูลประเภทนี้ จึงต้องเลือกใช้ภาษาการโปรแกรมที่เหมาะสม ผู้วิจัยได้ทดสอบภาษาในสามกลุ่มคือภาษาเชิงกระบวนคำสั่ง โดยใช้ภาษาซีเป็นกรณีศึกษา ภาษาเชิงฟังก์ชันโดยได้เลือกใช้ภาษาฮาสเกล และภาษาเชิงตรรกะด้วยภาษาโปรล็อก โดยสองภาษาหลังนี้รวมอยู่ในกลุ่มของภาษาที่ใช้ในการโปรแกรมเชิงประกาศ เนื่องจากวิธีการเขียนโปรแกรมใช้การประกาศรูปแบบอินพุทและเอาท์พุทโดยไม่ต้องเน้นรายละเอียดขั้นตอนการโปรแกรม จากผลการทดสอบในสามประเด็นหลักคือความยาวของซอร์สโค้ด ความเร็วในการประมวลผล และหน่วยความจำที่ใช้ในระหว่างการประมวลผล พบว่าภาษาฮาสเกลมีความเหมาะสมที่จะนำมาใช้ในการเขียนโปรแกรมประเภทการค้นพบรูปแบบที่ปรากฏบ่อย ดังนั้นการพัฒนางานวิจัยในอนาคตผู้วิจัยจะเพิ่มเติมเทคนิคการค้นพบโดยประมาณ และทดสอบการทำงานกับข้อมูลสตรีมในโปรแกรมการค้นพบรูปแบบที่ปรากฏบ่อยที่เขียนด้วยภาษาฮาสเกล

กิตติกรรมประกาศ

งานวิจัยนี้ได้รับการสนับสนุนงบประมาณจากหลายหน่วยงานได้แก่สำนักงานคณะกรรมการวิจัยแห่งชาติ (วช.) สำนักงานคณะกรรมการการอุดมศึกษาและสำนักงานกองทุนสนับสนุนการวิจัย (สกอ. และ สกว., รหัสโครงการ RMU-5080026) หน่วยวิจัยด้านวิศวกรรมข้อมูลและการค้นหาความรู้ เป็นหน่วยปฏิบัติการวิจัยที่ได้รับการสนับสนุนการดำเนินงานทั้งด้านสถานที่ เครื่องมือและงบประมาณจากมหาวิทยาลัยเทคโนโลยีสุรนารี

เอกสารอ้างอิง

[1] R. Agrawal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61:350-371, 2001.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'93)*, pages 207-216, Washington, DC, May 1993.

[3] R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. In *Research Report RJ 9839*, IBM Almaden Research Center, San Jose, CA, June 1994.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487-499, Santiago, Chile, Sept. 1994.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Engineering*, 8:962-969, 1996.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Model and issues in data stream systems. In *Proc. ACM Symp. Principles of Database Systems (PODS'02)*, 2002.

[7] Y. Cai, G.Pape, J. Han, M. Welge, and L. Auvil. MAIDS: Mining alarming incidents from data streams. In *Proc. Int. Conf. on Management of Data*, June 2004.

[8] J. Chang and W. Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering*; July 2004.

[9] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, Jan. 2004.

[10] D. W. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proc. 1996 Int. Conf. Parallel and Distributed Information Systems*, pages 31-44, Miami Beach, Florida, Dec. 1996.

[11] D. W. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 1996 Int. Conf. Data Engineering (ICDE'96)*, pages 106-114, New Orleans, Louisiana, Feb. 1996.

[12] Y. Chi, H. Wang, P. Yu, and R. R. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proc. IEEE Int. Conf. on Data Mining*, Nov. 2004.

[13] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Resource-aware knowledge discovery in data streams. In *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, Sept. 2004.

[14] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2), June 2005.

[15] A. Ghoting and S. Parthasarathy. Facilitating interactive distributed data stream processing and mining. In *Proc. IEEE Int. Symposium on Parallel and Distributed Processing Systems*, Apr. 2004.

[16] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Proc. *ICDM'03 Int. Workshop on Freuquent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[17] S. Guha, N. Koudas, and K. Shim. Data streams and histograms. In *Proc. ACM Symposium on Theory of Computing*, 2001.

[18] M. Halatchev and L. Gruenwald. Estimating missing values in related sensor data streams. In *Proc. Int. Conf. on Management of Data*, Jan. 2005.

[19] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 420-431, Zurich, Switzerland, Sept. 1995.

[20] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1-12, Dallas, TX, May 2000.

[21] M. Jiang and L. Gruenwald. Research issues in data stream association mining. *ACM SIGMOD Record*, 35(1): 14-19, 2006.

[22] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy. VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring. In *Proc. SIAM Int. Conf. on Data Mining*, 2004.

[23] K. Kerdprasop, N. Kerdprasop, and P. Sattayatham. A Monte Carlo method to data stream analysis. *Enformatika Transactions on Engineering, Computing and Technology*, 14: 240-245, Aug. 2006.

[24] H.-F. Li, S.-Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. Int. Workshop on Knowledge Discovery in Data Streams*, Sept. 2004.

[25] C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, and A. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proc. SIAM Int. Conf. on Data Mining*, April 2005.

[26] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 239-248, Edmonton, Canada, July 2002.

[27] G. Mao, X. Wu, C. Liu, X. Zhu, G. Chen, Y. Sun, and X. Liu. Online mining of maximal frequent item sequences from data streams. *Technical Report CS-05-07*, University of Vermont, Computer Science, June 2005.

[28] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'95)*, pages 175-186, San Jose, CA, May 1995.

[29] J. S. Park, M. S. Chen, and P. S. Yu. Efficient parallel mining for association rules. In *Proc. 4th Int. Conf. Information and Knowledge Management*, pages 31-36, Baltimore, Maryland, Nov. 1995.

[30] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215-224, Heidelberg, Germany, April 2001.

[31] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95), pages 432-443, Zurich, Switzerland, Sept. 1995.

[32] W.-G. Teng, M.-S. Chen, and P. Yu. Resource-aware mining with variable granularities in data streams. In *Proc. SIAM Int. Conf. on Data Mining*, 2004.

[33] H. Toivonen. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 134-145, Bombay, India, Sept. 1996.

[34] J. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[35] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithm for discovery of association rules. *Data Mining and Knowledge Discovery*, 1:343-374, 1997.