Preface

Ever since the early days of machine learning and data mining, it has been realized that the traditional attribute-value and item-set representations are too limited for many practically applications in domains such as chemistry, biology, network analysis and text mining. This has triggered a lot of research on mining and learning within alternative and more expressive representation formalisms such as computational logic, relational algebra, graphs, trees and sequences. The state-of-the-art is that attribute-value and item-set representations lie at one extreme end of the spectrum, and multi-relational data mining and inductive logic programming at the other end. The middle is occupied by traditional data structures employed throughout the field of computer science. These include graphs, trees and sequences (or strings). The motivation for using such representations is that they are 1) more expressive (and therefore more widely applicable) than flat representations, and 2) potentially more efficient than multi-relational learning and mining techniques. At the same time, the data structures of graphs, trees and sequences are among the best understood and most widely applied representations within computer science. Thus these representations offer ideal opportunities for developing interesting contributions in data mining and machine learning that are both theoretically well-founded and widely applicable.

Whereas there have been a large number of workshops devoted to multi-relational data mining and inductive logic programming as well as applications of intermediate representations in e.g. ontologies, bioinformatics, XML-data, text-mining, there has - to the best of our knowledge - not been any workshops specifically devoted to foundational issues in intermediate representations. It is precisely the goal of this workshop to bring together researchers interested in mining and learning within graphs, trees and sequences. Considerable number of papers have been submitted to this workshop, and the papers selected for presentation have high quality in terms of originality, theory and potential practicality. We believe that this workshop provides a great chance to exchange stimulative information among researchers and establish a new research field of data mining.

Program Committee Co-chairs
Luc De Raedt and Takashi Washio

Table of Contents

Invited Tutorial

An Introduction to Kernel Methods for Mining Graph Data
Thomas Gartner
Fraundhofer Institut Autonome Intelligence Systeme and University of Bonn

Papers **Page**

1. Constructing a Decision Tree for Graph Structured Data 1
Warodom Geamsakul, Takashi Matsuda, Tetsuya Yoshida,
Hiroshi Motoda and Takashi Washio
Osaka University

2. Probabilistic Approach for Reduction of Irrelevant Tree-structured Data 11
Amaury Habrard, Marc Bernard and Marc Sebban
Universite Jean Monnet de Saint-Etienne

3. Effective Rule Induction from Molecular Structures Represented by Labeled Graphs 21
Susanne Hoche, Tamas Horvath and Stefan Wrobel
Fraundhofer Institut Autonome Intelligence Systeme and University of Bonn

4. Learning Logic Programs 33
with Unary Partial Function Graph Background Knowledge
Tamas Horvath, Robert H. Sloan and Gyorgy Turan
University of Bonn, Fraundhofer Institut Autonome Intelligence Systeme,
University of Illinois at Chicago and Hungarian Academy of Sciences

5. Specific Biasis for Mining Frequent Substructures 45
Akihiro Inokuchi, Takashi Washio and Hiroshi Motoda
IBM Japan and Osaka University

6. Effcient Discovery of Frequent Unordered Trees 55
Siegfried Nijssen and Joost N. Kok
Leiden Institute of Advanced Computer Science

7. Expressivity versus Efficiency of Grapgh Kernels 65
Jan Ramon and Thomas Gartner
K.U. Luven, Fraundhofer Institut Autonome Intelligence Systeme
and University of Bonn

8. Distance-based Clustering of XML Documents 75
Francesco De Francesca, Gianluca Gordano, Riccardo Ortale and Andrea Tagarelli
University of Calabria

9. Graph Mining Approaches for the Discovery of Web Communities 79
Tsuyoshi Murata
National Institute of Informatics and Japan Science and Technology Corporation

Constructing a Decision Tree for Graph Structured Data

Warodom Geamsakul, Takashi Matsuda, Tetsuya Yoshida,
Hiroshi Motoda and Takashi Washio

Institute of Scientific and Industrial Research,
Osaka University
8-1 Mihogaoka, Ibaraki, Osaka 567-0047, JAPAN
{warodom,matsuda,yoshida,motoda,washio}@ar.sanken.osaka-u.ac.jp

Abstract. Decision tree Graph-Based Induction (DT-GBI) is proposed that constructs a decision tree for graph structured data. Substructures (patterns) are extracted at each node of a decision tree by stepwise pair expansion (pairwise chunking) in GBI to be used as attributes for testing. Since attributes (features) are constructed while a classifier is being constructed, DT-GBI can be conceived as a method for feature construction. The predictive accuracy of a decision tree is affected by which attributes (patterns) are used and how they are constructed. A beam search is employed to extract good enough discriminative patterns within the greedy search framework. Pessimistic pruning is incorporated to avoid overfitting to the training data. Experiments using a DNA dataset were conducted to see the effect of the beam width, the number of chunking at each node of a decision tree, and the pruning. The results indicate that DT-GBI that does not use any prior domain knowledge can construct a decision tree that is comparable to other classifiers constructed using the domain knowledge.

1 Introduction

Since structure is represented by proper relations and a graph can easily represent relations, knowledge discovery from graph structured data poses a general problem for mining from structured data. Examples amenable to graph mining problems are finding typical web browsing patterns, identifying typical substructures of chemical compounds, finding typical subsequences of DNA and discovering diagnostic rules from patient history records.

Graph-Based Induction (GBI) [9, 3], on which DT-GBI is based, discovers typical patterns in general graph structured data by recursively chunking two adjoining nodes. It can handle a graph data having loops with colored/uncolored nodes and links. GBI is very efficient because it employs greedy search. GBI does not lose any information of graph structure after chunking, and it can use various evaluation functions in so far as they are based on frequency. It is not, however, suitable for graph structured data where many nodes share the same label because of its greedy recursive chunking without backtracking, but it is

still effective in extracting patterns from data where each node has a distinct label (*e.g.,* World Wide Web browsing data) or where some typical structures exist even if some nodes share the same labels (*e.g.,* chemical structure data containing benzene rings etc). The decision tree construction method [5, 6] is a widely used technique for data classification and prediction, but the data must be represented by or transformed into attribute-value pairs. However, it is not trivial to define proper attributes for graph-structured data beforehand.

We have proposed a method called Decision Tree Graph-Based Induction (DT-GBI), which constructs a classifier (decision tree) for graph-structured data while constructing the attributes during the course of tree building using GBI recursively and did preliminary performance evaluation [8].

A pair extracted by GBI, consisting of nodes and links among them[1] that is treated as an attribute and the existence/non-existence of the pair in a graph is treated as its value for the graph. Thus, attributes (pairs) that split the data effectively are extracted by GBI while a decision tree is being constructed. To classify unseen graph-structured data by the constructed decision tree, attributes that appear in the nodes of the tree are produced from data before the classification.

In this paper we first report an improvement made on DT-GBI after the preliminary analysis [8] to increase its predictive accuracy. After that, we report on the performance evaluation of the improved DT-GBI through experiments using a DNA dataset from the UCI repository and show that the results are comparable to the results that are obtained by using the domain knowledge [7].

Section 2 briefly describes the framework of DT-GBI and Section 3 describes the improvement made on DT-GBI. Evaluation of the improved DT-GBI is reported in Section 4. Section 5 concludes the paper with a summary of the results and the planned future work.

2 Decision Tree Graph-Based Induction

2.1 Graph-Based Induction Revisited

GBI employs the idea of extracting typical patterns by stepwise pair expansion. In the original GBI an assumption was made that typical patterns represent some concepts/substructure and "typicality" is characterized by the pattern's frequency or the value of some evaluation function of its frequency. Repeated chunking enables to extract typical patterns of various sizes. The search is greedy and no backtracking is made. Because of this, all the "typical patterns" that exist in the input graph are not necessarily extracted. The problem of subgraph isomorphism is known to be NP-complete. GBI aims at extracting only meaningful typical patterns of certain sizes. Its objective is not finding all the typical patterns nor finding all the frequent patterns.

For finding a pattern that is of interest any of its subpatterns must be of interest because of the nature of repeated chunking, *i.e.*, a larger pattern can only

[1] Repeated chunking of pairs results in subgraph structure

be constructed by pairing two smaller subpatterns, which must have been constructed at earlier steps. Frequency measure satisfies this monotonicity. However, if the criterion chosen does not satisfy this monotonicity, repeated chunking may not find good patterns even though the best pair based on the criterion is selected at each iteration. To resolve this issue GBI was improved to use two criteria, one for frequency measure for chunking and the other for finding discriminative patterns after chunking. The latter criterion does not necessarily hold monotonicity property. Any function that is discriminative can be used, such as Information Gain [5], Gain Ratio [6] and Gini Index [2], and some others.

The improved stepwise pair expansion algorithm is summarized in Fig. 1. The output of the improved GBI is a set of ranked typical patterns. These patterns are typical in the sense that they are more discriminative than non-selected patterns in terms of the criterion used.

```
GBI(G)
  Enumerate all the pairs P_all in G
  Select a subset P of pairs from P_all (all the pairs
    in G) based on typicality criterion
  Select a pair from P_all based on chunking criterion
  Chunk the selected pair into one node c
  G_c := contracted graph of G
  while termination condition not reached
    P := P ∪ GBI(G_c)
  return P
```

Fig. 1. Algorithm of GBI

2.2 Feature Construction by GBI

We regard a subgraph in a graph as an attribute so that graph-structured data can be represented with attribute-value pairs according to the existence of particular subgraphs.

However, it is difficult to identify and extract those subgraphs selectively which are effective for classification task beforehand. If pairs are extended in a step-wise fashion by GBI and discriminative ones are selected and further extended while constructing a decision tree, discriminative patterns (subgraphs) can be constructed simultaneously during the construction of a

```
DT-GBI(D)
  Create a node DT for D
  if termination condition reached
    return DT
  else
    P := GBI(D) (with the number of chunking
      specified)
    Select a pair p from P
    Divide D into D_y (with p) and D_n (without p)
    Chunk the pair p into one node c
    D_yc := contracted data of D_y
    for D_i := D_yc, D_n
      DT_i := DT-GBI(D_i)
      Augment DT by attaching DT_i as its child
        along yes(no) branch
  return DT
```

Fig. 2. Algorithm of DT-GBI

decision tree. The algorithm of DT-GBI is summarized in Fig. 2. Since the value for an attribute is yes (contains pair) and no (does not contain pair), the constructed decision tree is represented as a binary tree. Each time when an attribute (pair) is selected to split the data, the pair is chunked into a larger node in size. Thus, although initial pairs consist of two nodes and the link between them, attributes useful for classification task are gradually grown up into larger pair (subgraphs) by applying chunking recursively. In this sense the proposed DT-GBI method can be conceived as a method for feature construction.

3 Enhancement of DT-GBI

3.1 Beam Search for Expanding Search Space

Since the search in GBI is greedy and no backtracking is made, which patterns are extracted by GBI depends on which pair is selected for chunking in Fig. 2. To increase the search space and extract good enough patterns still keeping the computational complexity within a tolerant level, a beam search is incorporated to GBI within the framework of greedy search [4]. A certain fixed number of pairs ranked from the top are allowed to be chunked individually in parallel. To prevent each branch from growing exponentially, the total number of pairs to chunk, thus the beam width, is fixed at each level of branch. Thus, at any iteration step, there is always a fixed number of chunking that is performed in parallel.

Fig. 3. An Example of state transition with beam search when the beam width = 5

An example of state transition with beam search is shown in Fig.3 in the case where the beam width is 5. The search starts with a single state *cs*. All pairs in *cs* are enumerated and ranked according to both the frequency measure and the typicality measure (*e.g.*, discriminatory measure). The top 5 pairs according to the frequency measure are selected, and each of them is used as a pattern to chunk, branching into 5 children c_{11}, c_{12}, ..., c_{15}, each rewritten by the chunked pair. All pairs within these 5 states are enumerated and ranked according to the two measures, and again the top 5 ranked pairs according to the frequency measure are selected. The state c_{11} is split into two states c_{21} and c_{22} because two pairs are selected, but the state c_{12} is deleted because no pair is selected. This is repeated until the stopping condition is satisfied. Increase in the search space improves the pattern extraction capability of GBI and thus that of DT-GBI.

3.2 Pruning Decision Tree

Recursive partitioning of data until each subset in the partition contains data of a single class results in overfitting to the training data and thus degrades the predictive accuracy of decision trees. Our previous approach [8] used a very naive *prepruning* method by limiting the threshold number of graphs in leaves to 10. To improve the predictive accuracy, a pessimistic pruning used in C4.5 [6] is implemented by growing an overfitted tree first and then pruning it based on the confidence interval for binomial distribution. The current algorithm has a step for postpruning in Fig. 2.

4 Performance Evaluation of DT-GBI

The proposed method is tested against the promoter dataset in UCI Machine Learning Repository[1]. This dataset consists of strings that represent nucleotides (one of A, G, T, or C). The input features are 57 sequential DNA nucleotides and the total number of instances is 106 including 53 positive instances (sample promoter sequence) and 53 negative instances (non-promoter sequence). This dataset was explained and analyzed in [7]. In their analysis, they first configured a neural network using the domain knowledge and refined it to best fit the results. The rules were then extracted from the converged network. Thus, their method needs the domain knowledge to guide the search.

One important thing is that the data is so prepared that each sequence of nucleotides is aligned at a reference point, which makes it possible to assign the n-th attribute to the n-th nucleotide in the attribute-attribute value representation (thus applicable to a neural network). In a sense, this dataset is encoded using the domain knowledge. This is confirmed by the following experiment. Running C4.5 gives a predictive error of 16.0% by leaving one out cross validation. Randomly shifting the sequence by 3 elements gives 21.7% and by 5 elements 44.3% (Fig. 4). If the data is not properly aligned, standard classifiers such as C4.5 that use attribute-attribute value representation does not solve this problem. One of the advantages of graph representation is that it does not require the data to be aligned around a reference point. In this paper, each sequence is converted to a graph representation assuming that an element interacts up to 10 elements on both sides (See Fig. 5). Each sequence thus results in a graph with 57 nodes and 515 lines.

Fig. 4. Change of error rate by shifting the sequence in the promoter dataset

In the experiment, frequency (chunking measure) was used to select a pair to chunk in GBI and information gain [5] (typicality measure) was used in DT-GBI to select a pair from the pairs that are returned by GBI. A decision tree was constructed in either of the following two ways: 1) apply chunking n_r times only at the root node and only once at other nodes of a decision tree, and 2) apply chunking n_e times at every node of a decision tree. Note that n_r and n_e are defined along the depth in Fig. 3. Thus, there is more chunking taking place during the search when the beam width is larger. The pair (subgraph) that is selected for each node of the decision tree is the one which maximizes the informaion gain among all the pairs that are enumerated. Pruning of decision tree was conducted either by a) prepruning: set the termination condition in DT-GBI in Fig. 2 to whether the number of graphs in D is equal to or less than 10 or b) postpruning: conduct the pessimistic pruning in Subsection 3.2 by setting the confidence level to 25%. The beam width was changed from 1 to 15. The prediction error rate of a decision tree constructed by DT-GBI was evaluated by the average of 10 runs of 10 fold cross validation.

Fig. 5. Conversion of DNA Sequence Data to a graph

The first experiment focused on the effect of the number of chunking at each node of a decision tree and thus beam width was set to 1 and the prepruning was used. The parameters n_r and n_e were changed from 1 to 10 in accordance with 1) and 2) explained in the previous paragraph, respectively. Fig. 6 shows the result of experiments. In this figure the dotted line indicates the error rate for 1) and the solid line for 2). The best error rate was 8.49% when $n_r = 5$ for 1) and 7.55% when $n_e = 3$ for 2). The corresonding induced decision trees are shown shown in Fig. 7 ($n_r = 5$) and Fig. 8 ($n_e = 3$). The decrease of error rate levels off when the the number of chunking increases for both 1) and 2). The result shows that repeated applica-

Fig. 6. Result of experiment (beam width=1, without pessimistic pruning)

tion of chunking at every node results in constructing a decision tree with better predictive accuracy.

The second experiment focused on the effect of beam width, changing its value from 1 to 15 using pessimistic pruning. The number of chunking was fixed at the best number which was determined by the first experiment in Fig. 6, namely, $n_r = 5$ for 1) and $n_e = 3$ for 2).

Fig. 7. Example of constructed decision tree (chunking applied 5 times only at the root node, beam width = 1, with prepruning)

Fig. 8. Example of constructed decision tree (chunking applied 3 times at every node, beam width = 1, with prepruning)

The result is summarized in Fig. 11. The best error rate was 4.06% when the beam width = 12 for 1) (n_r = 5) and 3.77% when the beam width = 8 for 1) (n_e = 3). Examples of decision tree are shown in Fig. 9 and Fig. 10. Fig. 12 shows yet another result when prepruning was used.

Fig. 9. Example of constructed decision tree (chunking applied 5 times only at the root node, beam width = 12, with pessimistic pruning)

Fig. 10. Example of constructed decision tree (chunking applied 3 times at every node, beam width = 8, with pessimistic pruning)

The result reported in [7] is 3.8 % (they also used 10-fold cross validation) which is obtained by the M-of-N expression rules extracted from KBANN

(Knowledge Based Artificial Neural Network). The obtained M-of-N rules are too much complicated and not easy to interprete. Since KBANN uses domain knowledge to configure the initial artificial neural network, it is worth mentioning that DT-GBI that does not use any domain knowledge induced a decision tree with comparable predictive accuracy. Comparing the decision trees in Figs. 9 and 10, the trees are not stable. Both gives a similar predictive accuracy but the patterns in the decision nodes are not the same. According to [7], there are many pieces of domain knowledge and the rule conditions are expressed by the various combinations of these pieces. Among these many pieces of knowledge, the pattern (a → a → a → a) in the second node in Fig. 9 and the one (a → a → t → t) in the root node in Fig. 10 match their domain knowledge, but the others do not match. We have assumed that two nucleotides that are apart more than 10 nodes are not directly correlated. Thus, the extracted patterns have no direct links longer than 9. It is interesting to note that the first node in Fig. 9 relates two pairs (g → a) that are 7 nodes apart as a discriminatory pattern. Indeed, All the sequence having this pattern are concluded to be non-promoter from the data. It is not clear at this stage whether the DT-GBI can extract the domain knowledge or not. The data size is too small to make any strong claims.

Fig. 11. Result of experiment (with pessimistic pruning)

Fig. 12. Result of experiment (with prepruning)

5 Conclusion

This paper reports the current status of DT-GBI, which constructs a classifier (decision tree) for graph-structured data by GBI. Substructures useful for classification task, are constructed on the fly by applying repeated chunking in

GBI during the construction process of a decision tree. Newly introduced beam search is very effective in increasing the predictive accuracy. The performance of DT-GBI is evaluated through experiments on a classification problem of DNA promoter sequences from the UCI repository and the results show that DT-GBI is comparable to other method that uses domain knowledge in modeling the classifier.

Immediate future work includes to incorporate more sophisticated method for determining the number of cycles to call GBI at each node to improve prediction accuracy. Utilizing the rate of change of information gain by successive chunking is a possible way to automatically determine the number. Another important direction is to explore how the partial domain knowledge is effectively incorporated to constrain the search space. DT-GBI is currently being applied to much larger medical dataset.

Acknowledgment

This work was partially supported by the grant-in-aid for scientific research on priority area "Active Mining" (No. 13131101, No. 13131206) funded by the Japanese Ministry of Education, Culture, Sport, Science and Technology.

References

1. C. L. Blake, E. Keogh, and C.J. Merz. Uci repository of machine leaning database, 1998. http://www.ics.uci.edu/∼mlearn/MLRepository.html.
2. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks/Cole Advanced Books & Software, 1984.
3. T. Matsuda, T. Horiuchi, H. Motoda, and T. Washio. Extension of graph-based induction for general graph structured data. In *Knowledge Discovery and Data Mining: Current Issues and New Applications, Springer Verlag, LNAI 1805*, pages 420–431, 2000.
4. T. Matsuda, H. Motoda, T. Yoshida, and T. Washio. Knowledge discovery from structured data by beam-wise graph-based induction. In *Proc. of the 7th Pacific Rim International Conference on Artificial Intelligence, Springer Verlag, LNAI 2417*, pages 255–264, 2002.
5. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
6. J. R. Quinlan. *C4.5:Programs For Machine Learning*. Morgan Kaufmann Publishers, 1993.
7. G. G. Towell and J. W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.
8. G. Warodom, T. Matsuda, T. Yoshida, H. Motoda, and T. Washio. Classifier construction by graph-based induction for graph-structured data. In *Advances in Knowledge Discovery and Data Mining, Springer Verlag, LNAI 2637*, pages 52–62, 2003.
9. K. Yoshida and H. Motoda. Clip : Concept learning from inference pattern. *Journal of Artificial Intelligence*, 75(1):63–92, 1995.

Probabilistic Approach for Reduction of Irrelevant Tree-structured Data

Amaury Habrard, Marc Bernard, Marc Sebban

EURISE – Université Jean Monnet de Saint-Etienne
23, rue du Dr Paul Michelon – 42023 Saint-Etienne cedex 2 – France
{amaury.habrard,marc.bernard,marc.sebban}@univ-st-etienne.fr

Abstract. This article aims at pruning noisy or irrelevant subtrees in a set of trees. The originality of this approach, in comparison with classic techniques in prototype selection, comes not from the non-deletion of the whole tree, but rather of some of its subtrees. Our method is based on the computation of confidence intervals on a set of subtrees according to a probability distribution. We propose an approach to assess these intervals on this specific type of data and show experimentally its interest in the context of learning from noisy data.
Keywords. data reduction, prototype selection, tree-structured data, noisy data

1 Introduction

The use of structured or semi-structured data is increasing in research domains such as knowledge discovery in databases or machine learning. For example, a learning sample can be represented by a relational database. The increasing interest for directly exploiting such data has lead to a new field of research named multi-relational learning [1]. XML data are another example of structured representation. They are easily available in huge quantity on the web, and have stimulated the interest for tree-structured data.

The tree representation gives an interesting compromise between graphs and linear representations. Actually, they allow the expression of hierarchical dependences and are less costly for processing than graphs. A lot of recent works are interested in knowledge discovery from tree-structured data. For example we can cite the extraction of frequent trees [2, 3], or the detection of tree patterns, either in the framework of relational databases or in the one of XML data [4, 5]. Trees are also studied in machine learning, particularly in the inference of tree patterns [6, 7], or in tree automata induction [8, 9].

While trees are less complex data structures than graphs, they are nevertheless more difficult to process than linear representations. Consequently, in domains where the amount of data is huge, and the level of noise very high (in data mining for example), it seems interesting to select only relevant data to work on. This way to proceed allows one to reduce not only the complexity of data storage but also the generalization error of the induced models. These tasks

are the matter of data reduction which can be achieved via two ways: prototype selection (PS) [10] and feature selection (FS) [11]. It is important to note that these methods are usually applied to non-structured data and aim at totally deleting either an example (PS) or a feature (FS). However, in the context of tree-structured data, we assume that only some particular subtrees of a given tree are noisy or irrelevant. Then it is not necessary to completely delete this tree. The suppression of a subtree can be seen as an hybrid approach of data reduction. It looks like prototype selection when a tree is completely deleted, and a local feature selection when only subtrees are removed.

In this paper, we propose a probabilistic approach based on confidence intervals to prune irrelevant subtrees. In fact we study the probability of a subtree to be in a class issued from a partitioning of the whole set of subtrees (Section 2). In Section 3, we describe a partitioning method based on tree patterns. Section 4 deals with application of our method both on artificial and real data.

2 Pruning Subtrees

We consider the problem of inferring an estimation model in the presence of noisy tree-structured data. We aim at pruning noisy and irrelevant subtrees before the learning process. Our approach consists in estimating a probability distribution on the set of all the subtrees of the learning sample. We construct a partition of the learning subtrees, and for each element of this partition, we compute a confidence interval containing $(100 - \alpha)\%$ of the considered learning subtrees. Those with a too small probability are pruned.

2.1 Definitions and Notations

A tree has a root node and a set, eventually empty, of children. A leaf is a node without child. Trees, we are interested in, are constructed over a signature. Each node is labeled by a functional symbol, and all the nodes labeled by the same symbol have exactly the same number of children. Finally the symbols are typed and the children ordered.

Definition 1 *A signature Σ is a 4-tuple $(S, V, arity, \sigma)$. S is a finite set whose elements are called sorts. V is a finite set whose elements are called functional symbols. arity is a mapping from V into $\mathbb{N}$, $arity(f)$ is called the arity of f. σ is a mapping from V into S, $\sigma(s)$ will be called the sort of s. We denote Σ^T the set of trees defined relative to a signature Σ.*

The data reduction method we propose is based on a probability distribution defined on a set of subtrees. Thus we introduce some notations related to the subtrees of a sample of trees.

Definition 2 *Let T be a sample of trees. We denote $Sub(T)$ the set of the subtrees of T and $MSub(T)$ the multi-set of subtrees of T.*

For example, in the sample $E = \{h(f(a,b)), f(h(b),a)\}$, $Sub(E) = \{a, b, f(a,b), h(b), h(f(a,b)), f(h(b),a)\}$ and $MSub(E) = \{a, b, a, b, f(a,b), h(b), h(f(a,b)), f(h(b),a)\}$

Definition 3 *A position is a couple (f,p) (denoted $f.p$) where $f \in V$ and $p \in I\!N$ such that $1 \leq p \leq arity(f)$. A position allows us to design the subtree corresponding to the child number p of the symbol f (the children are ordered from left to right).*

For example, in the sample $E = \{h(f(a,b)), f(h(b),a)\}$, the subtree a is at position $f.2$ of the tree $f(h(b),a)$ and at position $f.1$ of the tree $f(a,b)$.

2.2 Estimation of a Probability Distribution

In a first step, we estimate a probability distribution relatively to a sample of trees T. This distribution then allows us to compute a probability for each subtree in $Sub(T)$. We use a similar approach to the "N-grams" often used as a model of natural language modeling [12]. This approach assumes that the probability of a given symbol in a string, can be computed using the $n-1$ previous symbols. We use a similar principle, with $n = 2$, computing the probability of a symbol relatively to its parent. Note that a different adaptation of this approach in the context of trees has already been proposed in [9]. For each symbol a of the sample, we assess via $\hat{p_c}(a \mid f.i)$ its probability to be the child number i of any symbol f. Formally:

$$\forall a \in V, \forall f \in V, \forall 1 \leq i \leq arity(f),\ \hat{p_c}(a \mid f.i) = \frac{\text{Number of occurrences of } a \text{ in } f.i}{\text{Number of occurrences of } f}$$

Moreover we estimate for each symbol its probability to be the root of a tree.

$$\forall a \in V,\ \ \hat{p_r}(a) = \frac{\text{Number of examples with } a \text{ for root}}{\text{Number of examples}}$$

Finally the estimation of the probability of a tree $t = f(t_1, \ldots, t_n)$ is computed as follows:

$$\hat{p_a}(f(t_1, \ldots, t_n)) = \hat{p_r}(f) \times \hat{p_{pos}}(t_1 \mid f.1) \times \cdots \times \hat{p_{pos}}(t_n \mid f.n)$$

where $\hat{p_{pos}}$ is recursively[1] defined by:

$$\hat{p_{pos}}(g(u_1, \ldots, u_n) \mid f.n) = \hat{p_c}(g \mid f.n) \times \hat{p_{pos}}(u_1 \mid g.1) \times \cdots \times \hat{p_{pos}}(u_n \mid g.n)$$

For example, the set $E = \{h(f(a,b)), f(h(b),a)\}$ allows us to define the following conditional probabilities:

$$\begin{array}{llll}
\hat{p_c}(a \mid f.1) = \frac{1}{2} & \hat{p_c}(a \mid f.2) = \frac{1}{2} & \hat{p_c}(a \mid h.1) = 0 & \hat{p_r}(a) = 0 \\
\hat{p_c}(b \mid f.1) = 0 & \hat{p_c}(h \mid f.2) = 0 & \hat{p_c}(b \mid h.1) = \frac{1}{2} & \hat{p_r}(h) = \frac{1}{2} \\
\hat{p_c}(h \mid f.1) = \frac{1}{2} & \hat{p_c}(b \mid f.2) = \frac{1}{2} & \hat{p_c}(h \mid h.1) = 0 & \hat{p_r}(b) = 0 \\
\hat{p_c}(f \mid f.1) = 0 & \hat{p_c}(f \mid f.2) = 0 & \hat{p_c}(f \mid h.1) = \frac{1}{2} & \hat{p_r}(f) = \frac{1}{2}
\end{array}$$

and the probability of the tree $t = h(f(a,b))$ is computed as follows:

$$\hat{p_a}(t) = \hat{p_r}(h) \times \hat{p_c}(f \mid h.1) \times \hat{p_c}(a \mid f.1) \times \hat{p_c}(b \mid f.2) = \left(\tfrac{1}{2}\right)^4 = \tfrac{1}{16}$$

If t is a subtree of a tree of the learning sample (t must be different from the whole tree), then we compute its probability using $\hat{p_{pos}}$ and taking into account

[1] The base case of recursion corresponds to symbols of arity 0

its position relatively to its parent. For example, the probability of the subtree $f(a,b)$ of the set E is computed as follows:

$$\hat{p_a}(f(a,b)) = \hat{p_c}(f \mid h.1) \times \hat{p_c}(a \mid f.1) \times \hat{p_c}(b \mid f.2) = \left(\frac{1}{2}\right)^3 = \frac{1}{8}.$$

2.3 Pruning with Confidence Intervals

In the previous section, we have proposed a way to construct a probability distribution from a set of trees E. We show here how we can use this distribution to *a priori* prune subtrees considered statistically irrelevant for the future learning process. We consider a subset S of $MSub(E)$ and we compute a confidence interval, according to a risk α. We look for an interval $[pmin; 1]$ such that a proportion $1-\alpha$ of subtrees has a probability greater than or equal $pmin$: $p(p_a(t) \geq pmin) = 1-\alpha$.

According to the Central Limit Theorem, the mean $\overline{\hat{p_a}(t_S)}$ of the probabilities of the elements of S follows a normal distribution with an expected value μ and a standard deviation $\frac{\sigma}{\sqrt{|S|}}$. Then the confidence interval around μ is defined as follows: $\mu \in \overline{\hat{p_a}(t_S)} \pm \frac{\hat{\sigma}}{\sqrt{|S|}} \times u_\alpha$, where $\hat{\sigma}$ is the estimated standard deviation and u_α the $(1-\alpha)$-percentile of the normal distribution. Let $pmin$ be the lower bound of this interval, such that $pmin = \overline{\hat{p_a}(t_S)} - \frac{\hat{\sigma}}{\sqrt{|S|}} \times u_\alpha$.

Once this lower bound is computed, our decision rule is to delete all the subtrees of S whose probability is lower than $pmin$. This leads us to define formally the notion of relevance of a subtree. This definition is in relation with the definition of irrelevant features proposed in [11].

Definition 4 *Let a distribution D on a set of subtrees S, modeling the same concept, and let α be a risk. A subtree t is $(1-\alpha)$-relevant in S if and only if $\hat{p_a}(t) \geq \overline{\hat{p_a}(t_S)} - \frac{\hat{\sigma}}{\sqrt{|S|}} \times u_\alpha$.*

To end this section introducing our pruning method, we synthesize the main steps of our approach in Algorithm 1[2] .

3 Partitioning Subtrees with Regular Tree Patterns

In our method we need to compare the probabilities of subtrees of a sample. We think that the comparison of the probabilities of two subtrees that never appeared in the same places (positions) in the sample is irrelevant. Indeed they *a priori* do not model the same concept and then we assume that they are not comparable. We propose in this section a method of partitioning using regular tree patterns, *i.e.* tree patterns with only one variable. This approach ensures that two subtrees that appear in the same context (*i.e.* subtrees with the same ancestors and siblings) will be in the same partition.

Definition 5 *A regular tree pattern is a tree defined on a signature $(S, V \cup \{X\}, \alpha, \sigma)$ where X is a variable and the tree has exactly one leaf labeled by X.*

[2] The probability distribution is adapted to delete subtrees with specific symbols

```
Data: E set of trees
      α real
begin
    Construct the probability distribution with E;
    T ← partitioning MSub(E);
    foreach S ∈ T do
        Compute an interval [pmin; 1] for the probabilities of the subtrees of S
        to the risk α;
        foreach t ∈ S do
            Adapt the probability distribution in deleting the instance of t;
            Delete t if it is not (1 − α)-relevant or if p_a(t) = 0;
            Rebuild the probability distribution in adding the instance of t;
        end
    end
end
```

Algorithm 1: Pruning by confidence interval

Let t a regular tree pattern and t' an ordinary tree. We denote $t._{\#}t'$ the substitution of the variable X of t by the tree t'.

To construct a partition of the multi-set of subtrees, our approach consists in extracting all the regular tree patterns definable from a sample of trees. The set of all regular tree patterns is given by $\{t \mid \exists t' \in MSub(E) \text{ and } t._{\#}t' \in E\}$.

Each pattern t allows us to define a class π_t of the partition of the multi-set of subtrees. All the subtrees which can be concatenated to t to obtain a tree of the learning sample belong to this partition[3]: $\pi_t = \{t' \in MSub(E) \mid t._{\#}t' \in E\}$.

4 Evaluation in the Context of Learning Stochastic Tree Automata

We now present experimental results which justify the interest of our method as a pre-process of the learning task. Since we work on trees, we propose to evaluate our data reduction approach in the framework of learning stochastic tree automata from a sample of trees [8]. In such a context, we have a sample of trees, supposed to be generated from a probability distribution. The objective is to learn the probabilistic model which has generated the data. We propose to compare the automata inferred with noisy data and those induced after the pruning process. We achieved two series of experiments. The first one deals with situations where the target automaton is *a priori* known. In this case, we can use a measure of distance between the inferred model to the target automaton. However, we do not always know this one. Then, we also evaluate our approach in a second series of experiments, using a perplexity measure. This criterion assesses the relevance of the model on a test sample.

[3] The complexity of the approach, relative to the size of trees, depends on the partitioning

4.1 Stochastic Tree Automata

A tree automata [13, 14] defines a regular language on trees as a finite automaton defines a regular language on strings. Stochastic tree automata are an extension of tree automata, defining a probability distribution on the tree language defined by the automaton. We use an extension of these automata taking into account the notion of type: stochastic many-sorted tree automata (SMTA) defined on a signature [15]. We do not detail here these automata and their learning method. The interested reader can refer to [8, 4]. We only specify that a learned stochastic tree automaton allows one to associate a probability to each tree of a sample. Since we apply a pruning method on a set of sorted trees, we have to ensure that pruned trees still respect a signature. Practically, we replace pruned subtree by a unique symbol which does not appear in the sample and has a different type.

4.2 Evaluation Criteria

Distance from the Target Automaton: [16] defines distances between two hidden Markov models introducing the co-emission probability, as the probability that two independent models generate the same string. [17] presents an adaptation of the co-emission to stochastic tree automata. The co-emission probability of two stochastic tree automata $M1$ and $M2$, constructed over the same signature Σ, is denoted $A(M1, M2)$ and defined as follows: $A(M1, M2) = \sum_{t \in \Sigma^T} P_{M_1}(t) * P_{M_2}(t)$. Where $P_{M_i}(t)$ is the probability of t given the model M_i. The co-emission probability allows us to define a distance D_a which can be interpreted as the measure of the angle between the vectors representing automata in a space where the base is the set of trees of Σ^T.

Definition 6 *The distance D_a between two automata $M1$ and $M2$ is defined by:*

$$D_a(M1, M2) = \arccos\left(\frac{A(M1,M2)}{\sqrt{A(M1,M1) * A(M2,M2)}}\right)$$

Perplexity measure In the case of tree automata, the quality of a model A can be evaluated by the average likelihood on a set of trees S relative to the distribution defined by A: $LL = \left(\frac{1}{\|S\|}\sum_{j=1}^{|S|} \log P_A(t_j)\right)$[4]. A perfect model can predict each element of the sample with a probability equal to one, and so $LL = 0$. In a general way we consider the perplexity of the test set which is defined by $PP = 2^{LL}$. A minimal perplexity ($PP = 1$) is reached when the model can predict the probability of each element of the test sample. Therefore we consider that a model is more predictive than another if its perplexity is lower. A problem occurs when a tree of the test sample cannot be recognized by the learned automaton A. Actually the probability of this example is 0 and the perplexity cannot be computed. To avoid this problem a classical method consists in smoothing the distribution of the learned model using an interpolation approach [18] with an unigram model A_0 recognizing all the examples. The probability of a tree t, in our experiments, is then given by:

$$\hat{P}(t) = 0.9 * p(t|A) + 0.1 * p(t|A_0)$$

[4] $\|S\|$ is the number of nodes in S and $P_A(t_j)$ is the probability of t_j according to A

4.3 Experimentations

Recall that our objective is to study the interest of our pruning method in the context of learning tree automata from noisy data. Then we need to be able to artificially corrupt the data. When we work with positive and negative instances, a classic approach to noise data is to invert the labels of some examples. In our framework, all examples are unlabeled (more exactly they are only positive) and we need to introduce the noise in the structure of the examples. Our noise protocol consists in changing a proportion γ of the leaves of the trees. In our experiments we artificially corrupt the training samples with values of γ varying from 1% to 50%, and we apply our pruning method with different values of α from 0.01 to 0.25. We present, at first, the results on the experiments considering a target automaton. Then we present those with a target automaton *a priori* unknown. We use two criteria of performance to evaluate the results: the mean of the quality measure (the distance D_a or the perplexity measure) on all the levels of noise and the standard deviation around this mean. We also test the significance of our results using a Student paired t-test over the means with a critical risk of 5%. Note that all the results presented correspond to the optimal α value.

Experimentations Knowing the Target Automaton We suppose here that the target automaton is *a priori* known. This automaton allows us to generate a sample of trees according to the distribution defined by this automaton. We first infer an automaton from a noisy sample (one for each level of noise). Then we apply our pruning method on the noisy sample with different values of α. We infer an automaton with each pruned sample. We finally measure the distance D_a between each of the inferred automata and the target one. We achieved our experiments on five artificial datasets. One on a tree grammar representing stacks of objects, one on a simple grammar representing conditional statements (denoted Cond.), one on a grammar on boolean expressions (denoted Bool.) and two other artificial datasets (Art1 and Art2). The results are synthesized on Table 1. We give, for each dataset, the size of the initial sample (IS) in the number of subtrees, and the percentage of reduction of our pruning method (Red). In the same table we also give the average distance D_a [5] observed between the target automaton and the inferred automata $\pm$ the standard deviation. $\overline{D_a\ I}$ represents the distance relative to the automaton learned on the initial noisy sample, and $\overline{D_a\ Pr}$ the distance to the one learned from the pruned sample. In order to assess the relevance of our approach, we decided to compare its performances with those obtained by a simple Monte Carlo selection (denoted by MCS in the table). To achieve this comparison, we randomly removed the same proportion of subtrees (i.e. Red %) from the initial multi-set of subtrees ($MSub(E)$). Then, the described results are obtained from learning sets with exactly the same size. The last two columns concern the significance of the results (**Sig1** for Pr vs I and **Sig2** for Pr vs MCS). The experiments have shown that, not only the pruning of the noisy sample allows us to learn an automaton closer to the target one with

[5] The means computed are the means of the distances on the different levels of noise

Base	IS	Red	$\overline{D_a\ I}$	$\overline{D_a\ Pr}$	$\overline{D_a\ MCS}$	Sig1	Sig2
Stacks	35724	5%	0.721 ± 0.514	0.420 ± 0.332	0.869 ± 0.496	yes	yes
Cond.	71863	5%	0.618 ± 0.187	0.590 ± 0.224	1.050 ± 0.316	no	yes
Bool.	43185	26%	0.350 ± 0.240	0.242 ± 0.230	0.227 ± 0.142	yes	no
Art1	33137	0.5%	1.230 ± 0.265	0.556 ± 0.237	1.374 ± 0.339	yes	yes
Art2	30113	7.4%	0.846 ± 0.398	0.599 ± 0.247	1.011 ± 0.485	yes	yes

Table 1. Distances D_a to the target automaton

an average reduction of 8.8% of subtrees, but also our approach is highly better than the Monte Carlo sampling. The obtained results are particularly interesting on the databases Stacks, Bool., Art1 and Art2. On these databases, the standard deviation of computed distances without pruning is significantly higher than the one of computed distances with pruning.

Experimentations without Knowing the Target Automaton In this context, we consider to have a characteristic sample of trees. We divide this sample in two sets: a training set and a test set. As we want to evaluate our approach in the context of noise, we corrupt the training set with the different levels of noise. We learn an automaton for each of the noisy samples. Then we apply our pruning method on each of these samples with the different values of α. We learn a tree automaton for each of the pruned sample obtained. Then we evaluate all the inferred automata on the test set with the perplexity measure. Note that the test set is never noised. To make our experiments we use a 5-folds cross-validation. We did our experiments on 8 datasets. We have used a sample of each of the five artificial databases presented in the previous section. We also evaluate our method on real data with a sample of the PKDD'02 discovery challenge[6] database (dataset on hepatitis) obtained as described in [4]. We also use the database Student Loan of the UCI Irvine [19] and a dataset on the toxicity of the tacrine molecule presented in [20]. These two datasets correspond to structured data in 1st order logic and were converted into trees according to the principle presented in [15]. The results are presented on Table 2. We give for each dataset the average percentage of sample reduction with the pruning method (Red). The average perplexity measure with noisy data without pruning ($\overline{P\ I}$) and the average perplexity obtained after the pruning process ($\overline{P\ Pr}$) ± the standard deviation. The last column indicates if the results are significant. The results have shown that the perplexity is reduced in most of the cases, except for the Stacks and the Tacrine, with an average reduction of 10% of the training sample. In most of the experiments, the standard deviation is lower after a pruning phase than with no pruning. This remark tends to confirm that our method is relatively robust.

We summarize our results in a concise way on Figure 1. Each dot represents a database. A dot under the bisecting line expresses the fact that the pruning approach is better than the one with no pruning.

[6] http://lisp.vse.cz/challenge/ecmlpkdd2002/

Base	Red	$\overline{P\ I}$	$\overline{P\ Pr}$	Significant
Stacks	6%	2.23 ±0.274	2.24 ± 0.279	no
Cond.	4.6%	1.73 ± 0.241	1.70 ± 0.202	no
Bool.	13%	3.49 ± 0.302	3.47 ± 0.270	yes
Art1	17%	3.61 ± 0.219	3.26 ± 0.210	yes
Art2	3%	4.23 ± 1.456	3.96 ± 0.837	no
PKDD02	0.5%	14.84 ± 7.498	13.52 ± 7.235	yes
Tacrine	12%	5.78 ± 4.171	6.01 ± 4.111	no
Student Loan	17%	11.9 ± 1.871	11.43 ± 2.112	yes

Table 2. Results with the perplexity measure

Fig. 1. Results of the experimentations

5 Conclusion

In this paper we have presented an original approach allowing to prune subtrees in a set of trees. This approach of data reduction can be considered as an hybrid approach. It can actually delete whole trees and is related to prototype selection. It also allows one to delete some parts of the examples and the method is close to feature selection, with the difference that a subtree is deleted locally for one example. Our pruning method is based on the utilization of confidence intervals computed from a probability distribution on subtrees appearing in a same context in the examples. Our experiments have shown that our approach is robust and allows us to learn more efficient models closer to the target one in terms of distance or with a smaller perplexity. Moreover, our method allows us to reduce the size of the learning sample, this can be crucial in some applications.

Since we worked in the framework of the inference of probabilistic models on trees, we have to avoid the deletion of too many subtrees. Indeed, in this context, we aim at estimating correctly the distribution of the data. If we delete too many subtrees, we modify considerably the distribution of the initial learning sample and then it may be almost impossible to infer a correct distribution. A simple solution can be to tune the parameter α. Another complementary point of view consists in working on the method of partitioning. A larger partitioning of the learning subtrees, *i.e.* with a high number of classes, may more often lead to delete fewer subtrees than a smaller one. Our approach was experimentally evaluated as pre-process of a learning task. Since its works on any database of trees, and because a production rule can be seen as a tree, we are currently working on the simplification of knowledge bases.

References

1. De Raedt, L.: Data mining in multi-relational databases. In: 4th European Conference on Principles and Practice of Knowledge. (2000) Invited talk.
2. Zaki, M.: Efficiently mining frequent trees in a forest. In: Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining (KDD). (2002)
3. Termier, A., Rousset, M., Sebag, M.: Treefinder: a first step towards xml data mining. In: International Conference on Data Mining ICDM02. (2002)
4. Habrard, A., Bernard, M., Jacquenet, F.: Generalized stochastic tree automata for multi-relational data mining. In: 6th International Colloquium on Grammatical Inference. ICGI 2002. Volume 2484 of LNAI., Amsterdam, Springer (2002) 120–133
5. Miyahara, T., Suzuki, Y., Shoudai, T., Uchida, T., Takahashi, K., Ueda, H.: Discovery of frequent tag tree patterns in semistructured web documents. In: PAKDD 2002, Taipei, Taiwan (2002)
6. Goldman, S.A., Kwek, S.S.: On learning unions of pattern languages and tree patterns. In: 10th Algorithmic Learning Theory conference. Volume 1720 of LNAI. (1999) 347–363
7. Amoth, T.R., Cull, P., Tadepalli, P.: On exact learning of unordered tree patterns. Machine Learning **44** (2001) 211–243
8. Carrasco, R., Oncina, J., Calera-Rubio, J.: Stochastic Inference of Regular Tree Languages. Machine Learning **44** (2001) 185–197
9. Rico-Juan, J., Calera-Rubio, J., Carrasco, R.: Stochastic k-testable tree languages and applications. In: ICGI 2002. Volume 2484 of LNAI., Amsterdam (Nederland), Springer-Verlag (2002) 199–212
10. Wilson, D., Martinez, T.: Reduction techniques for instance-based learning algorithms. Machine Learning **38** (2000) 257–286
11. John, G., Kohavi, R., Pfleger, K.: Irrelevant features and the subset selection problem. In: 11th International Conference on Machine Learning. (1994) 121–129
12. Brown, P., Pietra, V.D., deSouza, P., Lai, J., Mercer, R.: Class-based n-gram models of natural language. Computational Linguistics **18** (1992) 467–479
13. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó, Budapest (1984)
14. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications . Available on: `http://www.grappa.univ-lille3.fr/tata` (1997)
15. Bernard, M., Habrard, A.: Learning stochastic logic programs. In Rouveirol, C., Sebag, M., eds.: Work-in-Progress Track at the 11th International Conference on Inductive Logic Programming. (2001) 19–26
16. Lyngsø, R., Pedersen, C., Nielsen, H.: Metrics and similarity measures for hidden Markov models. In: 7th International Conference on Intelligent Systems for Molecular Biology, ISMB '99 Proceedings, Heidelberg, Germany, AAAI Press USA (1999) 178–186
17. Carrasco, R.C., Rico-Juan, J.R.: A similarity between probabilistic tree languages: application to xml document families. Pattern Recognition, in press (2002)
18. Ney, H., Essen, U., Kneser, R.: On structuring probabilistic dependences in stochastic language modelling. Computer Speech and Language **8** (1994) 1–38
19. Blake, C., Merz, C.: University of California Irvine repository of machine learning databases. `http://www.ics.uci.edu/∼mlearn/` (1998)
20. King, R., Srinivasan, A., Sternberg, M.: Relating chemical activity to structure: An examination of ILP successes. New Generation Computing, Special issue on Inductive Logic Programming **13** (1995) 411–434

Effective Rule Induction from Molecular Structures Represented by Labeled Graphs

Susanne Hoche[1⋆], Tamás Horváth[2,1⋆⋆], and Stefan Wrobel[1,2]

[1] Fraunhofer AiS, Institute for Autonomous intelligent Systems, Schloss Birlinghoven, D-53754 Sankt Augustin,Germany
[2] University of Bonn, Department of Computer Science III, Römerstraße 164, D-53117 Bonn, Germany
{susanne.hoche,tamas.horvath,stefan.wrobel}@ais.fraunhofer.de

Abstract. Acyclic conjunctive queries form a polynomially evaluable fragment of definite nonrecursive first-order Horn clauses. Labeled graphs, a special class of relational structures, provide a natural way for representing chemical compounds. We propose an algorithm specific to learning acyclic conjunctive queries predicting certain properties of molecules represented by labeled graphs. To compensate for the reduced expressive power of the hypothesis language and thus the potential decrease in classification accuracy, we combine acyclic conjunctive queries with constrained confidence-rated boosting. Preliminary experimental results indicate the potential of the method for problems involving labeled graphs.

1 Introduction

Machine Learning is traditionally concerned with the problem of approximating an unknown target function $f : X \to Y$, where the domain or *instance space* X is the Cartesian product of a fixed set of *attributes*. Attributes are usually unordered or linearly ordered sets. Despite the number of successful real-world applications using attribute-value representation of the instances, the need of applying other representation languages in machine learning has long been recognized. One obvious argument has been the problem that attribute-value representation is not appropriate for describing learning tasks involving instances with complex structures. Multi-relational learning, also referred to as Inductive Logic Programming (ILP) [22, 30], is one of the most successful directions among the approaches of considering more expressive representation languages in learning.

In ILP, various classes of first-order languages are used to describe the input (i.e., examples and background knowledge) and output (i.e., hypotheses) components of the learning algorithms. First-order languages, on the one hand, provide a natural way for describing learning problems over structurally complex instance spaces. In addition, hypotheses in this language are relatively easy to understand for users. On the other

⋆ Partially supported by the DFG project (WR 40/1-3) *Nachhaltige Informationsfusion: Aktives Lernen*

⋆⋆ Partially supported by the DFG project (WR 40/2-1) *Hybride Methoden und Systemarchitekturen für heterogene Informationsräume*

hand, however, serious decidability and complexity problems may arise from their use during the learning process. As an example, the membership problem, i.e., the problem of deciding whether an instance belongs to the concept represented by an hypothesis, becomes undecidable in first-order logic. To control such problems, different techniques (e.g., hypothesis language and search biases) have been proposed in ILP.[1]

Labeled graphs are one of the most important tools describing objects and the way they are connected. They are relational structures defined usually over vocabularies consisting of a single binary and a finite set of unary predicate symbols. They provide, in particular, a natural way for representing chemical compounds. Although ILP is concerned with learning from relational structures, and many ILP applications have been devoted to computational chemistry, surprisingly there are only few results (see, e.g., [12]) in the direction of restricting instances to labeled graphs. Such a structural assumption could then be exploited in the learning process to control decidability and complexity problems mentioned above.

In this work, we propose a boosted algorithm designed to learn acyclic conjunctive queries predicting unknown properties of chemical compounds. Our algorithm assumes that compounds are represented by relational structures corresponding to labeled graphs. We consider learning problems of the following form: *Given* disjoint sets E^+ and E^- of labeled graphs representing chemical compounds, *find* a set of definite first-order Horn clauses consistent (within some error) with E^+ and E^-. Since examples are disjoint labeled graphs, we use the learning from interpretations ILP setting [7] as the most plausible model for our purpose. In the algorithm presented, we apply top-down induction, a popular technique based on refinement operators [23] for first-order clauses. In our approach, refinement operators are defined by building blocks. In this work, we assume that such building blocks are provided by an expert. We are working on automatic extraction of building blocks for labeled graphs. We will discuss this problem later on.

In computational chemistry, pattern matching is usually defined by subgraph isomorphism. Since subgraph isomorphism generalizes the Hamiltonian path problem, it is NP-complete. In planar graphs, however, it can be solved in linear time for any pattern of constant size [9]. The importance of this result is that many molecules can be represented by planar graphs. In contrast to this approach, we define pattern matching by first-order logical implication, which in turn is equivalent to homomorphism [20] between relational structures in the problem setting considered. Since isomorphisms are special homomorphisms, we thus apply a more general operator in pattern matching. This may be important e.g. in those applications, where the length of paths connecting substructures is not relevant. Homomorphism between finite relational structures generalizes the graph vertex k-coloring problem, and is thus NP-hard. It becomes, however, polynomial for patterns of small *tree-width* [26]. Intuitively, tree-width measures the degree of cyclicity of structures. In this paper, we restrict the search space to patterns of tree-width one, also referred to as acyclic patterns. We note that homomorphism for this fragment is LOGCFL-complete [14] and is therefore highly parallelizable.

[1] We note that limitations regarding expressive power are not resolved completely by first-order logic, as first-order sentences are only able to capture *local properties* of structures (see, e.g., [8]).

In [17], we have presented a greedy algorithm for learning acyclic patterns. By this restriction, however, we reduce the expressive power of the hypothesis language. To compensate for the reduced expressiveness and thus the decrease in classification accuracy potentially resulting from it, in the proposed algorithm we combine acyclic patterns with confidence-rated boosting [27]. Ensemble methods, in particular boosting, are successful tools for increasing the prediction accuracy of classification learners by combining a set of only moderately accurate base hypotheses into one highly accurate strong hypothesis. Boosting works by repeatedly calling a base learner on reweighted versions of the training data, and thereby constructing an ensemble of specialized rules, or base hypotheses, which are finally combined into one prediction by weighted majority vote. In the framework of confidence-rated boosting, each base hypothesis not only predicts a classification but also generates a confidence score for this prediction.

The rest of the paper is organized as follows. In Section 2, we first review the necessary notions and results related to acyclic conjunctive queries. Section 3 is devoted to constrained confidence-rated boosting. In Section 4, we present our algorithm, and in Section 5, we empirically evaluate it on the domain of mutagenicity [19]. Finally, in Section 6 we conclude and discuss directions for future works.

2 Acyclic Conjunctive Queries

In this work, we restrict the hypothesis space to acyclic conjunctive queries, a practically relevant, efficiently evaluable fragment of first-order definite Horn-clauses. As an advantage over other ILP approaches using standard PROLOG evaluation techniques, we note that acyclic conjunctive queries allow evaluation of a set of instances in one single step. In this section we repeat the necessary notions related to acyclic conjunctive queries from our previous work [17]. In the Appendix, we give further details on acyclic conjunctive queries. For a detailed introduction to acyclic conjunctive queries the reader is referred to e.g. [1].

Throughout this section, we consider vocabularies consisting of a set of constant symbols, a distinguished predicate symbol called the target predicate, and a set of predicates called the background predicates. Thus, (non-constant) function symbols are not included in the vocabulary. Examples are ground atoms of the target predicate, and the background knowledge is an extensional database consisting of ground atoms of the background predicates. Furthermore, we assume that hypotheses are definite non-recursive first-order clauses, or in the terminology of relational database theory, conjunctive queries of the form $L_0 \leftarrow L_1, \ldots, L_l$, where L_0 is a target atom, and L_i is a background atom for $i = 1, \ldots, l$.

In order to define a special class of conjunctive queries, called *acyclic* conjunctive queries, we first need the notion of acyclic hypergraphs. A *hypergraph* (or *set-system*) $H = (V, E)$ consists of a finite set V called *vertices*, and a family E of subsets of V called *hyperedges*. A hypergraph is α-acyclic [10], or simply *acyclic*, if one can remove all of its vertices and edges by deleting repeatedly either a hyperedge that is empty or is contained by another hyperedge, or a vertex contained by at most one hyperedge [15, 31]. Note that acyclicity as defined here is not a *hereditary* property, in contrast to e.g. the standard notion of acyclicity in ordinary undirected graphs, as it may happen that

an acyclic hypergraph has a cyclic subhypergraph. For example, consider the hypergraph $H = (\{a, b, c\}, \{e_1, e_2, e_3, e_4\})$ with $e_1 = \{a, b\}$, $e_2 = \{b, c\}$, $e_3 = \{a, c\}$, and $e_4 = \{a, b, c\}$. This is an acyclic hypergraph, as one can remove step by step first the hyperedges e_1, e_2, e_3 (as they are subsets of e_4), then the three vertices, and finally, the empty hypergraph is obtained by removing the empty hyperedge that remained from e_4. On the other hand, the hypergraph $H' = (\{a, b, c\}, \{e_1, e_2, e_3\})$, which is a subhypergraph of H, is cyclic, as there is no vertex or edge that could be deleted by the above definition. In [10], other degrees of acyclicity are also considered, and it is shown that among them, α-acyclic hypergraphs form the largest class properly containing the other classes.

Using the above notion of acyclicity, now we are ready to define the class of acyclic conjunctive queries. Let Q be a conjunctive query and L be a literal of Q. We denote by Var(Q) (resp. Var(L)) the set of variables occurring in Q (resp. L). We say that Q is acyclic if the hypergraph $H(Q) = (V, E)$ with $V = \text{Var}(Q)$ and $E = \{\text{Var}(L) : L \text{ is a literal in } Q\}$ is acyclic. For instance, from the conjunctive queries

$$P(X, Y, X) \leftarrow R(X, Y), R(Y, Z), R(Z, X)$$
$$P(X, Y, Z) \leftarrow R(X, Y), R(Y, Z), R(Z, X)$$

the first one is cyclic, while the second one is acyclic.

3 Constrained Confidence-Rated Boosting

Boosting has established itself as a successful method for improving the classification accuracy of a learning system by combining the predictions of several base classifiers learned in iterative calls to the underlying learner. Numerous algorithms have emerged which demonstrate superior performance on a broad range of application problems (see, e.g., [11, 25, 5, 24, 16]).

The idea common to all boosting algorithms is to "boost" a weak learner performing only slightly better than random guessing into an arbitrarily accurate learner by repeatedly calling it on reweighted versions of the training data, and thereby constructing an ensemble of specialized rules, or base hypotheses. Predictions are based on all members of the learned ensemble by combining the individual predictions by weighted majority vote into one strong hypothesis.

The reweighted versions of the training set $E = E^+ \cup E^-$ on which the base learner is repeatedly called are obtained by maintaining a probability distribution D^t over E modeling the weight D_i^t associated with each training example e_i in the t-th iteration of boosting. D_i^t indicates the influence of an instance e_i when learning a base classifier C_t. Initially, the influence of all the instances is identical, i.e., the probability distribution D^1 is uniform. In each iterative call t of the base learner, a base hypothesis C_t with an associated weight $\tilde{c}_t$ is learned based on E weighted according to the current distribution D^t.

In the framework of confidence-rated boosting, the prediction of a base hypothesis C_t is confidence-rated. The sign of $\tilde{c}_t$ indicates the label predicted by C_t to be assigned to an instance, whereas the absolute value of $\tilde{c}_t$ is interpreted as C_t's prediction confidence, or the reliability of C_t's prediction. A base hypothesis' prediction confidence is,

on the one hand, used as its vote in the final, strong, hypothesis H, and, on the other hand, to update the distribution D^t for the next iteration of the base learner. The distribution is modified such that the weights of misclassified instances are increased while the weights of correctly classified instances are decreased. This way, the learner has to focus on those examples which are not correctly classified by the current ensemble.

Depending on the exact framework, a base hypothesis can apply distinct prediction confidences to different examples. Here, we employ a form of confidence-rated boosting in which a base hypothesis is restricted to make a prediction only for those examples which are covered by it, and to abstain otherwise. We furthermore restrict, following Cohen and Singer's approach to constrained confidence-rated boosting [5], the base hypotheses to either of two forms. A hypothesis either predicts, in the binary case we deal with here, the positive class with a positive prediction confidence, or it is the default hypothesis, just comprising the target predicate to be learned and satisfying all examples, with an assigned negative confidence.

We note that hypotheses obtained by boosting algorithms are potentially more complex than those generated by standard ILP learning systems. However, constraining the base hypotheses to either of the above two forms improves comprehensibility of strong hypotheses.

As suggested by [5], we aim at minimizing the ensemble's training error by searching in each round of boosting for a base hypothesis maximizing the objective function $\tilde{Z}$ which is, for a base hypothesis C_t, defined based on the collective weight of all positive and negative instances covered by C_t (in what follows, $x \in C$ denotes that instance x is covered by hypothesis C):

$$\tilde{Z}(C_t) = \sqrt{\sum_{x_i \in E^+, x_i \in C_t} D_i^t} - \sqrt{\sum_{x_i \in E^-, x_i \in C_t} D_i^t} \ . \tag{1}$$

After the last iteration of the base learner, the strong hypothesis H is formed on the basis of all hypotheses C_t learned over the course of iterations, and their assigned prediction confidences $\tilde{c}_t$ defined by

$$\tilde{c}_t = \frac{1}{2} \ln \left(\frac{\sum_{x_i \in E^+, x_i \in C_t} D_i^t + \frac{1}{2m}}{\sum_{x_i \in E^-, x_i \in C_t} D_i^t + \frac{1}{2m}} \right) \ , \tag{2}$$

where $m = |E|$ (see also [5]). To classify an instance x, the prediction confidences of all base hypotheses covering x are summed up. If this sum is positive, the strong hypothesis classifies x as positive, otherwise x is classified as negative:

$$H(x) = sign \left(\sum_{h_t} h_t(x) \right) \ , \tag{3}$$

where $h_t : X \rightarrow \Re$ is defined by

$$h_t(x) = \begin{cases} \tilde{c}_t & \text{if } x \in C_t \\ 0 & \text{otherwise} \ . \end{cases} \tag{4}$$

Algorithm 1 BACQ

Require: set E of $+/-$ labeled ground P-atoms and a labeled graph represented by ground A and B atoms

Ensure: a set of confidence-rated acyclic conjunctive queries

let C_{default} be the unit clause $P(X_1, \ldots, X_{m_P}) \leftarrow$
let $D(x_i) = 1/m$ for $i = 1, \ldots, m$ // where $m = |E|$
for $t = 1, \ldots, T$ **do**
  for $k = 1, \ldots, K$ **do**
    $C_k = C_{\text{default}}$
    while $\exists C \in \mathcal{R}(C_k, N)$ such that $\widetilde{Z}(C) > \widetilde{Z}(C_k)$, // see (1) for the definition of $\widetilde{Z}$
      where $\mathcal{R}(C_k, N)$ is a set containing (at most)
      N randomly selected acyclic refinements of C_k **do**
      $C_k = C$
    end while
  end for
  let $C_t = C_j$ satisfying $\widetilde{Z}(C_j) = \max_{k=1,\ldots,K} \widetilde{Z}(C_k)$
  let $R_t = \begin{cases} C_t & \text{if } \widetilde{Z}(C_t) > |\widetilde{Z}(C_{\text{default}})| \\ C_{\text{default}} & \text{otherwise} \end{cases}$
  let $Cov \subseteq E$ be the set of examples covered by R_t
  // Cov is computed in a single step (see Algorithm EVALUATE in the Appendix)
  for $i = 1, \ldots, m$ **do**
    if $x_i \in Cov$ **then**
      let $D(x_i) = D(x_i) \cdot e^{-y_i \cdot \widetilde{c}_{R_t}}$
      // where $y_i \in \{+1, -1\}$ according to the label of x_i, and $\widetilde{c}_{R_t}$ is defined in (2)
    end if
  end for
  let $D(x_i) = D(x_i)/Z_t$ for $i = 1, \ldots, m$, where $Z_t = \sum_{i=1,\ldots,m} D(x_i)$
end for
return $\{(R_1, \widetilde{c}_{R_1}), \ldots, (R_T, \widetilde{c}_{R_T})\}$

4 Boosting Acyclic Conjunctive Queries

In this section, we present an algorithm designed to learn acyclic conjunctive queries predicting unknown properties of chemical compounds. Our algorithm assumes that compounds are represented by relational structures corresponding to labeled graphs. More precisely, we assume without loss of generality that the vocabulary consists of a target predicate P of arity m_P, and predicates A and B of arities m_A and m_B, respectively. For each compound, we have a ground target atom of the form $P(a_1, \ldots, a_{m_P})$, where a_1 is the identifier of the compound, and $a_2, \ldots, a_{m_P}$ are attribute values for the whole compound (e.g., ϵ-lumo). Each (chemical) atom is described by a fact of the form $A(b_1, \ldots, b_{m_A})$, where b_1 is the identifier of the compound containing the atom, b_2 is the atom's identifier, and $b_3, \ldots, b_{m_A}$ are attribute values for the atom. Finally, each bond is represented by a ground atom of the form $B(c_1, \ldots, c_{m_B})$, where c_1 is the compound identifier, c_2, c_3 are the identifiers of the atoms connected by the bond, and $c_4, \ldots, c_{m_B}$ are attribute values for the bond. Thus, the ground A and B atoms

represent the (labeled) vertices and (labeled) edges of the labeled graphs. We note that our approach assumes that molecules are represented in the learning from interpretation setting [7].

The algorithm combining top-down induction of acyclic conjunctive queries with constrained confidence-rated boosting is given in Algorithm 1. In Steps 1 and 2 of the algorithm, we first initialize the target clause and the distribution over the set of training examples. Then, we learn T weak hypotheses (Steps 3–19 of the Algorithm), where T is a user defined parameter.

To find a weak hypothesis, i.e., an acyclic conjunctive query, we apply top-down induction using the following refinement operator. We first select at random a literal with one of the predicate symbols P, A, or B from the clause to be refined. Then, depending on its predicate symbol, we add a set of literals to the clause as follows. If the selected literal is a P-literal (i.e., it is the head of the clause), with the same probability,

- either an atom or an acyclic building block (e.g., a benzene ring) is added to the clause,
- or one of the attributes of the P-atom is selected at random, and the best specialization for this attribute with respect to $\widetilde{Z}$ defined in Eq. (1) is computed.

If an A-atom (i.e., a labeled vertex in the graph) has been selected, we add to the clause

- either literals representing a labeled edge ending in this vertex,
- or an acyclic building block containing the selected vertex,
- or constraints specializing one of the attributes of the vertex in a similar fashion as described above.

Finally, for B-atoms, i.e., for labeled edges,

- we add either a set of literals defining an acyclic building block,
- or compute the best value for one of its attributes.

We note that none of the above refinements violates the acyclicity property. In particular, building blocks are restricted to be acyclic, and thus, as they share at most one edge with the labeled acyclic graph corresponding to C_k in Step 6, adding such a building block always results in an acyclic clause.

At the same time of adding a set of literals (i.e., atom, bond, or acyclic building block) to the clause, for each attribute of the literals we compute the best value with respect to $\widetilde{Z}$ and specialize the attribute with the value for which $\widetilde{Z}$ is maximal. In contrast to the greedy search used in [5], in Steps 5–8 of the algorithm, we apply simple local search for finding an acyclic conjunctive query. That is, we start the local search with the default clause, and refine it as long as its randomly selected refinement improves the quality measured by $\widetilde{Z}$. We repeat the random local search algorithm K times (see Step 4 of the algorithm) and select the acyclic conjunctive query of the best quality (Step 10). In Steps 12–18, we then update the distribution over the training set.

4.1 Building Blocks

In the work presented here, we assume that buildings blocks are provided by an expert. However, we are working on the automatic extraction of acyclic building blocks. Since structures are restricted to labeled chemical graphs, we are going to consider only cycle and tree patterns. To extract cycle patterns, we are going to compute the set of cycles of length ℓ, for every $\ell = 2, \ldots, L$. L is a user defined parameter bounding the length of cycles. For a fixed ℓ, this can be done by evaluating first the acyclic conjunctive query

$$\begin{aligned} cycle(Y_1, \ldots, Y_\ell) \leftarrow\ & B(X, Y_1, Y_2, Z_{1,1}, \ldots, Z_{1,m_B-3}), \\ & B(X, Y_2, Y_3, Z_{2,1}, \ldots, Z_{2,m_B-3}), \\ & \vdots \\ & B(X, Y_\ell, Y_1, Z_{\ell,1}, \ldots, Z_{\ell,m_B-3}) \end{aligned}$$

and then removing from the answer set the tuples $(a_1, \ldots, a_\ell)$ satisfying $a_i = a_j$ for some $1 \leq i < j \leq \ell$. By this step we filter those closed walks that are not cycles. If the remaining set is nonempty then for all possible subsets $\{V_1, \ldots, V_k\}$ of the nominal attributes of A and B, and for all possible combinations $\{v_1, \ldots, v_k\}$ of the values of these attributes, we can check in a similar way, whether there is a cycle of length ℓ such that $V_i = v_i$ in each atom and in each bond in the cycle for every $i = 1, \ldots, k$. This method is exponential in the number of nominal attributes of A and B. However, it is effective if the number of nominal attributes is small. To extract tree patterns, we are going to investigate the labeled graph obtained by removing all edges occurring in a cycle.

5 Empirical Results on the Domain of Mutagenicity

We evaluated our approach on the ILP-benchmark problem of *Mutagenicity* [19]. The learning task is to predict the mutagenicity of nitroaromatic compounds. Mutagenic compounds are often known to be carcinogenic and to cause damage to the DNA. Not all compounds can be empirically tested for mutagenicity, and the prediction of mutagenicity is vital to understanding and predicting carcinogenesis.

Of the several relational descriptions that are available for the domain [29], we use the strongly structured description $\mathcal{B}_4$ which comprises a description of the atoms of the molecules and the bonds between these atoms; global properties of the molecule as e.g. their hydrophobicity; chemical structures present in the molecules as e.g. benzenic or methylic groups.

Here, we consider the subset of 188 so called regression-friendly compounds 125 of which are classified as having positive levels of mutagenicity. The predictive accuracy is estimated by 10-fold-cross-validation, where we use the same folds as [29] for their experiments with Progol. The accuracy and standard deviation obtained in our experiments with BACQ is displayed in Table 1 for various numbers of iterations ranging from 50 to 400, together with reference results on the same dataset using background knowledge $\mathcal{B}_4$, and the sources from which these results are reported. ACQ is an ILP

Table 1. Accuracy $\pm$ standard deviation for the Mutagenicity domain for BACQ with different numbers of iterations ranging from 50 to 400 (i.e., T in Algorithm 1), in comparison to other systems

ACQ [17]	C^2RIB [16]	FOIL [19]	Fors [18]	G-Net [2]	Progol [19]	STILL [28]
87.0	88.0	82.0	89.0	92.0	88.0	90.0
n/a	$\pm$3.4	$\pm$3.0	$\pm$6.0	$\pm$8.0	$\pm$2.0	$\pm$5.0

BACQ 50	BACQ 100	BACQ 150	BACQ 200	BACQ 250	BACQ 300	BACQ 350	BACQ 400
89.9 $\pm$4.6	89.9 $\pm$4.6	91.5 $\pm$3.8	90.4 $\pm$4.9	92.0 $\pm$3.8	92.0 $\pm$3.8	91.5 $\pm$3.8	91.5 $\pm$4.5

learner based on acyclic conjunctive queries which we previously introduced [17] and which serves as a starting point for the work presented in this paper.

The classification accuracy obtained after only T=50 iterations with BACQ is lower than, however in the range of the standard deviation of, the best result reported so far for the Mutagenicity domain, accomplished with the system G-Net [2]. The result is also on par with the second best result reported for the domain, achieved with the system STILL [28]. For increasing T=250, and T=300, respectively, the classification accuracy is identical to the best one reported for this domain, G-Net [2], however with only half the standard deviation. Irrespective of the number of boosting iterations, our results with BACQ lie well in the range of the standard deviations reported for the learning systems most successful on the Mutagenicity domain. The classification accuracy of ACQ is significantly outperformed by any result of BACQ.

In our experiments, carbon_5_aromatic_ring, carbon_6_ring, carbon_5_ring, hetero_aromatic_6_ring, hetero_aromatic_5_ring, ring6, ring5 were used as cycle building blocks, and nitro and methyl as tree building blocks (see also [19]). As an example, the acyclic building block defining carbon_5_aromatic_ring is

$$\begin{aligned}&\text{carbon_5_aromatic_ring}(X, Y_1, \ldots, Y_5) \leftarrow \\ &\quad \text{atom}(X, Y_1, c, U_1, V_1), \ldots, \text{atom}(X, Y_5, c, U_5, V_5), \\ &\quad \text{bond}(X, Y_1, Y_2, 7), \text{bond}(X, Y_2, Y_3, 7), \ldots, \text{bond}(X, Y_5, Y_1, 7)\end{aligned}$$

6 Conclusion

In this paper, we have presented an algorithm specific to learning acyclic conjunctive queries predicting unknown properties of chemical compounds. Here, chemical compounds have been represented by relational structures corresponding to labeled graphs. In our work, building blocks have been used for top-down induction of acyclic conjunctive queries. In the experiments, we have assumed that such buildings blocks were

provided by an expert. Although this seems to be a reasonable assumption when considering chemical graphs, we are working on the automatic extraction of cycle and tree patterns as building blocks.

In ILP, examples are usually evaluated one by one (by some PROLOG system). One of the major advantages of our approach is that acyclic conjunctive queries allow, in contrast to the standard ILP evaluation approach, examples to be evaluated in one step.

Restricting the search space to acyclic patterns implies, however, a reduced expressiveness and a potential decrease in classification accuracy. These shortcomings are counteracted by applying constrained confidence-rated boosting. Our first experiments indicate that combining acyclic conjunctive queries with constrained confidence-rated boosting has indeed a potential for real-world problems involving labeled graphs. As future work, we are going to evaluate the method on further such domains.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Mass., 1995.
2. C. Anglano, A. Giordana, G. Lo Bello, and L. Saitta. An experimental evaluation of coevolutive concept learning. *Proc. of the 15th Int. Conf. on Machine Learning*, 1998.
3. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.
4. P. A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.
5. W. Cohen and Y. Singer. A Simple, Fast, and Effective Rule Learner. *Proc. of 16th National Conference on Artificial Intelligence*, 1999.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.
7. L. De Raedt and S. Dzeroski. First-Order jk-Clausal Theories are PAC-Learnable. *Artificial Intelligence*, 70(1-2): 375-392, 1994.
8. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory* Springer-Verlag,Berlin,1995.
9. D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms & Applications*, 3(3):1–27,1999.
10. R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.
11. Y. Freund, and R.E. Schapire. Experiments with a New Boosting Algorithm. *Proc. of 13th International Conference on Machine Learning*, 1996.
12. T. Gärtner, P. A. Flach, and S. Wrobel. On Graph Kernels: Hardness Results and Efficient Alternatives. *The Sixteenth Annual Conference on Computational Learning Theory and The Seventh Kernel Workshop (COLT-2003)*. To Appear.
13. G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.
14. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 706–715. IEEE Computer Society Press, 1998.
15. M. Graham. On the universal relation. Technical report, Univ. of Toronto, Toronto, Canada, 1979.

16. S. Hoche and S. Wrobel. Relational Learning Using Constrained Confidence-Rated Boosting. *Proc. 11th Int. Conf. on Inductive Logic Programming (ILP)*, 2001.
17. T. Horvath and S. Wrobel. Toward Discovery of Deep and Wide First-Order Structures: A Case Study in the Domain of Mutagenicity. *Proc. Discovery Science*, 2001.
18. A. Karalic. *First Order Regression*. PhD thesis, University of Ljubljana, Faculty of Computer Science, Ljubljana, Slovenia, 1995.
19. A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85:277-299, 1996.
20. Kolaitis and Vardi. Conjunctive-query containment and constraint satisfaction. *JCSS: Journal of Computer and System Sciences*, 61(2):302–332, 2000.
21. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.
22. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19/20:629–680, 1994.
23. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNAI*. Springer, Berlin, 1997.
24. D. Opitz, and R. Maclin. Popular Ensemble Method: An Empirical Study. *Journal of Artificial Intelligence Research 11, pages 169-198*, 1999.
25. J.R. Quinlan. Bagging, boosting, and C4.5. *Proc. of 14th Nat. Conf. on AI*, 1996.
26. N. Robertson, and P.D. Seymour. Graph minors II: algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322,1986.
27. R. E. Schapire, and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Proceedings of COLT'98*, pages 80–91, 1998.
28. M. Sebag. Distance Induction in First Order Logic. *Proc. 7th Int. Workshop on Inductive Logic Programming (ILP)*, 1997.
29. A. Srinivasan, S. Muggleton, and R. King. Comparing the use of background knowledge by inductive logic programming systems. *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.
30. S. Wrobel. Inductive logic programming. In G. Brewka, editor, *Advances in Knowledge Representation and Reasoning*, pages 153–189. CSLI-Publishers, Stanford, CA, USA, 1996. Studies in Logic, Language and Information.
31. C. T. Yu and Z. M. Ozsoyoglu. On determining tree query membership of a distributed query. *INFOR*, 22(3), 1984.

Appendix: Acyclic Conjunctive Queries

In this appendix we give an algorithm for acyclic conjunctive query evaluation. In [3] it is shown that the class of acyclic conjunctive queries is identical to the class of conjunctive queries that can be represented by *join forests* [4]. Given a conjunctive query Q, the join forest $JF(Q)$ representing Q is an ordinary undirected forest such that its vertices are the set of literals of Q, and for each variable $x \in \text{Var}(Q)$ it holds that the subgraph of $JF(Q)$ consisting of the vertices that contain x is connected (i.e., it is a tree).

Now we show how to use join forests for efficient acyclic query evaluation. Let E be a set of ground target atoms, B be a set of ground atoms, and let Q be an acyclic conjunctive query with join forest $JF(Q)$. In order to find the subset $E' \subseteq E$ implied by Q with respect to B, we can apply the following method. Let $T_0, T_1, \ldots, T_k$ ($k \geq 0$)

algorithm EVALUATE

```
input: extensional database D and join tree T with root
       labeled by n_0
output: {n_0θ: θ is a substitution mapping the nodes of T into D}

let R = {n_0θ: θ is a substitution mapping n_0 into D}
let the children of n_0 be labeled by n_1,...,n_k (k ≥ 0)
for i = 1 to k
  S = evaluate(D,T_i)     // T_i is the subtree of T rooted at n_i
  R = the natural semijoin of R and S wrt. n_0 and n_i
return R
```

denote the set of connected components of $JF(Q)$, where T_0 denotes the tree containing the head of Q, and let $Q_i \subseteq Q$ denote the query represented by T_i for $i = 0, \ldots, k$. The definition of the Q_i's implies that they form a partition of the set of literals of Q such that literals belonging to different blocks do not share common variables. Therefore, the subqueries $Q_0, \ldots, Q_k$ can be evaluated separately; if there is an i, $1 \leq i \leq k$, such that the Boolean conjunctive query Q_i (i.e., a conjunctive query with empty head) is false with respect to B then Q implies *none* of the elements of E with respect to B, otherwise Q and Q_0 imply the same subset of E with respect to B. By definition, Q_0 implies an atom $e \in E$ if there is a substitution mapping the head of Q_0 to e and the atoms in its body into B, and Q_i ($1 \leq i \leq k$) is true with respect to B if there is a substitution mapping Q_i's atom into B. That is, using algorithm EVALUATE given below, Q implies E' with respect to B if and only if

$$(E' \subseteq \textsc{Evaluate}(B \cup E, T_0)) \wedge \left(\bigwedge_{i=1}^{k} (\textsc{Evaluate}(B, T_i) \neq \emptyset) \right) .$$

It remains to discuss the problem of how to compute a join forest for an acyclic conjunctive query. Using maximal weight spanning forests of ordinary graphs, in [4] Bernstein and Goodman give the following method to this problem. Let Q be an acyclic conjunctive query, and let $G(Q) = (V, E, w)$ be a weighted graph with vertex set $V = \{L : L \text{ is a literal of } Q\}$, edge set $E = \{(u, v) : \text{Var}(u) \cap \text{Var}(v) \neq \emptyset\}$, and with weight function $w : E \to \mathbb{N}$ defined by $w : (u, v) \mapsto |\text{Var}(u) \cap \text{Var}(v)|$. Let $MSF(Q)$ be a maximal weight spanning forest of $G(Q)$. Note that maximal weight spanning forests can be computed in polynomial time (see, e.g., [6]). It holds that if Q is acyclic then $MSF(Q)$ is a joint forest representing Q. In addition, given a maximal weight spanning forest $MSF(Q)$ of a conjunctive query Q, instead of using the method given in the definition of acyclic hypergraphs, in order to decide whether Q is acyclic, one can check whether the equation

$$\sum_{(u,v) \in MSF(Q)} w(u, v) = \sum_{x \in \text{Var}(Q)} (\text{Class}(x) - 1) \tag{5}$$

holds, where $\text{Class}(x)$ denotes the number of literals in Q that contain x (see also [4]).

Learning Logic Programs with Unary Partial Function Graph Background Knowledge (extended abstract)

Tamás Horváth[1⋆], Robert H. Sloan[2], and György Turán[3]

[1] Institute of Computer Science III, University of Bonn and Fraunhofer Institute for Autonomous intelligent Systems, tamas.horvath@ais.fhg.de
[2] Department of Computer Science, University of Illinois at Chicago, sloan@uic.edu
[3] Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago and Research Group on Artificial Intelligence, Hungarian Academy of Sciences, Szeged, gyt@uic.edu

Abstract. The product homomorphism method is a combinatorial tool that can be used to develop polynomial PAC-learning algorithms in predicate logic. Using the product homomorphism method, we show that a single nonrecursive definite Horn clause is polynomially PAC-learnable if the background knowledge is a function-free extensional database over a single binary predicate and the ground atoms in the background knowledge form a unary partial function. That is, the background knowledge corresponds to a directed graph, where each node has outdegree at most 1. The proof is based on a detailed analysis of products and homomorphisms of the class of digraphs corresponding to unary partial functions.

1 Introduction

Attribute-value languages are often not suitable for representing complex real-world machine learning problems. Therefore, one of the research challenges in machine learning is to study learning in other representation languages. Among such approaches, inductive logic programming (ILP) [8] is concerned with learning in predicate logic, in particular, with learning logic programs.

The general ILP learning problem is computationally intractable. Therefore, one of the challenging problems in ILP is to show positive and negative theoretical results about the efficient learnability of different fragments of predicate logic in the formal models of computational learning theory. Most of such positive results have been obtained by restricting the hypothesis language. In particular, the most frequently used restrictions are *determinateness* and learning with *constant depth bound* [5].

In contrast to these approaches, in this work we present a positive learnability result by restricting the background knowledge. We assume that there is a single binary background predicate R, and that the ground R-atoms in the background knowledge represent a unary partial function. This structural assumption implies that the background knowledge corresponds to a digraph where each vertex has outdegree at most

⋆ Partially supported by the DFG project (WR 40/2-1) *Hybride Methoden und Systemarchitekturen für heterogene Informationsräume*.

1. To prove polynomial learnability for such family of learning problems in the PAC model of learning [9], we use the product homomorphism method [4], a general combinatorial method specific to deriving polynomial learning algorithms in predicate logic. The method is based on finding a combinatorial characterization for the existence of a certain homomorphism from products of relational structures. From the structural assumption on the background knowledge it follows that we have to study products and homomorphisms related to unary partial function graphs.

Using the product homomorphism method, we obtained positive PAC result for the cases when the ground atoms in the background knowledge form a forest [4] or a unary function graph [3] (i.e., when each vertex has outdegree 1). The result of this paper generalizes these results, as unary partial function graphs include both cases; a connected component of a unary partial function graph is always either a tree or a function graph consisting of a single connected component. Though the structural difference between unary function and unary partial function graphs may seem to be insignificant, it turns out that the presence of both types of components requires a careful revision of the results in [3].

The paper is organized as follows. In Section 2, we first give the necessary concepts related to unary partial function graphs, and in Section 3 we then formulate our learning problem. In Section 4, we briefly describe the product homomorphism method, and in Section 5, we derive a polynomial PAC-learning algorithm by using the product homomorphism method. Finally, in Section 6, we give some concluding remarks along with some open problems. Due to space limitation, we omit the proofs in this extended abstract.

2 Graphs and unary partial function graphs

We assume the reader is familiar with the basic concepts of graph theory (see, e.g., [2]). Throughout this paper, by graphs we always mean directed graphs. For a graph G, we denote by $V(G)$ (resp. $E(G)$) the set of vertices (resp. edges) of G.

Let G_i be a graph for $1 \leq i \leq t$. The *product* $G = \prod_{i=1}^{t} G_i$ is a graph with $V(G) = \prod_{i=1}^{t} V(G_i)$ such that for all $\vec{u} = (u_1, \ldots, u_t)$, $\vec{v} = (v_1, \ldots, v_t) \in V(G)$ it holds that $(\vec{u}, \vec{v}) \in E(G)$ iff $(u_i, v_i) \in E(G_i)$ for every $i = 1, \ldots, t$. The t-th power of G, denoted G^t, is the product of t copies of G.

A *homomorphism* from a graph G_1 to a graph G_2 is a map $\varphi : V(G_1) \to V(G_2)$ such that $(\varphi(u), \varphi(v)) \in E(G_2)$ whenever $(u, v) \in E(G_1)$. We call a homomorphism *singly rooted* if we specify the image of one vertex in $V(G_1)$ in advance, and we call a homomorphism *multiply rooted* if we specify images of multiple vertices in $V(G_1)$ in advance. A homomorphism from G_1 to G_2 mapping u_i to v_i for $i = 1, \ldots, k$ is denoted by

$$G_1 \xrightarrow[\{u_1/v_1, \ldots, u_k/v_k\}]{} G_2 \ .$$

We note that a homomorphism always maps one connected component into one connected component.

2.1 Unary partial function graphs

A graph G is a *unary partial function graph* if every vertex of G has outdegree at most 1. The name is justified by viewing G as a graph representing a *unary partial function* $f : V(G) \hookrightarrow V(G)$ such that $f(u) = v$ iff $(u, v) \in E(G)$ for every $u, v \in V(G)$. As an example, the graph given in Fig. 1 is a unary partial function graph consisting of three connected components. Except a_5, each vertex has outdegree 1.

Fig. 1. The directed graph representing the unary function f.

For the rest of this section, let G denote a unary partial function graph consisting of a single connected component. Then it holds that G has at most one vertex of outdegree 0. If G has such a vertex then G is *acyclic*, as in this case it is a directed tree such that the edges are directed towards the root (which is the vertex of outdegree 0). Otherwise (i.e., when each of the vertices of the connected component has outdegree 1) G is *cyclic* and it may be viewed as a directed cycle with directed trees "hanging" from some vertices of the cycle. The edges of the trees are directed towards the cycle. We note that the directed cycle may be a *loop* (i.e., a cycle may have length 1). *Cyclic* vertices are those on the cycle (e.g., a_{15} on Fig. 1). The other vertices are called *noncyclic*.

For a vertex $v \in V(G)$, we denote by $f(v)$ the successor of v. We define $f^{(0)}(v) = v$ and $f^{(k)}(v)$ denotes $f(f^{(k-1)}(v))$ for every $k > 0$. Note that $f^{(k)}(v)$ may be undefined. For instance, for the graph on Fig. 1 it holds that $f^{(2)}(a_3) = a_5$ and $f^{(3)}(a_3)$ is undefined.

Let $k \geq 0$ be an integer. We define $h(f^{(k)}(v))$, the *height* of $f^{(k)}(v)$, by

$$h(f^{(k)}(v)) = \begin{cases} \max\{d : f^{(k)}(v) = f^{(d)}(u) \text{ for some } u \in V(G)\} & \text{if } f^{(k)}(v) \text{ is noncyclic} \\ \infty & \text{if } f^{(k)}(v) \text{ is cyclic} \\ 0 & \text{otherwise .} \end{cases}$$

For the graph on Fig. 1, we have $h(f^{(2)}(a_3)) = h(a_5) = 4$, $h(f^{(3)}(a_3)) = 0$, and $h(a_{15}) = \infty$.

If G is cyclic then $\delta(v)$ denotes the length of the unique directed path connecting v to the cycle, otherwise (i.e., if G is a tree) it denotes the length of the unique directed path connecting v to the root. In both cases, the other endpoint $f^{(\delta(v))}(v)$ of the path is denoted by $r(v)$ and is referred to as the *root* of v. In our example on Fig. 1, $\delta(a_3) = 2$, as $r(a_3) = a_5$, and $\delta(a_{11}) = 2$, as $r(a_{11}) = a_{13}$.

If G is cyclic then we denote by $L_{\text{cycle}}(G)$ the length of its cycle; if G is a tree then $L_{\text{cycle}}(G) = 0$. If v is a vertex of a general unary partial function graph (i.e., one which may consist of more than one connected component) then $L_{\text{cycle}}(v)$ denotes $L_{\text{cycle}}(G')$, where G' is the connected component containing v. In Fig. 1, $L_{\text{cycle}}(a_{11}) = 3$.

Let $u, v \in V(G)$ such that $u = r(u)$ and $v = r(v)$. Then $\sigma(u, v)$ is the smallest nonnegative integer d satisfying $f^{(d)}(u) = v$. Note that by the definition of u and v, both of them are either cyclic or noncyclic. In the first case, $\sigma(u, v)$ denotes the length of the (smallest) directed path leading from u to v on the cycle of G. In the second case, both u and v must be roots of a tree. Since G consists of a single connected component, $u = v$ and hence, $\sigma(u, v) = 0$ always holds for this case.

Now let $u, v \in V(G)$ such that $r(u) = r(v)$. Then there is a unique maximal integer d, $0 \leq d \leq \min\{\delta(u), \delta(v)\}$, such that $f^{(\delta(u)-d)}(u) = f^{(\delta(v)-d)}(v)$. This node is called the *least common ancestor* of u and v, and is denoted by $\text{lca}(u, v)$.

We are ready to define the *distance* between two vertices. Let $u, v \in V(G)$. Then their distance is an ordered pair of nonnegative integers defined by

$$d_{pf}(u, v) = \begin{cases} (d_1, d_2) \text{ such that } f^{(d_1)}(u) = \text{lca}(u, v) = f^{(d_2)}(v) & \text{if } r(u) = r(v) \\ (\delta(u) + \sigma(r(u), r(v)), \delta(v)) & \text{otherwise .} \end{cases}$$

On Fig. 1, $d_{pf}(a_3, a_8) = (1, 2)$, as $\text{lca}(a_3, a_8) = a_4$, and $d_{pf}(a_{11}, a_{16}) = (2 + 1, 1)$.

In the following proposition we formulate some properties of products of unary partial function graphs, that will be used many times in what follows.

Proposition 1. *Let G_i be unary partial function graphs and $b_i \in V(G_i)$ for $i = 1, \ldots, t$. Let $G = \prod_{i=1}^{t} G_i$ be the product of the G_i's and $\vec{b} = (b_1, \ldots, b_t)$ be the product vertex obtained from the b_i's. Then for G and $\vec{b}$ the following properties hold.*

(i) *G is a unary partial function graph.*
(ii) *$\vec{b}$ is cyclic iff all the b_i's are cyclic.*
(iii) *If $\vec{b}$ is cyclic then $L_{cycle}(\vec{b}) = \text{lcm}(L_{cycle}(b_1), \ldots, L_{cycle}(b_t))$, where $\text{lcm}(n_1, \ldots, n_t)$ denotes the least common multiple of $n_1, \ldots, n_t$.*

3 Learning simple logic programs

In this section we define a special class of logic programs [7] that will be discussed from the point of view of learnability.

3.1 Simple logic programs

Throughout this paper we consider (relational) *vocabularies* consisting of a *target predicate* P of arity m, a binary *background predicate* R, and constants $a_1, \ldots, a_n$. Thus, a *term* is either a variable or a constant, and an *atom* is of the form $P(t_1, \ldots, t_m)$ or $R(t_1, t_2)$, where the t's are terms. Depending on its predicate symbol, an atom is said to be a P-*atom* or an R-*atom*. A *literal* is an atom or its negation. An atom is *ground* if it contains no variables. A *basic clause* is a first-order Horn clause of the form $L_0 \leftarrow L_1, \ldots, L_l$ where L_0 is a P-atom and L_i is an R-atom for $i = 1, \ldots, l$. It is also viewed as the set of literals it contains. A *simple logic program* consists of a basic clause and a set $\mathcal{B}$ of ground R-atoms. Since R is binary, the ground R-atoms in $\mathcal{B}$ form a directed graph with vertices $a_1, \ldots, a_n$.

A *substitution* $\theta = \{x_1/t_1, \ldots, x_s/t_s\}$ is a mapping of variables to terms such that $x_i \neq t_i$ for $i = 1, \ldots, s$. Let W be a literal (respectively a clause). Then $W\theta$ is the literal (respectively clause) obtained from W by rewriting simultaneously each variable x_i to t_i in W for $i = 1, \ldots, s$. A clause C *subsumes* a clause D, denoted $C \leq_\theta D$, if there exists a substitution θ such that $C\theta \subseteq D$.

To close this subsection, let C be a basic clause, $\mathcal{B}$ be a set of ground R-atoms, and A be a ground P-atom. We say that C *subsumes* A *with respect to* $\mathcal{B}$, denoted $C \leq_{\theta,\mathcal{B}} A$, if C subsumes the basic clause $A \leftarrow \mathcal{B}$, i.e., $C \leq_\theta (A \leftarrow \mathcal{B})$. It holds that $C \leq_{\theta,\mathcal{B}} A$ iff A is *implied* by the simple logic program consisting of C and $\mathcal{B}$.

3.2 The learning problem

In this section we give a formal description of the family of learning problems considered in this paper. We assume that the reader is familiar with the basic notions of the PAC-model of learning [9].

Let $\mathcal{B}$ be a set of ground R-atoms. In what follows, $\mathcal{B}$ is referred to as *background knowledge*, and its elements are called *background atoms*. As R is a binary predicate, $\mathcal{B}$ can be viewed as a graph with vertices $a_1, \ldots, a_n$.

The *instance space* of the learning problem is the set of all ground P-atoms. Let C be a basic clause. Then the *concept* $C_\mathcal{B}$ represented by C wrt. $\mathcal{B}$ is the set of ground P-atoms implied by the simple logic program consisting of C and $\mathcal{B}$, i.e.,

$$C_\mathcal{B} = \{A : A \text{ is a ground } P\text{-atom and } C \leq_{\theta,\mathcal{B}} A\} \ .$$

The *concept class* $\mathcal{C}_{\mathcal{B},m}$, corresponding to $\mathcal{B}$ is the family of concepts $C_\mathcal{B}$, where C is a basic clause. (For the next definition, we recall that m is the arity of the target predicate P.) Throughout this paper, we consider the family $\mathcal{F}_{m,n}$ $(m, n > 0)$ of learning problems defined by

$$\mathcal{F}_{m,n} = \{\mathcal{C}_{\mathcal{B},m} : \mathcal{B} \text{ corresponds to a unary partial function graph over } n \text{ vertices}\} \ .$$

That is, a concept class $\mathcal{C}_{\mathcal{B},m}$ belongs to $\mathcal{F}_{m,n}$ iff for every a_i there is at most one a_j such that $R(a_i, a_j) \in \mathcal{B}$ $(1 \leq i, j \leq n)$. The *parameters* measuring the size of a learning problem in $\mathcal{F}_{m,n}$ are m and n.[1]

[1] In Section 5.4, we shall show that the size of the target basic clause as parameter can be omitted by extending the standard representation language.

4 The product homomorphism method

In order to prove polynomial PAC-learnability for $\mathcal{F}_{m,n}$, we shall apply the following basic result [1] from computational learning theory.

Theorem 1. *A family of learning problems is polynomially PAC-learnable if*

(i) the hypothesis finding task can be solved in time polynomial in the parameters,
(ii) the VC-dimension of the concept classes is bounded by a polynomial of the parameters.

According to the first step of the above theorem, we have to show that the hypothesis finding problem for the concept classes in $\mathcal{F}_{m,n}$ can be solved in time polynomial in m and n. More precisely, we consider the following *single clause hypothesis finding problem*: *Given* $\mathcal{C}_{\mathcal{B},m} \in \mathcal{F}_{m,n}$ and disjoint sets E^+ and E^- of ground P-atoms, *find* a basic clause C such that $E^+ \subseteq C_{\mathcal{B}}$ and $E^- \cap C_{\mathcal{B}} = \emptyset$, if such a basic clause exists, and *output* "no", otherwise.

In [4], we have shown that $\mathcal{C}_{\mathcal{B},m}$ is *closed under nonempty intersection* for every $\mathcal{C}_{\mathcal{B},m} \in \mathcal{F}_{m,n}$. That is, for every subset $\mathcal{C} \subseteq \mathcal{C}_{\mathcal{B},m}$ satisfying $\bigcap_{c\in\mathcal{C}} c \neq \emptyset$ it holds that $\bigcap_{c\in\mathcal{C}} c \in \mathcal{C}_{\mathcal{B},m}$. This implies that for a set S of ground P-atoms, the intersection of all concepts containing S, denoted $G_{\mathcal{B}}(S)$, is also a concept in $\mathcal{C}_{\mathcal{B},m}$, i.e., $G_{\mathcal{B}}(S) \in \mathcal{C}_{\mathcal{B},m}$, where

$$G_{\mathcal{B}}(S) = \bigcap \{C_{\mathcal{B}} \in \mathcal{C}_{\mathcal{B},m} : S \subseteq C_{\mathcal{B}}\} \ .$$

In other words, $G_{\mathcal{B}}(S)$, also referred to as the *concept generated by* S, is the smallest concept in $\mathcal{C}_{\mathcal{B},m}$ that contains S. But this means that a consistent clause for the above defined single clause hypothesis finding problem exists iff $G_{\mathcal{B}}(S)$ and E^- are disjoint. Thus, the single clause hypothesis finding problem can be solved by computing first an *efficiently evaluable* basic clause representing the concept $G_{\mathcal{B}}(E^+)$ and then testing whether $G_{\mathcal{B}}(E^+) \cap E^- = \emptyset$ holds.

The following theorem, a special case of the product homomorphism theorem in [4], gives a combinatorial characterization of the concept generated by a set of ground P-atoms.

Theorem 2. *Let* $\mathcal{C}_{\mathcal{B},m} \in \mathcal{F}_{m,n}$, $S = \{P(b_{1,1}, \ldots, b_{1,m}), \ldots, P(b_{t,1}, \ldots, b_{t,m})\}$ *for some* $t > 0$, *and let* $\vec{b}_j$ *denote* $(b_{1,j}, \ldots, b_{t,j})$ *for* $j = 1, \ldots, m$. *Then*

$$G_{\mathcal{B}}(S) = \left\{ P(b_1, \ldots, b_m) : \mathcal{B}^t \xrightarrow[\{\vec{b}_1/b_1,\ldots,\vec{b}_m/b_m,\vec{a}_1/a_1,\ldots,\vec{a}_n/a_n\}]{} \mathcal{B} \right\} \ ,$$

where $\vec{a}_k$ *denotes the (*t*-tuple) product constants* $(a_k, \ldots, a_k)$ *for* $k = 1, \ldots, n$.

Theorem 2 above provides the following method, called the *product homomorphism method* [4], for obtaining a hypothesis finding algorithm for $\mathcal{F}_{m,n}$:

1. Find a combinatorial characterization for the existence of multiply rooted homomorphisms from products of unary partial function graphs to unary partial function graphs.

2. Give an algorithm such that for every $\mathcal{C}_{\mathcal{B},m} \in \mathcal{F}_{m,n}$ and for every set S of ground P-atoms it translates the combinatorial characterization in time polynomial in m, n, and $|S|$ into a basic clause C such that
   - $C_{\mathcal{B}} = G_{\mathcal{B}}(S)$ and
   - C can be evaluated with respect to $\mathcal{B}$ in time polynomial in m and n.

5 Application of the product homomorphism method

Using the product homomorphism method, in this section we derive an efficient PAC algorithm for learning simple logic programs with partial function graph background knowledge.

5.1 Homomorphisms between unary partial function graphs

In order to apply the product homomorphism method to unary partial function graphs, we first need to find necessary and sufficient conditions for the existence of multiply rooted homomorphisms from products of unary partial function graphs into unary partial function graphs. Since unary partial function graphs are closed under product by (i) of Proposition 1, in the next theorems we study rooted homomorphisms between unary partial function graphs. Furthermore, as a homomorphism always maps one connected component into one connected component, it is sufficient to consider unary partial function graphs consisting of a single connected component.

Theorem 3. *Let G_1 and G_2 be unary partial function graphs consisting of a single connected component, let $b_1, \ldots, b_k \in V(G_1)$ be distinct vertices for some $k \geq 1$, and $c_1, \ldots, c_k \in V(G_2)$. Then $G_1 \xrightarrow[\{b_1/c_1,\ldots,b_k/c_k\}]{} G_2$ iff*

(i) $G_1 \xrightarrow[\{b_i/c_i\}]{} G_2$ *for every* $i = 1, \ldots, k$,

(ii) $f^{(d_1)}(c_u) = f^{(d_2)}(c_v)$ *for every* $1 \leq u < v \leq k$, *where* $(d_1, d_2) = d_{pf}(b_u, b_v)$.

Condition (i) of the above theorem indicates that one has to study *singly* rooted homomorphisms between unary partial function graphs. The following theorem gives a necessary and sufficient condition for the existence of a singly rooted homomorphism between unary partial function graphs. We denote by $n_1 \mid n_2$ that n_1 divides n_2.

Theorem 4. *Let G_1 and G_2 be unary partial function graphs consisting of a single connected component and let $b \in V(G_1)$, $c \in V(G_2)$. Then $G_1 \xrightarrow[\{b/c\}]{} G_2$ iff*

(i) G_2 *is cyclic satisfying* $L_{cycle}(G_2) \mid L_{cycle}(G_1)$ *whenever* G_1 *is cyclic,*

(ii) $h(f^{(k)}(c)) \geq h(f^{(k)}(b))$ *for every* $k \geq 0$.

5.2 Products of unary partial function graphs

The product homomorphism method indicates that the learning algorithm must consider the product of t copies of the graph representing the background knowledge, where t is the number of positive examples. However, that product is exponentially large; it contains n^t nodes, where n is the number of constants mentioned in the background knowledge. Therefore we cannot work with this graph explicitly. Instead, we must show that the relevant parameters implied by Theorems 3 and 4, i.e., cycle lengths, heights, and distances between vertices can be computed directly from those of the original graph corresponding to the background knowledge.

For computing cycle lengths, we can directly apply (iii) of Proposition 1. We start by giving a lemma that can be used for determining the height of a product vertex.

Lemma 1. *Let G_i be unary partial function graphs for $i = 1, \ldots, t$, $G = \prod_{i=1}^{t} G_i$, and $\vec{b} = (b_1, \ldots, b_t) \in V(G)$. Then $h(f^{(k)}(\vec{b})) = \min_{i=1,\ldots,t} h(f^{(k)}(b_i))$ for every $k \geq 0$.*

To state Lemma 4 below for computing the distance between two product vertices, in the following lemma we first characterize the distance of a product vertex from its root. Then, in Lemma 3, we give a necessary and sufficient condition for two vertices of the product graph to be in the same connected component.

Lemma 2. *Let G_i be unary partial function graphs, $b_i \in V(G_i)$ for $i = 1, \ldots, t$, and consider the product graph $G = \prod_{i=1}^{t} G_i$ and product vertex $\vec{b} = (b_1, \ldots, b_t) \in V(G)$. Let I denote the set of indices $\{i : 1 \leq i \leq t,\ r(b_i)$ is noncyclic$\}$. Then the distance of $\vec{b}$ from its root $r(\vec{b})$ is given by*

$$\delta(\vec{b}) = \begin{cases} \max_{i=1,\ldots,t} \delta(b_i) & \text{if } I = \emptyset \\ \min_{i \in I} \delta(b_i) & \text{otherwise .} \end{cases}$$

Lemma 3. *Let G_i be unary partial function graphs for $i = 1, \ldots, t$, $G = \prod_{i=1}^{t} G_i$, and $\vec{b} = (b_1, \ldots, b_t)$ $\vec{c} = (c_1, \ldots, c_t) \in V(G)$. Then $\vec{b}$ and $\vec{c}$ are in the same connected component of G iff for every $i = 1, \ldots, t$ it holds that $f^{(\delta(\vec{b})+d)}(b_i) = f^{(\delta(\vec{c}))}(c_i)$, where $d = 0$ if some of the b_i's belongs to a noncyclic connected component; otherwise, d is a nonnegative integer satisfying*

$$d \equiv \sigma(f^{(\delta(\vec{b}))}(b_k), f^{(\delta(\vec{c}))}(c_k)) \pmod{L_{cycle}(b_k)}$$

for every $k = 1, \ldots, t$.

The following lemma follows directly from Lemmas 2 and 3.

Lemma 4. *Let G_i, G, b_i, and c_i, $1 \leq i \leq t$, be defined as in the previous lemma and assume that the product vertices $\vec{b} = (b_1, \ldots, b_t)$ and $\vec{c} = (c_1, \ldots, c_t)$ belong to the same connected component of G. Then*

$$d_{pf}(\vec{b}, \vec{c}) = \begin{cases} (\delta(\vec{b}) - d_1, \delta(\vec{c}) - d_1) & \text{if } r(\vec{b}) = r(\vec{c}) \\ (\delta(\vec{b}) + d_2, \delta(\vec{c})) & \text{otherwise ,} \end{cases}$$

where $d_1 = \max\{d \geq 0 : f^{(\delta(\vec{b})-d)}(b_i) = f^{(\delta(\vec{c})-d)}(c_i)$ *for* $i = 1, \ldots, t\}$ *and* d_2 *is the smallest nonnegative integer satisfying*

$$d_2 \equiv \sigma(f^{(\delta(\vec{b}))}(b_i), f^{(\delta(\vec{c}))}(c_i)) \pmod{l_i}$$

for every $i = 1, \ldots, t$.

Since the congruence system in Lemmas 3 and 4 can be solved efficiently (see, e.g., [6]), one can decide efficiently whether $\vec{b}$ and $\vec{c}$ are in the same connected component, and if so, then their distance $d_{pf}(\vec{b}, \vec{c})$ can be computed in polynomial time.

5.3 A combinatorial characterization of $G_{\mathcal{B}}$

Combining the results of Sections 5.1 and 5.2 with Theorem 2, in this section we give a combinatorial characterization of the concept generated by a set of ground P-atoms wrt. unary partial function graph background knowledge. Let S be the set of ground atoms $\{P(\mathbf{b}_1), \ldots, P(\mathbf{b}_t)\}$, where $\mathbf{b}_i = (b_{i,1}, \ldots, b_{i,m})$ for $i = 1, \ldots, t$ ($t > 1$). Let $\vec{b}_j$ denote the product vertex $(b_{1,j}, \ldots, b_{t,j})$ for $j = 1, \ldots, m$. Let

$$\begin{aligned}
I_{\text{const}}(S) &= \{(j,q) : 1 \leq j \leq m, 1 \leq q \leq n, \text{ and } \vec{b}_j = \vec{a}_q\} \ , \\
I_{\text{var}}(S) &= \{j : 1 \leq j \leq m, \vec{b}_j \neq \vec{a}_q \text{ for some } q, 1 \leq q \leq n\} \ , \\
I_{\text{cyclic}}(S) &= \{j \in I_{\text{var}}(S) : r(\vec{b}_j) \text{ is cyclic}\} \ , \\
H_{\text{pairs}}(S) &= \{(u,v) : u, v \in I_{\text{var}}(S), u < v, \text{ and } \vec{b}_u, \vec{b}_v \text{ are in the same connected component}\} \ , \\
H_{\text{const}}(S) &= \{(j,q) : j \in I_{\text{var}}(S), 1 \leq q \leq n, \text{ and } \vec{b}_j, \vec{a}_q \text{ are in the same connected component}\} \ .
\end{aligned}$$

Theorem 5. *If* $\mathcal{B}$ *is a unary partial function graph then*

$$\begin{aligned}
G_{\mathcal{B}}(S) = \{P(b_1, \ldots, b_m) :& \\
& b_j = a_q \text{ for all } (j,q) \in I_{const}(S), && (1)\\
& f^{(L_{cycle}(\vec{b}_j))}(r(b_j)) = r(b_j) \text{ for all } j \in I_{cyclic}(S), && (2)\\
& h(f^{(k)}(b_j)) \geq h(f^{(k)}(\vec{b}_j)) \text{ for all } k = 0, \ldots, n \text{ and } j \in I_{var}(S), && (3)\\
& f^{(d_1)}(b_u) = f^{(d_2)}(b_v) \text{ for all } (u,v) \in H_{pairs}(S), \text{ where } (d_1, d_2) = d_{pf}(\vec{b}_u, \vec{b}_v), && (4)\\
& f^{(d_1)}(b_j) = f^{(d_2)}(a_q) \text{ for all } (j,q) \in H_{const}(S), \text{ where } (d_1, d_2) = d_{pf}(\vec{b}_j, \vec{a}_q)\} && (5)
\end{aligned}$$

Proof sketch. By Theorem 2, $P(b_1, \ldots, b_m) \in G_{\mathcal{B}}(S)$ iff

$$(\mathcal{B})^t \xrightarrow[\{\vec{b}_1/b_1, \ldots, \vec{b}_m/b_m, \vec{a}_1/a_1, \ldots, \vec{a}_n/a_n\}]{} \mathcal{B} \ . \qquad (6)$$

Thus, it is sufficient to show that (6) holds iff conditions (1) - (5) hold. For the "only if" part, the proof of (1) is automatic, (2), (3) hold by Theorem 4, and (4), (5) by (ii) of Theorem 3. To prove the "if" part, the connected components of $(\mathcal{B})^t$ can be considered separately.

(i) For connected components not containing any non-constant product vertex from $\vec{b}_1, \ldots, \vec{b}_m$, there are projections providing a homomorphism into $\mathcal{B}$ and mapping each occurring product constant $\vec{a}$ to a.

(ii) By Theorems 3 and 4, (2)–(4) provide the required rooted homomorphisms for connected components containing at least one non-constant vertex from $\vec{b}_1, \ldots, \vec{b}_m$.

(iii) Finally, for connected components containing at least one non-constant vertex from $\vec{b}_1, \ldots, \vec{b}_m$ and at least one constant product vertex, (2)–(3) and (5) provide the required rooted homomorphisms by Theorems 3 and 4. □

5.4 Concept representation and polynomial learnability

The last step of the product homomorphism method is to give an algorithm translating the combinatorial characterization given by Theorem 5 into an efficiently evaluable basic clause. From our previous results [4] on the length of product cycles it follows that there are cases when the size of any consistent basic clause is exponential in n, i.e., when $L_{\text{cycle}}(\vec{b}_j)$ in (2) is exponential in n. Thus, the standard representation using only the predicates P and R is not suitable for polynomial learnability if the size of the target concept is not considered as a learning parameter. Therefore, we introduce new predicates of the form $\text{PATH}_d(x, y)$, which hold if there is a path of length d from x to y, for every d. Note that $\text{PATH}_d(x, x)$ holds iff $d = 0$ or $d > 0$ and $L_{\text{cycle}}(x) \mid d$. Using the extended representation language, we are ready to give Algorithm 1 computing a clause that represents $G_{\mathcal{B}}(S)$ for a set S of ground P-atoms and unary partial function graph background knowledge $\mathcal{B}$.

Algorithm 1 UNARYPARTIALFUNCTIONGRAPH

Require: ground set $S = \{P(\mathbf{b}_1), \ldots, P(\mathbf{b}_t)\}$ and a unary partial function graph $\mathcal{B}$
Ensure: clause C such that $C_{\mathcal{B}} = G_{\mathcal{B}}(S)$

1: **let** $C = \{P(t_1, \ldots, t_m)\}$, where $t_j = a_q$ if $(j, q) \in I_{\text{const}}(S)$ for some q, otherwise t_j is the variable $x_{\vec{b}_j}$ for $j = 1, \ldots, m$
2: **for all** $j \in I_{\text{cyclic}}(S)$ **do**
  if $\delta(\vec{b}_j) = 0$ **then** $C = C \cup \{\neg\text{PATH}_{L_{\text{cycle}}(\vec{b}_j)}(t_j, t_j)\}$
  else $C = C \cup \{\neg\text{PATH}_{\delta(\vec{b}_j)}(t_j, y_j), \neg\text{PATH}_{L_{\text{cycle}}(\vec{b}_j)}(y_j, y_j)\}$
3: **for all** $j \in I_{\text{var}}(S)$ and $k = 0, \ldots, \delta(\vec{b}_j)$ **do**
  if $k < h(f^{(k)}(\delta(\vec{b}_j))) < \infty$ **then** $C = C \cup \{\neg\text{PATH}_k(t_j, y'_j), \neg\text{PATH}_{h(f^{(k)}(\vec{b}_j))}(y''_j, y'_j)\}$
4: **for all** $(u, v) \in H_{\text{pairs}}(S)$ **do**
  $C = C \cup \{\neg\text{PATH}_{d_1}(t_u, z_{u,v}), \neg\text{PATH}_{d_2}(t_v, z_{u,v})\}$ where $(d_1, d_2) = d_{pf}(\vec{b}_u, \vec{b}_v)$
5: **for all** $(j, q) \in H_{\text{const}}(S)$ **do**
  $C = C \cup \{\neg\text{PATH}_{d_1}(t_j, z'_{j,q}), \neg\text{PATH}_{d_2}(a_q, z'_{j,q})\}$ where $(d_1, d_2) = d_{pf}(\vec{b}_j, \vec{a}_q)$
6: **return** C

Steps (1)–(5) of Algorithm 1 translate Conditions (1)–(5) of Theorem 5, respectively. We have the following theorem.

Theorem 6. *Algorithm 1 is correct, i.e., $C_{\mathcal{B}} = G_{\mathcal{B}}(S)$, and it is polynomial in m, n, and $|S|$. The size of C (in the extended representation language) is $O\left(m^2 + mn\right)$. C can be evaluated wrt. $\mathcal{B}$ in time polynomial in m and n.*

Example 1. To illustrate Algorithm 1, consider the unary partial function graph G given in Fig. 1. Using the binary background predicate R, let $\mathcal{B}$ be the corresponding background knowledge, i.e., $R(u, v) \in \mathcal{B}$ iff $(u, v) \in E(G)$ for every $u, v \in V(G)$.[2]

Let P be a ternary target predicate and $S = \{P(a_6, a_3, a_{12}), P(a_{11}, a_9, a_{18})\}$. Thus, $\vec{b}_1 = (a_6, a_{11})$, $\vec{b}_2 = (a_3, a_9)$, and $\vec{b}_3 = (a_{12}, a_{18})$. By definition, $I_{\text{const}}(S) = \emptyset$, $I_{\text{var}}(S) = \{1, 2, 3\}$, and by (ii) of Proposition 1, $I_{\text{cyclic}}(S) = \{3\}$. To compute $H_{\text{pairs}}(S)$, we apply Lemma 3 for every (u, v), $u, v \in I_{\text{var}}(S)$. For $(1, 2)$ corresponding to $(\vec{b}_1, \vec{b}_2) = ((a_6, a_{11}), (a_3, a_9))$, we have to check whether

$$f^{(\delta(\vec{b}_1)+d)}(a_6) = f^{(\delta(\vec{b}_2))}(a_3) \text{ and } f^{(\delta(\vec{b}_1)+d)}(a_{11}) = f^{(\delta(\vec{b}_2))}(a_9) \tag{7}$$

hold for $d = 0$. By Lemma 2, we have

$$\begin{aligned} \delta(\vec{b}_1) = \delta((a_6, a_{11})) &= \min_{i \in \{1\}} \delta(b_i) = \delta(a_6) \quad = 1 \\ \delta(\vec{b}_2) = \delta((a_3, a_9)) &= \min_{i \in \{1\}} \delta(b_i) = \delta(a_3) = 2 \ . \end{aligned}$$

In both equations, $I = \{1\}$, as $r(a_{11})$ and $r(a_9)$ are cyclic. The equations in (7) thus hold by

$$f^{(1)}(a_6) = a_5 = f^{(2)}(a_3) \text{ and } f^{(1)}(a_{11}) = a_{12} = f^{(2)}(a_9)$$

respectively. Hence, $(1, 2) \in H_{\text{pairs}}(S)$. It can be shown in the same way that $H_{\text{pairs}}(S) = \{(1, 2)\}$ and $H_{\text{const}}(S) = \emptyset$.

Now we are ready to illustrate Algorithm 1 step by step on inputs S and $\mathcal{B}$.

Step 1: $C = \{P(x_{\vec{b}_1}, x_{\vec{b}_2}, x_{\vec{b}_3})\}$ because $I_{\text{const}} = \emptyset$.

Step 2: Since $I_{\text{cyclic}}(S) = \{3\}$, we have to compute $\delta(\vec{b}_3)$ and $L_{\text{cycle}}(\vec{b}_3)$. By Lemma 2, $\delta(\vec{b}_3) = 0$, and from (iii) of Proposition 1 we have $L_{\text{cycle}}(\vec{b}_3) = \text{lcm}\{3, 4\} = 12$. Thus, in this step we add the literal $\neg\textsc{Path}_{12}(x_{\vec{b}_3}, x_{\vec{b}_3})$ to C.

Step 3: For $j = 1$ and $k = 0$ we first have to compute $h(f^{(0)}(\vec{b}_1))$. By Lemma 1, we have $h(f^{(0)}(\vec{b}_1)) = \min\{h(f^{(0)}(a_6)), h(f^{(0)}(a_{11}))\} = \min\{0, 1\} = 0$. For $j = 1$ and $k = \delta(\vec{b}_1) = 1$, $h(f^{(1)}(\vec{b}_1)) = 1$ by Lemma 1. Since $h(f^{(1)}(\vec{b}_1)) \not> 1$, we add no literals to C. For similar reasons, we add no new literals to C for $j = 2, 3$.

Step 4: As $H_{\text{pairs}} = \{(1, 2)\}$, we have $d_{pf}(\vec{b}_1, \vec{b}_2) = (1, 2)$ by Lemma 4, and add therefore $\neg\textsc{Path}_1(x_{\vec{b}_1}, z_{1,2}), \neg\textsc{Path}_2(x_{\vec{b}_2}, z_{1,2})$ to C.

Step 5: $H_{\text{const}}(S) = \emptyset$ and thus, no literals will be added to C in this step.

Step 6: The algorithm finally returns the clause

$$P(x_{\vec{b}_1}, x_{\vec{b}_2}, x_{\vec{b}_3}) \leftarrow \textsc{Path}_{12}(x_{\vec{b}_3}, x_{\vec{b}_3}), \textsc{Path}_1(x_{\vec{b}_1}, z_{1,2}), \textsc{Path}_2(x_{\vec{b}_2}, z_{1,2}) \ .$$

[2] We note that B is obtained from the background knowledge in the running example of [3] by removing edge (a_5, a_4).

Since the VC-dimension of a concept class $\mathcal{C}$ is at most $\log(\mathcal{C})$, from the bound on the size of the target concept in Theorem 6 it follows that the VC-dimension of $\mathcal{C}_{\mathcal{B},m}$ is polynomial in m and n if $\mathcal{B}$ is a unary partial function graph. Thus, by Theorems 1 and 6 we have the following main result of this paper.

Theorem 7. *Using the extended representation language, simple logic programs with $\mathcal{B}$ being a unary partial function graph are efficiently PAC-learnable.*

6 Remarks and open problems

Using the product homomorphism method, we have shown that simple logic programs with unary partial function graph background knowledge are polynomially PAC-learnable in an extended representation language. The importance of this result is that we have not assumed any bound on the size of the target clause, the target clause is not necessarily determinate, and its size may be exponential in the standard representation language.

Finally, we list some interesting open problems for further research. In practical applications, usually there is no consistent hypothesis consisting of a single clause. Since unary partial function graphs generalize forests, from our previous results in [4] it follows that the problem of deciding whether there exists a consistent hypothesis consisting of k clauses is NP-complete for any fixed $k \geq 3$ for unary partial function graph background knowledge. It would be interesting to see whether the optimal solution can be approximated in polynomial time. A further research topic would be to investigate whether the positive PAC result of this paper holds for colored unary partial function graphs, i.e., when the vocabulary is extended by a set of unary background predicates. Finally, it would be interesting to apply the product homomorphism method to other classes of directed graphs, in particular to classes generalizing unary partial function graphs.

References

1. A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989.
2. R. Diestel. *Graph theory*. Springer-Verlag, New York, 2nd edition, 2000.
3. T. Horváth, R. Sloan, and G. Turán. Learning logic programs by using the product homomorphism method. In *Proc. of the 10th Annual Conference on Computational Learning Theory (COLT-97)*, pages 10–20, New York, 1997. ACM Press.
4. T. Horváth and G. Turán. Learning logic programs with structured background knowledge. *Artificial Intelligence*, 128(1-2):31–97, May 2001.
5. J.-U. Kietz and S. Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.
6. D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Second Edition, Addison-Wesley, Reading, 1981.
7. J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.
8. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
9. L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1985.

Specific Biases for Mining Frequent Substructures

Akihiro Inokuchi[1,2], Takashi Washio[2], and Hiroshi Motoda[2]

[1] Tokyo Research Laboratory, IBM Japan,
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
inokuchi@jp.ibm.com
[2] The Institute of Scientific and Industrial Research, Osaka Univ.,
8-1 Mihogaoka, Ibaraki, Osaka, 567-0047 Japan
{washio, motoda}@ar.sanken.osaka-u.ac.jp

Abstract. Data mining to derive frequent subgraphs from a dataset of general graphs has high computational complexity because it includes the explosively combinatorial search for candidate subgraphs and subgraph isomorphism matching. Although some approaches have been proposed to derive characteristic patters from graphs, they limit the graphs to be searched within a specific class. In this paper, we propose an approach to conduct a complete search of various classes of frequent subgraphs in a massive dataset of labeled graphs within practical time, and discuss the relation between the biases and diffusion kernels. The power of our approach comes from the algebraic representation of graphs, its associated operations and well-organized bias constraints to limit the search space efficiently. Its performance has been evaluated using real world datasets, and the high scalability of our approach has been confirmed with respect to the amount of data and the computation time.

1 Introduction

Graph mining which discovers characteristic subgraph patterns embedded in a general graph dataset is an important problem having broad applications. This problem is very difficult to solve in practical time because the search of candidate frequent subgraphs is explosively combinatorial, and includes subgraph isomorphism matching which is known to be NP-complete. For example, WARMR is a system to mine frequent subgraphs based on inductive logic programming (ILP) [3]. It could derive all subgraphs consisting of only a few vertices within tractable time under its application to a raw graph dataset without introducing descriptions of higher level structures.

To alleviate the difficulty of the computation time, some other approaches have introduced certain approximations. The most well known approximation is to apply a greedy search as in SUBDUE [2] and GBI [14, 11]. However, they are not very suitable to many applications requiring a complete search of the results. Another approach is to limit the graphs to be searched within simpler classes. The levelwise version space algorithm proposed by De Raedt et al. limits the search space to frequent paths embedded in the graphs for the tractable computation [4]. However, this is also weak for certain application areas to require the mining of subgraphs but not paths.

To overcome these limitations, some approaches which achieve to mine a complete set of frequent patterns from a massive dataset of graphs or ordered trees have been proposed. Although each method can efficiently discover the patterns, the searched patterns are limited within a specific class. For example, TreeMiner [15] and FREQT [1] can quickly discover all frequent patterns from ordered trees or non-ordered trees. However, they can not be applied to more complex structures such as labeled graphs. On the other hand, AGM [5], FSG [10], and gSpan [13] can mine frequent patterns from a set of graphs. However, they can not efficiently discover frequent patterns of paths and/or trees, because their data structures and their search operations are not dedicated to path and/or tree structure mining.

In this paper, we propose a generic and efficient framework to mine various classes of substructures. By introducing bias for each class of substructure, *e.g.*, connected subgraphs, path structures, ordered subtrees, and general subtrees, to the principle of AGM, the complete search of the frequent substructures of each class is achieved. We call this new framework *Biased Apriori-based Graph Mining (B-AGM)*. We discuss the relation between the biases for frequent subgraph mining and diffusion kernels. We evaluate its performance in terms of the required computation time under various amounts of real world datasets.

2 Problem Definitions and AGM

We use the basic principle of AGM algorithm in our extended framework. By applying some specific biases to the algorithm, our B-AGM discovers frequent subgraphs of various classes. In this section, we explain the problem definitions and the AGM framework. In the next section we propose the biases which enable the graph mining of various classes.

2.1 Graph structured data

A graph G consists of 4 sets, the set of vertices V, the set of edges E, the set of vertex labels L_V, and the set of edge labels L_E. When V, E, L_V and L_E are provided as $V = \{v_1, v_2, \ldots, v_k\}$, $E = \{e_h = (v_i, v_j) | v_i, v_j \in V\}$, $L_V = \{lb(v_i) | \forall v_i \in V\}$, and $L_E = \{lb(e_h) | \forall e_h \in E\}$, respectively, the graph G is mathematically defined as $G = (V, E, L_V, L_E)$, where multiple vertices can have an identical label, and multiple edges also can have. For the convenience of description, the tuples of G are represented as $V(G)$, $E(G)$, $L_V(G)$, and $L_E(G)$, respectively. The number of vertices, $|V(G)|$, is called the *size* of the graph G.

A graph structure can be expressed by using an adjacency matrix. Let $num(lb(e_h))$ and $num(lb(v_i))$ be natural numbers assigned to an edge label $lb(e_h)$ and a vertex label $lb(v_i)$, respectively for calculation efficiency. Given a graph $G = (V, E, L_V, L_E)$, (i, j)-element $x_{i,j}$ of an adjacency matrix X_k representing the graph G of size k is defined as follows.

$$x_{i,j} = \begin{cases} num(lb(e_h)) & \text{if } e_h = (v_i, v_j) \in E(G) \\ 0 & \text{if } (v_i, v_j) \notin E(G) \end{cases},$$

where $i, j \in \{1, \cdots, k\}$. The vertex corresponding to the i-th row (i-th column) of an adjacency matrix is called the i-th vertex. The graph structure represented by an adjacency matrix X_k is denoted as $G(X_k)$. The representation of the adjacency matrix on an identical graph differs depending on the assignment of rows and columns to the vertices of the graph. To remove this ambiguity, a *canonical form* of adjacency matrices representing an identical graph is introduced.

To mathematically define the canonical form of a graph and for the efficient matrix handling, an adjacency matrix X_k is represented and processed in form of the following code[3].

$$code(X_k) = x_{1,2}x_{1,3}x_{2,3}x_{1,4}\cdots x_{k-2,k}x_{k-1,k},$$

where it is a concatenation of (i, j)-element $x_{i,j}$. Furthermore, the $CODE$ including the vertex labels is defined as

$$CODE(X_k) = num(lb(v_1))\cdots num(lb(v_k))code(X_k),$$

where it is a concatenation of $num(lb(v_i))$s and $code(X_k)$. The canonical form of the adjacency matrices representing a graph is the unique matrix having the maximum (minimum) $CODE$.

Given a set GD of graph structured data, the support $sup(G_s)$ of a subgraph G_s is defined as the ratio of the number of graphs including G_s to the total number of graph data in the dataset GD. The subgraph having the support more than or equal to a specified *minimum support* is called a *frequent subgraph*. A frequent subgraph of size k is called a frequent k-subgraph. When the dataset of graph structured data and the minimum support are given as input, the data mining of graph structure is to derive all frequent subgraphs in the dataset [6].

2.2 AGM algorithm

We proposed an approach named AGM (Apriori-based Graph Mining) algorithm in which the knowledge representation and the search operations are highly dedicated to the graph structured data mining [5, 7]. The AGM algorithm is so generic that it can discover not only frequent connected subgraphs but also frequent unconnected subgraphs. AGM framework derives all frequent subgraphs based on the monotonic property of the support measure. Based on this property, frequent subgraphs are derived stepwisely in ascending order of their sizes beginning with frequent 1-subgraphs. Fig. 1 is the outline of the AGM algorithm. First, a 1×1 adjacency matrix representing a vertex is generated for every vertex label appearing in the dataset, and they are substituted for C_1. Next, the support of each candidate frequent subgraph is calculated by accessing the database, and only the frequent subgraphs are retained in a set F_1. Subsequently, Generate-Candidate function generates the candidate frequent subgraphs of size $k+1$ from frequent k-subgraphs in F_k, and they are substituted for C_{k+1}. This process is repeated until C_k becomes empty. Finally, the complete set of frequent subgraphs is provided.

[3] This definition can be expanded to directed graph [7].

Generate-Candidate function consists of three parts, *i.e.*, join operation, subgraph checking operation, and canonical form derivation. In join operation, adjacency matrices of the candidate frequent subgraphs of size $k+1$ are generated by joining two adjacency matrices of frequent k-subgraphs in F_k. Given two adjacency matrices X_k and Y_k in F_k, they are joinable if and only if all of the following conditions are fulfilled.
<u>*Condition 1*</u> X_k and Y_k are identical except the k-th row and the k-th column, *i.e.*,

$$X_k = \begin{pmatrix} X_{k-1} & \mathbf{x}_1 \\ \mathbf{x}_2^T & 0 \end{pmatrix}, Y_k = \begin{pmatrix} X_{k-1} & \mathbf{y}_1 \\ \mathbf{y}_2^T & 0 \end{pmatrix}, \text{and}$$

$$lb(v_i \in V(G(X_k))) = lb(v_i \in V(G(Y_k))), \ (i = 1, \cdots, k-1).$$

<u>*Condition 2*</u> X_k is the canonical form of $G(X_k)$. This avoids the redundant join.
<u>*Condition 3*</u> $CODE(X_k) \geq CODE(Y_k)$ is fulfilled.[4].

If X_k and Y_k are joinable, their join operation is defined as follows.

$$Z_{k+1} = \begin{pmatrix} X_{k-1} & \mathbf{x}_1 & \mathbf{y}_1 \\ \mathbf{x}_2^T & 0 & z_{k,k+1} \\ \mathbf{y}_2^T & z_{k+1,k} & 0 \end{pmatrix},$$

$$lb(v_i \in V(G(Z_{k+1}))) = lb(v_i \in V(G(X_k))), \ (i = 1, \cdots, k), \text{and}$$
$$lb(v_{k+1} \in V(G(Z_{k+1}))) = lb(v_k \in V(G(Y_k))).$$

X_k and Y_k are called the first generator matrix and the second generator matrix of Z_{k+1}, respectively. Two elements $z_{k,k+1}$ and $z_{k+1,k}$ of Z_{k+1} are not determined by X_k and Y_k. The possible graph structures for $G(Z_{k+1})$ are those wherein there is a labeled edge or wherein there is no edge between k-th vertex and $\{k+1\}$-th vertex. Then, $(|L_E(E)|+1)$ adjacency matrices with $z_{k,k+1} = z_{k+1,k}$ are generated, where $|L_E(E)|$ is the number of edge labels. The adjacency matrix generated under the above conditions is called a *normal form*.

For the necessary condition of $G(Z_{k+1})$ being a frequent subgraph, all induced subgraphs of $G(Z_{k+1})$ must be frequent subgraphs. The application of this condition reduces the candidates. This is done though subgraph checking in $G(Z_{k+1})$ which is described in detail in the literature [6]. After generating the matrices of candidate subgraphs, a database is accessed to calculate their supports. However, the canonical form of these matrices must be identified to collect all counts of the graph, since multiple normal form matrices can represent an identical graph. This is done through canonical form derivation which is described in detail in the literature [5, 7].

3 Extension to Mine Various Class Structures

The original AGM performs the complete mining of the frequent and general subgraphs. However, a version of AGM contains a bias to derive the frequent

[4] In the case that a canonical form is the minimum $CODE$, $CODE(X_k) \leq CODE(Y_k)$ is fulfilled.

// GD is a database consisting of graph structured data.
// F_k and C_k is a set of adjacency matrix of frequent and candidate k-graphs, resp.
// $minsup$ is minimum support.

```
Main(GD, minsup){
  C_1 ←{all adjacency matrices consisting of one vertex};
  k ← 1;
  while(C_k ≠ ∅) {
    Count(GD, C_k);
    F_k ← {c_k ∈ C_k | sup(G(c_k)) ≥ minsup};
    C_{k+1} ← Generate-Candidate(F_k);
    k ← k + 1;
  }
  return ⋃_k {f_k ∈ F_k | f_k is canonical};
}
```

Fig. 1. Outline of Algorithm

induced subgraph only [5, 7]. The induced subgraph of a graph G has a subset of the vertices of G and the same edges between pairs of vertices as in G. To limit the search of the frequent subgraph within this class, the following bias has been applied in the past work. When counting frequency of each candidate frequent subgraph, AGM checks whether it is contained in each graph in a database as an induced subgraph.

In the following subsection, we propose further biases which enable the graph mining of various classes based on the AGM framework as depicted in Fig. 2. We call this framework *B-AGM*. A bias for a specific class of the graph structure consists of the dedicated definitions of the canonical form and the join operation. By choosing appropriate bias on the platform of the AGM framework, the complete mining for the frequent subgraphs of the objective class we are seeking for is defined.

Fig. 2. B-AGM Framework

Fig. 3. Tree Coding Method

3.1 Bias for ordered subtree derivation

This bias derives frequent ordered trees included in a forest.

Canonical Form

The definition of canonical form is altered from the original. When the preorder

natural numbers which are assigned to the vertices of an ordered tree are consistent with the columns and rows of its adjacency matrix, the matrix is defined as the canonical form of the ordered tree as shown in Fig. 3. This is because the numbering of the preorder in the ordered tree uniquely specifies the topological tree structure. Accordingly, the definition of the preorder tree represents the canonical form. Under this definition, AGM can search only canonical form matrices but not non-canonical form matrices. This extremely enhances the search efficiency. Its proof is omitted due to the space limitation.

Join Operation

The original condition 1 and 2 are retained, and the following new condition 3 and an extra condition 4 are introduced.

Condition 3 In the case that $G(Y_k)$ is connected, $code(X_k) \leq code(Y_k)$ is fulfilled.

Condition 4 $G(X_k)$ is a connected graph[5].

4 Relation between Mining Biases and Diffusion Kernels

In this section, we discuss the relation between our biases for frequent subgraph mining and diffusion kernels. Kernel-based algorithms are paid much attention in the statistical learning community. Konder et. al introduced diffusion kernels for direct use to learning algorithm which was used to quantifying the similarity between vertices in a graph [9]. Diffusion process on a graph for discrete time is modeled as follows. Heat or other substances on each vertex diffuse to its respective neighbors through the corresponding edges with time. Given an undirected graph G, a matrix to determine diffusion process is defined as

$$H_{i,j} = \begin{cases} 1 & \text{for } i \sim j \\ -d_i & \text{for } i = j \\ 0 & \text{otherwise} \end{cases},$$

where d_i is the degree of vertex i, and $i \sim j$ means that there is an undirected edge between i-th and j-th vertices. It is similar to an adjacency matrix. Consider the random field obtained by attaching independent, zero mean variance σ^2 random variables $Z_i(0)$ to each vertex i. The random variables diffuse a fraction α to the neighbor at discrete time steps $t = 1, 2, \cdots$; that is,

$$Z_i(t+1) = Z_i(t) + \alpha \sum_{j \in V : i \sim j} (Z_j(t) - Z_i(t)), \tag{1}$$

where V is a set of vertixes of the graph. Assuming $T(t) = (1 + \alpha H)^t$, $Z(t) = (Z_1(t), \cdots, Z_{|V|}(t))^T$ can be written as $Z(t) = T(t)Z(0)$. An exponential kernel K for a graph is defined as $K = e^{\beta H}$. The (i, j)-element of K is used to quantifying a similarity between i-th and j-th vertices of a graph, which means if structures of neighborhoods of two vertices are similar, quantities of diffusing substances should also be similar after infinity time steps.

[5] Other biases for connected graph, path, and general tree derivations and their corresponding experiments are described in [8].

Fig. 4. Diffusion Process

Fig. 5. Candidate Graph Generation

For sake of simplicity, we discuss a complete graph G. Eq. (1) is represented as

$$Z(t+1) = (I + \alpha H)Z(t). \tag{2}$$

Let G be a complete graph with five vertices, and one of them be filled with a certain substance, *i.e.*, $Z_1(0) = 1$, and $Z_i(0) = 0$ for $i = 2, 3, 4, 5$. If it comes to equilibrium in one step, the value of α is $\frac{1}{5}$ according to Eq. (2). It is denoted as $Z(1) = [I + \frac{1}{5}H]Z(0)$. Fig. 4 shows that the $\frac{1}{5}$ of substance diffuses from a vertex filled with the substance to another vertex not containing the substance in one step, and the substance is at equilibrium.

On the other hand, our AGM mines frequent subgraphs in a stepwise manner by expanding vertices. A bias to derive frequent subgraphs which are complete graphs with no vertex and edge labels are defined as follows.

Canonical Form

All matrices generated by the following join operation are canonical forms.

Join Operation

The original condition 1 and 2 are retained, and the following new condition is added. $X_k = Y_k$ and both $z_{k,k+1}$ and $z_{k+1,k}$ of generated matrices are 1.

Fig. 6. Diffusion Process (2)

Adjacency matrices of candidate graphs are stepwisely generated as

$$X_1 = \begin{pmatrix} 0 \end{pmatrix} \Rightarrow X_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow X_3 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \Rightarrow X_4 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \Rightarrow X_5 = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix},$$

where the number of matrices in each size becomes 1. Assuming A is an adjacency matrix of a complete graph G with no self loops, Eq. (2) is

$$Z(t+1) = [I + \alpha(A - d_i I)]Z(t) = [(1 - \alpha d_i)I + \alpha A]Z(t). \tag{3}$$

In the case of general diffusion, the substance diffuses to all adjacent vertices. However, we assume that the substance diffuses to one of vertices not containing the substance one by one as shown in Fig. 6. First, a vertex is filled with the substance in the initial state as well as Fig. 4. Next, a vertex is added to the vertex with the substance. Subsequently, the substance in the vertex diffuses to the added vertex in the one step. The substance gradually diffuses to the other vertexes. Finally, all of the vertices are filled with 1/5 of the substance. H, α, and d_i change with time in our case, although they are uniform in the general case. Therefore, we rewrite Eq. (2) and (3) as

$$Z(t+1) = [I + \alpha(t)H(t)]Z(t) = [\{1 - \alpha(t)d_i(t)\}I + \alpha(t)A(t)]Z(t). \tag{4}$$

In concrete terms, $\alpha(t)$, $d_i(t)$ and $A(t)$ is denoted as

$$\alpha(0) = \frac{1}{2},\ \alpha(1) = \frac{1}{3},\ \alpha(2) = \frac{1}{4},\ \alpha(3) = \frac{1}{5},\ d_i(0) = 1,\ d_i(1) = 2,\ d_i(2) = 3,\ d_i(3) = 4,$$

$$A(0) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\ A(1) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\ A(2) = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\ A(3) = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

In general, $\alpha(t) = \frac{1}{t+2}$, $d_i(t) = t + 1$. Hence, Eq. (4) for a complete graph is

$$Z(t+1) = \frac{1}{t+2}[I + A(t)]Z(t). \tag{5}$$

The substance diffuses to one of vertices not containing the substance one by one in our case, although the substance diffuses to all adjacent vertices in the case of general diffusion. Since to limit the search space to a specific class of frequent subgraphs by a bias corresponds to constraining a matrix H in the diffusion kernel, we can conclude that our biases for frequent subgraph mining are related to diffusion kernels. There is a systematic regularity for the complete graph, and furthermore such systematic regularities for other classes of graphs may similarly exist. If Eq. (5) is generalized for the other classes of the graphs, approximate subgraph matching instead of exact subgraph matching may be able to be applied, when the support values of candidate graphs are calculated. We will investigate other cases that derived subgraphs are not complete graphs in our future plan.

Fig. 7. Examples of Ordered Tree Data & Pattern

Fig. 8. Minimum Support vs. Computation Time

5 Experiment and Discussion

IBM PC 300PL with Windows 2000 was used for the experiments, with a Pentium III-667 MHz and 192 MB of main memory. Zaki presented experimental results of substructure discovery from a set of logs files over 1 month at the RPI computer science department's web site [15]. After the preprocessing, the dataset had 595,691 user browsing subtree with 13,361 unique labels (web pages). We used the same data provided by Zaki on our experiment and recalculated with the same machine to compare our approach with TreeMiner. TreeMiner can find frequent patterns embedded in tree dataset. It is defined that pattern P discovered by TreeMiner is embedded in tree data D if and only if two vertices in a branch in P are on the same path from the root to a leaf in D. For example, Fig. 7, an ordered tree in the right is embedded in the data. We used the same definition in the comparison of the result to obtain equivalent result. The function to count frequency is changed to adjust to the definition of frequent patterns of TreeMiner in B-AGM. Fig. 8 shows computation time for various minimum support values. Fig. 8 shows that B-AGM is comparable to TreeMiner.

FSG [10], gSpan [13], MolFea [4], TreeMiner [15] and FREQT [1] take a complete search strategy similarly to AGM. FREQT and TreeMiner require $O(n)$ memory to store tree data, where n is the number of vertices in the tree. Although our approach requires $O(n^2)$ memory for the storage, FREQT and TreeMiner can not be applied to graph mining problems. Our proposed method can conduct a complete search of various classes of frequent subgraphs. The advantage of our approach is very flexible and quickly adaptable for new classes of patterns in graph mining. AGM with each bias can derive the complete result within practical time for each class of problem respectively where the performance is far faster than or comparable to the other approaches which are optimized within a specific class [8]. This is because the representation of graphs and its associated operations are well-organized and dedicated to the graph mining. Our method is considered to be highly practical in many applications.

6 Conclusion

We proposed a generic framework for the data mining of graph structures. By introducing additional biases, it can easily derive various types of frequent sub-

structures. We evaluated its performance in terms of the required computation time for the real world datasets. The wide cover on problem classes and the computational efficiency which are superior to the other approaches have been confirmed. We will investigate the relation between our biases for frequent subgraph mining and diffusion kernels in other cases that derived subgraphs are not complete graphs in our future plan.

Acknowledgement

We would like to thank Professor Mohammed Zaki at Rensselaer Polytechnic Institute and Professor Hiroki Arimura and Mr. Tatsuya Asai at Kyushu University for their help and advice.

References

1. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H. & Arikawa, S. Efficient Substructure Discovery from Large Semi-Structured Data. *Proc. of SIAM Int'l. Conf. on Data Mining*, pp. 158–174, (2002).
2. Cook, D. & Holder, L. Substructure Discovery Using Minimum Description Length and Background Knowledge. *Journal of Artificial Intelligence Research*, Vol.1, pp. 231–255, (1994).
3. Dehaspe, L., Toivonen, H., & King, R. D. Finding frequent substructures in chemical compounds. *Proc. of the 4th International Conference on Knowledge Discovery and Data Mining* (KDD-1998), pp. 30–36, 1998.
4. De Raedt, L. & Kramer, S. The Levelwise Version Space Algorithm and its Application to Molecular Fragment Finding. *Proc. of Int'l. Joint Conf. on Artificial Intelligence*, pp. 853–859, (2001).
5. Inokuchi, A., Washio, T. & Motoda, H. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. *Proc. of European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pp. 13–23, (2000).
6. Inokuchi, A., Washio, T., Nishimura, Y. & Motoda, H. A Fast Algorithm for Mining Frequent Connected Graph. *IBM Research Report* RT0448, (2002), February.
7. Inokuchi, A., Washio, T. & Motoda, H. Complete Mining of Frequent Patterns from Graphs: Mining Graph Data, *Machine Learning*, 50 (3): 321-354, March, 2003,
8. Inokuchi, A., Washio, T., & Motoda, H. A General Framework for Mining Frequent Patterns in Structures. *IBM Research Report* RT0513, (2003), February.
9. Konder, R. I., & Lafferty, J. Diffusion kernels on graphs and other discrete input spaces. *Proc. of Int'l Conference on Machine Learning* (ICML-2002), 2002.
10. Kuramochi, M., & Karypis, G. Frequent Subgraph Discovery. *Proc. of the 1st International Conference on Data Mining* (ICDM-2001), pp.313–320, 2001.
11. Motoda, H. & Yoshida, K. Machine Learning Techniques to Make Computers Easier to Use. *Proc. of Int'l. Joint Conf. on Artificial Intelligence*, Vol. 2, pp. 1622–1631, 1997.
12. Yan, X. & Han, J. gSpan: Graph-Based Substructure Pattern Mining. *Proc. of the 3rd International Conference on Data Mining*, (ICDM-2002), pp.721–724, 2002.
13. Yoshida, K., & Motoda, H. CLIP: Concept Learning from Inference Patterns. *Artificial Intelligence*, Vol. 75, No. 1 pp. 63–92, 1995.
14. Zaki, M. Efficiently Mining Frequent Trees in a Forest. *Proc. of the 8th International Conference on Knowledge Discovery and Data Mining* , pp. 71–80, 2002.

Efficient Discovery of Frequent Unordered Trees

Siegfried Nijssen and Joost N. Kok

Leiden Institute of Advanced Computer Science,
Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands
snijssen@liacs.nl

Abstract. Recently, an algorithm called FREQT was introduced which enumerates all frequent induced subtrees in an *ordered* data tree. We propose a new algorithm for mining *un*ordered frequent induced subtrees. We show that the complexity of enumerating unordered trees is not higher than the complexity of enumerating ordered trees; a strategy for determining the frequency of unordered trees is introduced.

1 Introduction

Recently, several exploratory data mining algorithms for structural databases have been proposed which search for *frequent* structures in such databases. These frequent pattern mining algorithms use the same principles as APRIORI [2], the well-known frequent item set mining algorithm. A structure may be a sequence, a tree or a graph. In this paper, we focus on frequent tree mining. In general, the frequent tree discovery task is the task of discovering all trees — referred to as the *pattern* trees — that occur frequently in some large tree called a *data tree*. Within this general setup, there are several blanks to be filled in:

- What kind of trees are considered? Is the given database an ordered tree, a node labeled tree or an edge labeled tree?
- What kind of occurrence relation is used? Does a pattern tree occur in a data tree when the pattern tree is an *induced* subtree (which means that parental relations between vertices in the data tree must be the same as in the pattern tree) or when it is an *embedded* subtree (where a parent in a pattern tree may be an ancestor in the data tree)?
- How are tree occurrences counted? Is each occurrence in the data tree counted, or is the data tree partitioned into several separate trees, and is only the occurrence of a tree in a sufficient number of partitions interesting?

Previous publications have dealt with several of these possibilities:

- Wang and Liu [5] developed an algorithm for discovering both ordered and unordered induced edge-labeled subtrees. A pattern tree is frequent when its root can be mapped to the *root* of a sufficient number of partitions in the data tree. To determine the inclusion relation, for each pair of pattern and partition, an $O(nm^{1.5})$ algorithm is used, where n is the size of the pattern tree and m is the size of the partition. The algorithm features a strategy for enumerating unordered trees which requires some redundant trees to be generated.

- Asai et al. [1] developed an algorithm called FREQT for discovering induced subtrees in a node labeled, ordered data tree. The algorithm counts each occurrence in a data tree, but can easily also be used for partition counting. It uses an efficient scheme for generating and counting ordered trees.
- Zaki [6] developed an algorithm for discovering embedded subtrees in a node labeled, ordered data tree. The algorithm counts occurrences in partitions, but can easily also be used to count occurrences separately. It enumerates trees in a similar way as FREQT, but uses a different evaluation technique based on *scope lists*.

The algorithms of Asai et al. [1] and Zaki [6] share their efficient enumeration technique for *ordered* trees; both algorithms define their own evaluation technique for such trees. As also indicated by Wang et al. [5], in some structured databases a child order is of minor importance or even unavailable; for such databases, it is more interesting to search for patterns that do not take the order into account. In this paper, we will propose an algorithm for discovering *unordered* frequent induced subtrees.

In section 2 we will extend the enumeration technique of [1] and [6] to efficiently enumerate unordered trees. We will define one ordered tree to be the normal form of the unordered trees. As a consequence of the absence of order, a new evaluation technique is required to compute the frequency of trees. In section 3 we will therefore define a new bottom-up algorithm for this task. This algorithm promises to be more efficient than the approach of [5] as it reuses the matchings of previous trees to compute the frequency of new trees.

2 Unordered Tree Enumeration

For an efficient algorithm, it is of major importance that all possible pattern trees are enumerated efficiently. An efficient enumeration technique is a technique that enumerates each unordered tree exactly once. We will first show why techniques from [1] and [6] cannot straightforwardly be applied to unordered trees.

The algorithms of [1] and [6] use rightmost path expansions. Starting with pattern trees with only one node, nodes are added only to the rightmost path to generate new pattern trees. The technique is illustrated in Fig. 1. As can be seen in this figure, several ordered pattern trees may be constructed by this technique which represent the same unordered pattern trees. Ordered trees which represent the same unordered pattern trees are considered to be *equivalent*. In an efficient enumeration technique, no two equivalent pattern trees are constructed. We will propose an efficient technique here; the technique is novel to the best of our knowledge.

In our technique, we define one of the equivalent ordered trees to be the normal form of the corresponding unordered tree. We first introduce some notation. Given a node v in an ordered tree T, firstchild(v) denotes the first child of v, lastchild(v) is the last child, nextsibling(v) is the next node in the ordered child list of v's parent and prevsibling(v) is the previous node in that list. With subtree(v) we denote the subtree in T of which v is the root.

Tree T_i^*	String $l(T_i^*)$
T_1^*	$(0,A)(1,A)(2,A)(1,A)$
T_2^*	$(0,A)(1,A)(1,A)(2,A)$
T_3^*	$(0,A)(1,A)(1,A)(1,A)$
T_4^*	$(0,A)(1,A)(1,A)$
T_5^*	$(0,A)(1,A)(1,B)$
T_6^*	$(0,A)(1,B)(1,A)$
T_7^*	$(0,B)(1,A)(1,B)$
T_8^*	$(0,B)(1,B)(1,A)$

Fig. 1. Enumeration of pattern trees in the alphabet $\{A, B\}$ using rightmost path expansions. Only a selected number of trees is expanded after 2 steps. Dotted lines indicate an expansion. By conceiving pattern trees as nodes and expansions as edges, an enumeration tree [1] is obtained which relates pattern trees to each other. Pattern trees T_1^* and T_2^*, trees T_5^* and T_6^* and trees T_7^* and T_8^* are equivalent. Of a selected number of trees, the pre-order label is given.

Given an ordered tree T, we define the following pre-order string $l(T)$ for this tree: in a depth-first tree traversal, add a tuple label $l(v)$ =(depth(v),label(v)) for each node v to an initially empty string when this node v is visited for the first time. Some examples of this pre-order notation are also given in Fig. 1.

Note that each of these strings corresponds to exactly one tree. In a string, the order of the tuples exactly matches the order of rightmost path expansions; a pre-order string of a tree can therefore also be read as a series of subsequent tree expansions that leads to this tree.

A tree T_1 is called a prefix of a tree T_2 if $l(T_1)$ is a prefix of $l(T_2)$. Tree T_1 is an immediate prefix of T_2 if T_1 is a prefix of T_2 and $|T_1| + 1 = |T_2|$. A suffix is defined analogously.

Given an order on the labels, we define the following order on tuples: $(d_1, l_1) < (d_2, l_2)$ iff $d_1 > d_2$ (this may sound counterintuitive, but will become clear later) or $l_1 < l_2$ if $d_1 = d_2$. Other (in)equalities are derived from this order.

Given two trees T_1 and T_2, we define that $l(T_1) < l(T_2)$ iff:

- either, T_2 is a prefix of T_1,
- or, at the leftmost position i at which $l(T_1)$ and $l(T_2)$ differ, $(d_1, l_1) < (d_2, l_2)$ for the tuples $(d_1, l_1) \in T_1$ and $(d_2, l_2) \in T_2$ occurring at that position.

In the example, $l(T_1^*) < l(T_2^*) < l(T_3^*) < l(T_4^*) < l(T_5^*) < l(T_6^*) < l(T_7^*) < l(T_8^*)$. The enumeration tree of Fig. 1 was obtained by expanding the rightmost path bottom-up; the order of the pattern trees is obtained by performing a post-order walk in the enumeration tree.

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
$(0, A)$	$(1, A)$	$(1, B)$	$(2, B)$	$(3, A)$	$(4, A)$	$(3, A)$	$(4, B)$	$(4, B)$	$(3, A)$	$(4, B)$	$(2, B)$	$(3, A)$	$(4, A)$	$(3, A)$

Fig. 2. A tree in ordered normal form and its pre-order string. The nodes are numbered in the order of expansion.

We define that a tree T_1 is in (ordered) normal form if no equivalent tree T_2 exists with $l(T_2) < l(T_1)$.

We will now illustrate some observations with respect to this normal form.

Lemma 1. *Given an ordered tree T, T is in normal form if and only if for each node $v \in T$, $l(subtree(v)) \leq l(subtree(v'))$, for each next sibling v' of v.*

Proof. Omitted here; see [3]. □

We will illustrate this lemma on the example of Fig. 2. Under the assumption that this tree is in normal form, we consider two node order changes: an exchange of nodes v_5 and v_7 (exchange 1) and an exchange of nodes v_7 and v_{10} (exchange 2). In the pre-order string, these exchanges correspond to exchanges of subtrings:

$$(0,A)(1,A)(1,B)(2,B)|\underline{(3,A)(4,A)}|(3,A)(4,B)(4,B)|(3,A)(4,B)(2,B)(3,A)(4,A)(3,A)$$
$$\Rightarrow \text{Exchange 1} \Rightarrow$$
$$(0,A)(1,A)(1,B)(2,B)|\underline{(3,A)(4,B)(4,B)}|\underline{(3,A)(4,A)}|(3,A)(4,B)(2,B)(3,A)(4,A)(3,A)$$

$$(0,A)(1,A)(1,B)(2,B)(3,A)(4,A)|\underline{(3,A)(4,B)(4,B)}|\underline{(3,A)(4,B)}|(2,B)(3,A)(4,A)(3,A)$$
$$\Rightarrow \text{Exchange 2} \Rightarrow$$
$$(0,A)(1,A)(1,B)(2,B)(3,A)(4,A)|\underline{(3,A)(4,B)}|\underline{(3,A)(4,B)(4,B)}|(2,B)(3,A)(4,A)(3,A)$$

As the tree was in normal form, both exchanges should yield an equivalent tree with a higher pre-order string. Indeed, in case of exchange 1, $(4, B) > (4, A)$ due to $l(\text{subtree}(v_7)) = (3, A)(4, B)(4, B) > (3, A)(4, A) = l(\text{subtree}(v_5))$; in case of exchange 2, $(3, A) > (4, B)$ as $l(\text{subtree}(v_{10}))$ is a prefix of $l(\text{subtree}(v_7))$.

Lemma 2. *Let T be a pattern tree in normal form. Then every prefix of T is also in normal form.*

Proof. This follows from the previous lemma. □

According to this lemma it is sufficient to only generate rightmost path expansions that immediately yield trees in normal form; even then one still enumerates a normal form for each possible tree: the enumeration is *complete*. We will now consider such expansion techniques in more detail. How can valid expansions be characterized?

By a rightmost path expansion only subtrees of some nodes on the rightmost path are modified. Only for these subtrees one has to check again that they are higher than their previous sibling subtree. In the example of Fig. 2, expansions which lead to a tree in normal form, are:

- $(4, l)$, with $l \geq l(v_8) = B$. If $l < B$: subtree(v_{12}) < subtree(v_4), which is not allowed; if $l = B$: subtree(v_{12}) is still a prefix of subtree(v_4), and therefore higher. If $l > B$: subtree(v_{12}) > subtree(v_4). From subtree(v_7) > subtree(v_5) follows that subtree(v_{15}) > subtree(v_{13}) if $l \geq B$. Finally, before expansion, already subtree(v_3) > subtree(v_2), while subtree(v_3) was not a prefix. This shows that all subtrees are still higher than their previous sibling;
- $(3, l)$, with $l \geq l(v_{15}) = A$; obviously, in this case the new node is higher than or equal to subtree(v_{15}), and subtree(v_3) > subtree(v_2). Also subtree(v_{12}) > subtree(v_4), as the new node in subtree(v_{12}) is at a higher level in the tree than the 'next' node v_8 in subtree(v_4);
- $(2, l)$ with $l \geq l(v_{12}) = B$: the new node is higher than the previous sibling subtree(v_{12}); subtree(v_3) was already higher than subtree(v_1);
- $(1, l)$ with $l \geq l(v_3) = B$: the new node is higher than its previous sibling.

The example shows the importance of knowing the largest suffix subtree (subtree(v_{12})) which is a prefix of its previous sibling subtree (subtree(v_4)). This previous sibling restricts the level and the label of new nodes. One can show the following:

Lemma 3. *Given a tree T in normal form, the* lowest prefix node *v is the node on the rightmost path for which the size of subtree(v) is maximized and subtree(v) is a prefix of subtree(prevsibling(v)). A tree may not have a lowest prefix node, in which case the lowest prefix node is undefined. The* next prefix node *(d', l') is the node in l(subtree(prevsibling(v))) immediately after l(subtree(v)), if a lowest prefix node v is defined. An expansion (d, l) yields a tree in normal form iff: $(d, l) \geq p(d)$ and $(d, l) \geq (d', l')$ (if the lowest prefix node is defined). Here $p(d)$ is the label of the node at depth d on the rightmost path of T.*

Proof. Omitted here; see [3]. □

Furthermore, one can also show the following:

Lemma 4. *Given is a tree T in normal form which is normally expanded with a node (d, l). Then the location of the lowest prefix node either:*

- *does not change if (d, l) equals the next prefix node (d', l') (when defined);*
- *or, otherwise, becomes (d, l) if l equals the label of $p(d)$;*
- *or, otherwise, becomes undefined.*

Proof. Omitted here; see [3]. □

Algorithm Enumerate
Input: a tree T in normal form, its representation as a string $l(T)$, and an index t which is either undefined or points to a position in $l(T)$.
Output: a print of each tree that can be obtained by expanding this tree to a new tree in normal form.

1. print(T)
2. **if** t is defined **then**
3. Increase t
4. Let (d, l) be the tuple at position t in $l(T)$.
5. Enumerate (T expanded with (d, l), $l(T) \cdot (d, l)$, t).
6. **for each** rightmost expansion $(d', l') > (d, l)$, $l' \geq l(p(d'))$ **do**
7. **if** $l' = l(p(d'))$ **then**
8. Enumerate (T expanded with (d', l'), $l(T) \cdot (d', l')$, position of $p(d')$ in $l(T)$);
9. **else**
10. Enumerate (T expanded with (d', l'), $l(T) \cdot (d', l')$, undefined);
11. Decrease t
12. **else**
13. **for each** rightmost expansion (d', l'), $l' \geq l(p(d'))$ **do**
14. **if** $l' = l(p(d'))$ **then**
15. Enumerate (T expanded with (d', l'), $l(T) \cdot (d', l')$, position of $p(d')$ in $l(T)$);
16. **else**
17. Enumerate (T expanded with (d', l'), $l(T) \cdot (d', l')$, undefined);

Fig. 3. An algorithm for enumerating all trees in normal form.

If the example tree is expanded with $(4, B)$, v_{12} remains the lowest prefix node. If the example tree is expanded with $(4, C)$, the tree no longer has a lowest prefix node.

All these observations can be used to construct an efficient enumeration algorithm, as given in Fig. 3. With $l(T) \cdot (d, l)$ we denote the concatenation of $l(T)$ and (d, l). Index t points to the next prefix node. By increasing t in line 3., we either obtain the next prefix node (which should be added to the tree to maintain the prefix), or we walk out of the lowest prefix node's sibling subtree. In this latter case, the complete previous tree was copied, and we may continue copying the next tree. In lines 8., 10., 15. and 17., we redefine the value of t as indicated by our observations.

Theorem 1. *Given an alphabet of symbols, Algorithm Enumerate enumerates exactly one ordered normal form for each unordered node-labeled tree that can be constructed using this alphabet.*

Proof. This follows from the lemmas. □

The overhead of this procedure is small. A datastructure is needed which efficiently stores the pre-order string and allows for a quick lookup of the rightmost path in the tree that is represented by the pre-order string. The additional constraints on rightmost path expansions can be checked in constant time. This shows that the problem of enumerating unordered trees is not much more complex than the problem of enumerating ordered trees.

Fig. 4. To make sure children are mapped injectively, a bipartite matching problem has to be solved between the sets of children. If $G(v, w)$ is solvable, we store pointers between the mapping (v, w) and the mappings between the children of v and w.

3 Tree counting

In the previous section we introduced an algorithm which enumerates all unordered pattern trees. In practice, this is impossible as the number of unordered trees is infinite given an alphabet of labels. A frequency criterion is used to stop further expansion of a tree. If a tree occurs less frequently than a certain predefined threshold in a large data tree, it is not expanded further, as every tree that could be constructed subsequently can only be less frequent. As the overhead of the enumeration technique is minimal, the performance of the algorithm largely depends on the speed with which the (in)frequency of a tree is determined.

We will first define the frequency of a pattern tree T. A node v in a pattern tree T can be mapped to a node w in a data tree iff v has the same label as w and there is an injective mapping from the children of v to the children of w. The frequency of a tree T is the number of nodes in the data tree to which the root of T can be mapped. Other frequency criteria built on partitioning can be determined in similar ways.

An easy — but not very efficient way — of determining the frequency is to determine for each pattern tree anew how many times it can be mapped to a data tree. An $O(nm^{1.5})$ algorithm (with n the number of nodes in the pattern tree and m the number of nodes in the data tree) exists for this task, as given in [4]. We take this algorithm as starting point for our counting strategy.

The first step of our algorithm is to determine for each label all locations in the database at which this label occurs. Those labels which fail to meet the predefined frequency criterion, are removed from further consideration.

An important task of the tree mapping algorithm is to determine that there is an *injective* mapping from 'pattern children' to 'database children'. Assume that a node v in a pattern tree and a node w with the same label in a data tree are given, and that each child of v can be mapped to one or more children of w, then the algorithm still has to make sure that an injective mapping can be obtained. The situation is clarified in Fig. 4. The mappings between children of v and children of w constitute a bipartite graph; to determine whether there is an injective mapping is a problem known as the *maximum bipartite matching problem*. The most efficient algorithm for this task has complexity $O(|E|\sqrt{|V|})$, where E is the set of edges in the bipartite graph and V is the set of vertices.

Algorithm Update
Input: a tree T in normal form, its associated mappings, and an expansion (d, l).
Output: an expanded tree T in which the mappings have been updated.

1. Let v be the node at depth $d-1$ on the rightmost path of the pattern tree.
2. Add a node v' at depth d with label l
3. **for all** $m \in \text{Map}(v)$ **do**
4. Let w be the node in the data tree to which m maps
5. $k_1, k_2 :=$ number of children of w (respectively v) with label l
6. **if** $k_1 - k_2 \geq 0$ **then**
7. **for all** children w' of w with label l **do**
8. Add to Map(v') a mapping from v' to w'
9. **else** Remove Mappings(m)

Procedure Remove Mappings
Input: a mapping m from a node v in T to a node w in a data tree
Output: a tree in which m is removed and the mappings of all nodes which are a child of the rightmost path are updated accordingly.

10. Let m' be parent(m), if v is not the root
11. Remove Mappings Below(m)
12. **if** v is not the root of T **then**
13. Let G be the (new) bipartite graph matching problem associated with m'
14. **if** G has no bipartite matching **then** Remove Mappings(m')

Procedure Remove Mappings Below
Input: a mapping m from a node v in T to a node w in a data tree
Output: a tree in which m is removed and the mappings of all nodes which are below v and are child of the rightmost path are updated accordingly.

15. **for all** children v' of v not on the rightmost path of T **do**
16. **for all** $m' \in \text{Map}(v', m)$ **do** Remove m' from Map(v')
17. Let v' be the child of v on the rightmost path of T
18. **for all** $m' \in \text{Map}(v', m)$ **do** Remove Mappings Below (m')
19. Remove m from Map(v)

Fig. 5. An algorithm for updating the datastructure that is associated to a pattern tree T.

With $G(v, w)$ we denote the bipartite graph that is involved in the determination of the injective child mapping of v to w. This graph contains all children of v and w, as well as all the mappings between these children. A node v can be mapped to a node w in the data tree iff there is an associated $G(v, w)$ for which a bipartite matching can be computed that maps each child of v to a different child of w.

The datastructure that is used by our counting algorithm has the following invariant property. Given a pattern tree, with each node v' that is a child of a node v on the rightmost path, we store exactly those mappings that are included in some solvable bipartite graph $G(v, w)$ that belongs to a mapping stored in the parent v; furthermore, we maintain pointers between each mapping of v' and the mapping in the parent to which this mapping is associated. For the root node we store all possible mappings to the data tree.

Fig. 6. Example of tree mappings, before and after expansion with $v_5 = (2, A)$.

The algorithm in Fig. 5 describes how the invariant is maintained when a tree T is expanded with a node (d, l). From the set of mappings of the parent, after this update, we can determine the frequency of the expanded tree.

In this algorithm, we use the following notation. With parent(v') we denote the parent of a node v' in a tree; Map(v) denotes all mappings stored for a given node v in the pattern tree. Given a node v' and a mapping $m \in$ Map(parent(v')), with Map(v',m) we denote those mappings in Map(v') which have a pointer to mapping m. If m is a mapping from v' to w', with parent(m) we denote the mapping from the parent of v' to the parent of w'.

We will briefly discuss some elements of the algorithm and illustrate these using the example of Fig. 6. In line 5.-6. we use the observation that a new node v' (in the example, v_5) can be mapped to every node which has the same label l (in the example, w_4, w_6 and w_9). In general, given a mapping $(v \rightarrow w) \in$ Map(parent(v')), the siblings of the new node could be mapped injectively to the children of w in the old situation. If the number of children of w with label l is larger than the number of children of v with label l, the added node can always be mapped to one of those additional nodes to solve the bipartite matching $G(v, w)$. If the number of children in the data tree is insufficient, such as in the example for $v_4 \rightarrow w_{10}$, the bipartite matching problem $G(v, w)$ can no longer be solved and the mapping associated to that matching problem must be removed. This mapping is in its turn part of some other matching problem of its parent node (in the example, $G(v_1, w_7)$). The parent's bipartite matching may no longer be solvable either. Therefore, it is necessary to recursively check that the matching problem for that ancestor node can still be solved (line 9.).

If some bipartite matching problem can no longer be solved, the corresponding mapping is removed. This mapping may have pointers to some mappings in the children ($v_1 \rightarrow w_7$ has pointers to $v_2 \rightarrow w_8$, $v_4 \rightarrow w_8$ and $v_4 \rightarrow w_{10}$); according to the definition of the invariant, these child mappings should also be removed, which is done by the Remove Mapping Below procedure.

The advantage of the invariant is that the number of (active) child mappings is kept very small. Still all those mappings are updated which are later required to determine the frequency of expanded trees. For this reason the procedure is also restricted to the rightmost path; the bipartite matching of other subtrees is not required to recompute all matchings after rightmost path expansion.

We propose to compute the frequent trees by traversing the enumeration tree in a depth-first fashion. The disadvantage of such a strategy is that it is difficult to apply some pruning strategies that are frequently used in Apriori-like

algorithms. The advantage is that the memory demand is much smaller; it is sufficient to store the mappings of the current tree only, together with information for undoing the removal of mappings when the enumeration backtracks over an expansion. For the latter purpose, with every expansion, we store a list of all the mappings that have been removed by that expansion. As every mapping can only be removed once, one can easily see that once a mapping is added to the pattern tree in line 8., information about this mapping is not removed from memory before the enumeration backtracks over the expansion to which this mapping belongs. The memory requirement of the algorithm is therefore still quite large: if n is the number of nodes in the largest frequent pattern tree, and m is the length of longest mapping list, the memory demand is of order $O(nm)$. In practice, of course, the memory footprint is much lower as large trees are not very frequent in most databases.

4 Conclusions and future research

In this paper we introduced an algorithm for mining frequent unordered induced subtrees. It extends the enumeration technique that was introduced in [1] and [6] for ordered subtrees. We showed that the enumeration of unordered subtrees is not much more difficult than the enumeration of ordered trees. The evaluation of unordered trees turns out to be more complex, from space as well as time complexity point of view. We propose to reduce the memory demand by a depth-first enumeration strategy, but this turns some pruning strategies difficult.

We envision many directions for future research. First of all, the validity of our approach should be verified experimentally. One can imagine several small optimizations to our algorithm which have not been discussed here; for example, one could also compute all solutions to a bipartite matching problem in stead of only computing one. Furthermore, we are also interested in ways to efficiently combine unordered trees with ordered trees to obtain a similar algorithm as the algorithm of Wang and Liu [5]; we are also interested in an algorithm for mining frequent unordered embedded subtrees.

References

1. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data. In: Proceedings of the 2nd Annual SIAM Symposium on Data Mining. (2002).
2. Agrawal, R., Manilla, H., Srikant, R., Toivonen, H., Verkamo, A.: Fast Discovery of Association Rules. In: U.M. Fayyad et al. (eds). Advances in Knowledge Discovery and Datamining. AAAI/MIT Press. (1996).
3. Nijssen, S., Kok, J.N.: Efficient Discovery of Frequent Unordered Trees: Proofs. Technical Report 2003-01, Leiden Institute of Advanced Computer Science. (2003).
4. Reyner, S.W.: An analysis of a good algorithm for the subtree problem. SIAM Journal of Computing, Vol. 6, No. 4. (1977).
5. Wang, K., Liu, H.: Discovering Structural Association of Semistructured Data. (1999).
6. Zaki, J.: Efficiently Mining Frequent Trees in a Forest. In: Proceedings of the SIGKDD'02, Edmonton, Canada (2002).

Expressivity versus Efficiency of Graph Kernels

Jan Ramon[1] and Thomas Gärtner[2,3]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Fraunhofer Institut Autonome Intelligente Systeme, Germany
[3] Department of Computer Science III, University of Bonn, Germany
jan.ramon@cs.kuleuven.ac.be
Thomas.Gaertner@ais.fraunhofer.de

Abstract. Recently, kernel methods have become a popular tool for machine learning and data mining. As most 'real-world' data is structured, research in kernel methods has begun investigating kernels for various kinds of structured data. One of the most widely used tools for modeling structured data are graphs. In this paper we study the trade-off between expressivity and efficiency of graph kernels. First, we motivate the need for this discussion by showing that fully general graph kernels can not even be approximated efficiently. We also discuss generalizations of graph kernels defined in literature and show that they are either not positive definite or not very useful. Finally, we propose a new graph kernel based on subtree patterns. We argue that while a little more computationally expensive, this kernel is more expressive than kernels based on walks.

1 Introduction

Support vector machines [1] are among the most successful recent developments within the machine learning community. Along with some other learning algorithms they form the class of kernel methods [10]. The computational attractiveness of kernel methods is due to the fact that they can be applied in high dimensional feature spaces without suffering from the high cost of explicitly computing the feature map. This is possible by using a positive definite kernel k on any set $\mathcal{X}$. For such $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ it is known that a map $\phi : \mathcal{X} \to \mathcal{H}$ into a Hilbert space $\mathcal{H}$ exists, such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for all $x, x' \in \mathcal{X}$.

Kernel methods have so far successfully been applied to various tasks in attribute-value learning. Much 'real-world' data, however, is structured – there is no natural representation of the instances of the learning problem as a tuple of constants. In computer science graphs are a widely used tool for modeling structured data. They can be used, for example, as a representation for molecules.

Unfortunately, due to the powerful expressiveness of graphs, defining appropriate kernel functions for graphs has proven difficult. In order to control the complexity of such kernels, one line of existing research has concentrated on special kinds of graphs, in particular, trees [2] or strings [11, 9] which results in efficient kernels, but loses most of the power of general graphs. Others [4, 7] have investigated efficient kernels for general graphs based on particular kinds of walks, which captures more, but still far from all of the structure of the graph.

More recently [5] answered the questions whether it is possible to define kernels that take the entire structure of graphs into account. While those kernels can be defined, computing them is hard. For that reason, alternative graph kernels based on common walks are investigated that, for example, allow for gaps in the label sequences corresponding to the walks.

In this paper we show several other results concerning graph kernels. Complete graph kernels are those graph kernels that distinguish between two graphs if and only if they are not isomorphic. It is known that computing complete graph kernels is at least as hard as deciding whether two graphs are isomorphic. In this paper we will show that even approximating k with a constant bound on the approximation-error is as hard as deciding whether two graphs are isomorphic. Also we review the basic ideas of previous work on graph kernels and show that these ideas can not directly be generalized to more expressive graph kernels. We show this by demonstrating that straight-forward generalizations do either lead to non positive definite graph kernels or to trivial feature spaces. These results motivate the search for other, more expressive graph kernels.

The remainder of this paper is structured as follows: In Section 2 we review some basic definitions. In Section 3 we show that complete graph kernels can not be efficient. In Section 4 we discuss a general framework for graph kernels based on common subgraphs, and discuss consequences of generalizing previously studied graph kernels. In Section 5 we propose a new instantiation of this framework, a graph kernel that uses the count of subtrees as features. Finally, in Section 6 we give conclusions and directions for further work.

2 Graphs

We first review a few basic definitions and introduce some notations that will be used in the sequel of this paper. For a more in-depth discussion of graphs and related concepts the reader is referred to [3, 8].

Labeled directed graphs Generally, a *graph* G is described by a finite set of *vertices* $\mathcal{V}$, a finite set of *edges* $\mathcal{E}$. Therefore, a graph is commonly denoted $G(\mathcal{V}, \mathcal{E})$. For *labeled* graphs there is additionally a set of labels $\mathcal{L}$ along with a function *label* assigning a label to each edge and vertex. For unlabeled graphs, we will assume $label(v)$ has the same value for all vertices. We will denote the space of all graphs with $\mathcal{G}$. We will sometimes assume some enumeration of the vertices and labels in a graph, i.e., $\mathcal{V} = \{\nu_i\}_{i=1}^n$ where $n = |\mathcal{V}|$ and $\mathcal{L} = \{\ell_r\}_{r \in \mathbb{N}}$ [1]. For *undirected* graphs, each edge is a set containing two vertices. For *directed* graphs without parallel edges each edge is a tuple consisting of the initial and terminal vertex of the edge $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Edges (v, v) in a directed graph are called *loops*.

[1] While ℓ_1 will be used to always denote the same label, l_1 is a variable that can take different values, e.g., $\ell_1, \ell_2, \ldots$. The same holds for vertex ν_1 and variable v_1.

Some special graphs, relevant for the description of graph kernels are walks, paths, cycles, trees and forests. A *walk*[2] w of a graph $G(\mathcal{V}, \mathcal{E})$ is a sequence of vertices $w = v_1, v_2, \ldots v_{n+1}$; $v_i \in \mathcal{V}$ such that $(v_i, v_{i+1}) \in \mathcal{E}$. The *length* of the walk is equal to the number of edges in this sequence, i.e., n in the above case. A *path* is a walk in which $v_i \neq v_j \Leftrightarrow i \neq j$. A *cycle* is a path with $(v_{n+1}, v_1) \in \mathcal{E}$.

A graph $G = (\mathcal{V}, \mathcal{E})$ is called *connected* if there is a walk between any two vertices in the following graph: $(\mathcal{V},\ \mathcal{E} \cup \{(u, v) : (v, u) \in \mathcal{E}\})$ For a graph $G = (\mathcal{V}(G), \mathcal{E}(G))$, we denote by $G[\mathcal{V}^*]$ the subgraph *induced* by the set of vertices $\mathcal{V}^* \subseteq \mathcal{V}(G)$, that is $G[\mathcal{V}^*] = (\mathcal{V}^*, \{(u, v) \in \mathcal{E}(G) : u, v \in \mathcal{V}^*\})$. A *subgraph* of G is a graph $H = (\mathcal{V}(H), \mathcal{E}(H))$ with $\mathcal{V}(H) \subseteq \mathcal{V}(G)$ and $\mathcal{E}(H) \subseteq \mathcal{E}(G[\mathcal{V}(H)])$. A graph is *acyclic* if no subgraph of a graph is a cycle. A *subforest* is an acyclic subgraph; a *subtree* is an connected subforest. We denote the set of all graphs by $\mathcal{G}$.

A graph is isomorphic to another graph if there is an edge (and label) preserving bijection between all vertices in one graphs and all vertices in the other graph. A graph is homomorphic to another graph if there is an edge (and label) preserving surjection between all vertices in one graphs and all vertices in the other graph.

We also need to define some functions describing the neighborhood of a vertex v in a graph $G(\mathcal{V}, \mathcal{E})$: $\delta^+(v) = \{u : (v, u) \in \mathcal{E}\}$ and $\delta^-(v) = \{u : (u, v) \in \mathcal{E}\}$. Here, $|\delta^+(v)|$ is called the *outdegree* of a vertex and $|\delta^-(v)|$ the *indegree*.

Product Graphs Product graphs [6] are a very interesting tool in discrete mathematics. The four most important graph products are the Cartesian, the strong, the direct, and the lexicographic product. While the most fundamental one is the Cartesian graph product, in our context the direct graph product is the most important ones.

Usually, graph products are defined on unlabeled graphs. However, in many real-world machine learning problems it could be important to be able to deal with labeled graphs. Here is the definition of the direct product graph of two labeled graphs as given in [5]

We denote the direct product of two graphs $G_1 = (\mathcal{V}_1, \mathcal{E}_1), G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ by $G_1 \times G_2$. The vertex and edge set of the direct product are respectively defined as:

$$\begin{aligned}
\mathcal{V}(G_1 \times G_2) &= \{(v_1, v_2) \in \mathcal{V}_1 \times \mathcal{V}_2 : (\mathit{label}(v_1) = \mathit{label}(v_2))\} \\
\mathcal{E}(G_1 \times G_2) &= \{((u_1, u_2), (v_1, v_2)) \in \mathcal{V}^2(G_1 \times G_2) : \\
&\qquad (u_1, v_1) \in \mathcal{E}_1 \wedge (u_2, v_2) \in \mathcal{E}_2 \wedge (\mathit{label}(u_1, v_1) = \mathit{label}(u_2, v_2))\}
\end{aligned}$$

A vertex (edge) in graph $G_1 \times G_2$ has the same label as the corresponding vertices (edges) in G_1 and G_2. The graphs G_1, G_2 are called the factors of $G_1 \times G_2$.

[2] What we call 'walk' is sometimes called an 'edge progression'.

3 Complete Graph Kernels

Given some set, there are many possible ways to define positive definite kernels on it. The optimal choice not only depends on the structure of the data but also on the concept class and the data itself. The concept class is the class of all relevant concepts. The concept that is used to determine the class of examples belongs to the concept class but is unknown to the learning system. There exists a wide range of learning theory results putting bounds on the number of examples needed to learn certain concept classes assuming that a kernel evaluation can be performed in unit time. However, for graphs this is not evident. In this section, we show this by proving that kernels that approximate kernels that map graphs on separated points in feature space, are necessarily hard to compute. First, we need to define positive definite kernel, complete graph kernel, and distance induced by a kernel. A thorough discussion on kernels and kernel based learning is given in [10].

Positive Definite Kernel Let $\mathcal{X}$ be a set. A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *positive definite kernel* on $\mathcal{X}$ if, for all $n \in \mathbb{Z}^+$, $x_1, \ldots, x_n \in \mathcal{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; k(x_i, x_j) \geq 0$.

Complete Kernel Let $\Phi : \mathcal{G} \to \mathcal{H}$ be a map from this set into a Hilbert space $\mathcal{H}$. Furthermore, let $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ be such that $\langle \Phi(G), \Phi(G') \rangle = k(G, G')$. If Φ is , k is called a *complete* graph kernel.

Induced Distance Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a positive definite kernel on any set $\mathcal{X}$ and let $x, y \in \mathcal{X}$. Then, $d_k(x, y) = \sqrt{k(x, x) - 2k(x, y) + k(y, y)}$ is the distance induced by k. Note that for the map $\phi : \mathcal{X} \to \mathcal{H}$ such that $\langle \phi(x), \phi(y) \rangle = k(x, y)$ for all x, y, it holds that $d_k(x, y) = \|\phi(x) - \phi(y)\|$. If k is complete, d_k is a distance, otherwise d_k is a pseudo-distance.

A proposition similar to the following result has been presented in [5]

Proposition 1. *Let $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ be a complete graph kernel. If there is an algorithm such that the time needed to compute $k(G_1, G_2)$ for any graphs $G_1(\mathcal{V}_1, \mathcal{E}_1), G_2(\mathcal{V}_2, \mathcal{E}_2) \in \mathcal{G}$ is bounded by a function $f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|)$, then there is an algorithm that decides whether two graphs G_1 and G_2 are isomorphic and runs in time $\mathcal{O}(f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|))$.*

Proof. Let ϕ be the map corresponding to the complete kernel k. As ϕ is injective,

$$k(G, G) - 2k(G, G') + k(G', G') = \langle \phi(G) - \phi(G'), \phi(G) - \phi(G') \rangle = 0$$

if and only if G, G' are isomorphic. Hence, to decide whether G_1 and G_2 are isomorphic, it is sufficient to evaluate the three kernel expressions $k(G_1, G_1)$, $k(G_1, G_2)$ and $k(G_2, G_2)$ and do a simple addition, which can clearly be done in time $\mathcal{O}(f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|))$ □

As deciding graph isomorphism is known to be hard and the previous theorem shows that evaluating a complete kernel is equally hard, one can conclude that no efficiently computable complete kernel on $\mathcal{G}$ exists. We can even prove a stronger result: it is impossible to define an efficiently computable function that approximates closely a complete kernel.

Proposition 2. *Let $k : \mathcal{G} \times \mathcal{G} \to \mathbb{R}$ be a complete graph kernel such that for any two non-isomorphic graphs G_1 and G_2, $d_k(G_1, G_2) \geq d$. Let k' be a function that approximates k such that $\forall x, y : |k(x, y) - k'(x, y)| < d/8$. Then, if there is an algorithm such that the time needed to compute $K(G_1, G_2)$ for any graphs $G_1, G_2 \in \mathcal{G}$ is bounded by a function $f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|)$, then there is an algorithm that decides whether two graphs G_1 and G_2 are isomorphic and runs in time $\mathcal{O}(f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|))$.*

Proof. Assume that such a k' exists. Then, we could for any two graphs G_1 and G_2 compute $d_{k'}(G_1, G_2)$ in time $\mathcal{O}(f(|\mathcal{V}_1|, |\mathcal{E}_1|, |\mathcal{V}_2|, |\mathcal{E}_2|))$. We then have

$$\begin{aligned}
&d_{k'}(G_1, G_2) - d_k(G_1, G_2) \\
&= \big(k'(x, x) - 2k'(x, y) + k'(y, y)\big) - \big(k(x, x) - 2k(x, y) + k(y, y)\big) \\
&< d/2.
\end{aligned}$$

Therefore, it is sufficient to compare $d_{k'}(G_1, G_2)$ to $d/2$ to decide whether the graphs are isomorphic or not: if $d_{k'}(G_1, G_2)$ is smaller than $d/2$, $d_k(G_1, G_2)$ can not be larger than d and the graph are isomorphic while if it is larger than $d/2$, the distance induced by k between the graphs can not be zero. □

This theorem says in fact that there is no efficiently computable function that approximates a complete kernel sufficiently well to be able to distinguish between near (in feature space) but non-isomorphic graphs. Apart from the hardness of graph isomorphism, one can apply other computational complexity results on graphs to kernels. E.g. one can not expect to be able to define graphs kernels to learn efficiently concepts which are known to be hard to compute (such as containing subgraphs).

4 Graph Kernels based on Common Subgraphs

In this section we first briefly introduce the general idea behind previous approaches to define efficient graph kernels. Then we present you results indicating that straight-forward generalizations of these ideas do not result in positive definite functions. This motivates our search for other graph kernels, presented in the next section.

We will consider only those subgraphs that do not contain isolated vertices. To be able to give different weights to different sizes of subgraphs we assume a sequence $\lambda_0, \lambda_1, \ldots$ of weights with $\lambda_i > 0$.

4.1 General Approach

The general idea of graph kernels defined so far is to measure common subgraphs of two graphs. Conceptually, the feature map ϕ of the graph kernel k with $k(G, G') = \langle \phi(G), \phi(G') \rangle$ has one feature ϕ_h for every graph h from some given set H.

The graph kernel mentioned in the previous section can then be described by:

$$\Phi_h(G) = \sqrt{\lambda_{|\mathcal{E}(h)|}} \Big| \{g \text{ is subgraph of } G : h \text{ is isomorphic to } g\} \Big| \tag{1}$$

As computing this kernel is *NP*-hard, alternative kernels are based on homomorphism

$$\phi_h(G) = \sqrt{\lambda_{|\mathcal{E}(h)|}} \Big| \{g \text{ is subgraph of } G : h \text{ is homomorphic to } g\} \Big| \tag{2}$$

4.2 Common Walks

In [4, 7, 5] kernels are considered where H is the set of paths. The kernel between two graphs can then efficiently be computed as a matrix power series of the adjacency matrix of their direct product graph. This principle can be formulated as follows:

$$f_H(G, G') = \sum_{h \in H} \lambda_{|\mathcal{E}(h)|} \Big| \{g \text{ is subgraph of } G \times G' : h \simeq g\} \Big| \tag{3}$$

To be precise, not all kernels considered in [4, 7, 5] fit directly in this framework. This is because for some kernels the cardinality of the set of walks is replaced by the probability of observing a random walk with a label sequence corresponding to this walk; for other kernels only some particular labels along the walks are compared and not all. Both modifications are, however, not conceptually different from this framework.

The computation of f_H does not necessarily require to check graph isomorphism, as the summation can be over all subgraphs of $G \times G'$ that satisfy some property. That property must be the characteristic property of H. When counting common walks (H is the set of walks) f_H is a positive definite kernel function corresponding to the inner product under the map $\phi_h, h \in H$ [5]. For appropriate weight sequences f_H can be computed in cubic time.

4.3 Common Paths, Trees, and Forests

In this section we consider functions f_H where H is some set of acyclic graphs, either all paths, all trees, or all forests. We call these functions a subpath, subtree, and subforest function, respectively.

Proposition 3. *The subpath, subtree, and subforest functions are not positive definite.*

Proof. Let G be a single loop and let G' be a graph with two vertices and a single edge connecting these vertices. Then $G \times G$ and G are isomorphic, as well as $G' \times G'$, $G \times G'$, and G'. In G the only path is the trivial path consisting of just one vertex. G' has one edge, so it has three paths: two trivial ones consisting of one vertex and one consisting of the two vertices and the edge connecting them. Hence,

$$k(G,G) - 2k(G,G') + k(G',G') = \lambda_0 - 2(2\lambda_0 + \lambda_1) + 2\lambda_0 + \lambda_1 = -\lambda_0 - \lambda_1 < 0$$

□

4.4 Common Graphs

In this section we consider the function f_H where H is the set of all graphs and μ is the set cardinality. We call this function the subgraph function.

Proposition 4. *The subgraph function is positive definite.*

Proof. Let G, G' be any two graphs. The number of subgraphs of their product graph is $2^{|\mathcal{E}(G \times G')|}$. The number of edges of a given label in the product graph is simply the product of the number of edges of that label in each factor. As kernels are closed under products and as $a^x, a \geq 1$ can be written as the limit of a polynomial series with positive coefficients, the subgraph function is positive definite. □

Although the subgraph function is positive definite, it is not likely to be useful in practice, as it does not take the structure of the graphs into account

5 Tree-structured Pattern Kernels

In section 3, we argued that generally usable graph kernels can not even be approximated efficiently. All graph kernels investigated in literature so far are conceptually based on some measure of the walks in two graphs that have some or all labels in common. Computation of these kernels is made possible by using the direct product graph and computing a closed form of the limit of a matrix power series of the adjacency matrix of the product graph. For such kernels, one can easily find pairs of graphs which are mapped to the same point in feature space, e.g. Figure 1 shows two directed graphs and Figure 2 shows two undirected graphs whose images in the Hilbert space are the same. Such graphs can not be distinguished by any kernel machine using a walk-based graph kernel.

In section 4 we then showed that functions based on counting paths, trees, and forests in the product graph are not positive definite. In this section we will push the limit of efficiently computable graph kernels by describing a method for counting the number of common *subtree patterns* in two graphs. Roughly, the subtree patterns we consider are rooted subgraphs such that there is a tree homomorphic to the subgraph, and the number of distinct children of both root nodes in the pattern and tree are the same.

Formally, let $G(\mathcal{V}, \mathcal{E}) \in \mathcal{G}$ be a graph. If $r \in \mathcal{V}$, then r is a *subtree pattern* of G *rooted* at r. If $t_1, t_2, \ldots, t_n$ are subtree patterns of G rooted at respectively $r_1, r_2, \ldots, r_n$ (with all r_i different), and if $(r, r_1), (r, r_2), \ldots (r, r_n) \in \mathcal{E}$, then $r(t_1, t_2, \ldots, t_n)$ is a subtree pattern of G rooted at r. r is also called the parent node of the nodes r_i of the nodes r_i of the subtree pattern.

Fig. 1. Directed graphs mapped to the same point in walks feature space

Fig. 2. Undirected graphs mapped to the same point in walks feature space

Every subtree pattern has a tree-structured signature just as every walk had a signature represented by the sequence of labels of the vertices in the sequence. So for each possible subtree pattern signature, we associate a feature whose value is the number of times that a subtree of that signature occurs in the graph.

Let $G_1(\mathcal{V}_1, \mathcal{E}_1), G_2(\mathcal{V}_2, \mathcal{E}_2) \in \mathcal{G}$ be two graphs. We will denote the weighted count of pairs of subtrees of the same signature of height less than or equal to h, with the first one rooted at $r \in \mathcal{V}(G_1)$ and the second one rooted at $s \in \mathcal{V}(G_2)$ with $k_{r,s,h}$. Now, if $h = 1$ and $\mathit{label}(s) = \mathit{label}(r)$ we have $k_{r,s,h} = 1$. If $h = 1$ and $\mathit{label}(s) \neq \mathit{label}(r)$ we have $k_{r,s,h} = 0$. For $h > 1$, one can compute $k_{r,s,h}$ as follows:

- Let $M_{r,s}$ be the set of all matchings from $\delta^+(r)$ to $\delta^+(s)$, i.e.

$$M_{r,s} = \Big\{ R \subseteq \delta^+(r) \times \delta^+(s) \mid \big(\forall (a,b),(c,d) \in R : a = c \Leftrightarrow b = d\big) \\ \wedge \big(\forall (a,b) \in R : \mathit{label}(a) = \mathit{label}(b)\big) \Big\}$$

– Compute

$$k_{r,s,h} = \lambda_r \lambda_s \sum_{R \in M_{r,s}} \prod_{(r',s') \in R} k_{r',s',h-1}$$

Here, λ_r and λ_s are positive values smaller than 1 to cause higher trees to have a smaller weight in the overall sum.

Given two graphs $G_1(\mathcal{V}_1, \mathcal{E}_1), G_2(\mathcal{V}_2, \mathcal{E}_2) \in \mathcal{G}$, we can then define the subtree-pattern kernel of G_1 and G_2 by

$$k_{tree,h}(G_1, G_2) = \sum_{r \in V_1} \sum_{s \in V_2} k_{r,s,h}.$$

Proposition 5. *$k_{tree,h}$ is a positive definite kernel.*

Proof. The feature space induced by this kernel contains one feature for each subtree pattern signature. The value of such a feature is the sum of the weights of all its occurrences in the graph. An occurrence of a subtree pattern $r(t_1, \ldots, t_n)$ has weight $weight(r(t_1, \ldots, t_n)) = \lambda_r \sum_i weight(t_i)$ and the weight of a trivial subtree pattern r is 1. We have defined the feature space explicitely, $k_{tree,h}$ is now equal to the inner product in this space and hence $k_{tree,h}$ is positive definite. □

One can also define

$$k_{tree}(G_1, G_2) = \lim_{h \to \infty} k_{tree,h}(G_1, G_2).$$

For suitable λ_r, λ_s (i.e. causing the sum to remain bounded for increasing values of h) this limit will converge. Since $k_{tree,h}$ is already a kernel for every h, in practice $k_{tree,h}$ can be used and for sufficiently large h it will be a good approximation of k_{tree}.

The graphs in Figure 2 are not equal for this kernel k_{tree}. The feature space of k_{tree} includes the feature space with all walks, but also includes features such as the subtree pattern $a(b, b, b)$ which has value 0 for G_1 and value 1 for G_2. The computational complexity is higher than the complexity for kernels based on walks. This is mainly due to the summation over $M_{r,s}$. However, if the graph is not too connected or if there is sufficient diversity in the labels of the vertices, this extra cost will be fairly low.

6 Conclusions and Further Work

In this paper we discussed several aspects of graph kernels and the trade-off between efficiency and expressivity. We improved a result that shows that no efficiently complete graph kernels exist. We presented a framework for kernels based on common subgraphs and argued that many existing kernels fit into this framework. We argued that kernels using linear patterns to define feature will not suffice in all practical cases and proposed a tree-pattern based graph kernel which

has a strictly larger feature space. We also pointed out that some kernels that count weighted patterns which do not allow vertices to repeat (walks, subtrees and forests) in the product graph are not positive definite and often hard to compute and that using all subgraphs (without weighting) results in a kernel which is not very useful.

There are several possible directions for further work. First, there exists much work on learnability results for related representation languages such as inductive logic programming. Adapting these results to use graphs as representations could give more insight on efficiently learnable classes of graph concepts and in particular provide hints at classes for which efficient kernels could exist. Second, as many different kernels on graphs are proposed, it would be useful to compare them in more detail. One possible direction is to empirically evaluate on which kind of applications which kernel performs better.

Acknowledgments Jan Ramon is a post-doctoral fellow of the Katholieke Universiteit Leuven. Research supported in part by the EU Framework V project (IST-1999-11495) *Data Mining and Decision Support for Business Competitiveness: Solomon Virtual Enterprise* and by the DFG project (WR 40/2-1) *Hybride Methoden und Systemarchitekturen für heterogene Informationsräume.* The authors thank Peter Flach, Tamás Horváth, Stefan Kramer and Stefan Wrobel.

References

1. B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In David Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, July 1992.
2. M. Collins and N. Duffy. Convolution kernels for natural language. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.
3. R. Diestel. *Graph Theory.* Springer-Verlag, 2000.
4. T. Gärtner. Exponential and geometric kernels for graphs. In *NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data*, 2002.
5. T. Gärtner, P. A. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and the 7th Kernel Workshop*, 2003.
6. W. Imrich and S. Klavžar. *Product Graphs: Structure and Recognition.* John Wiley, 2000.
7. H. Kashima and A. Inokuchi. Kernels for graph classification. In *ICDM Workshop on Active Mining*, 2002.
8. B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms.* Springer-Verlag, 2002.
9. H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2, 2002.
10. B. Schölkopf and A. J. Smola. *Learning with Kernels.* MIT Press, 2002.
11. C. Watkins. Kernels from matching operations. Technical report, Department of Computer Science, Royal Holloway, University of London, 1999.

Distance-based Clustering of XML Documents*

Francesco De Francesca, Gianluca Gordano,
Riccardo Ortale, and Andrea Tagarelli

DEIS, University of Calabria, 87036 Rende (CS) – Italy
e-mail: {ortale, tagarelli}@si.deis.unical.it

The increasing relevance of the Web as a mean for sharing information around the world has posed several new interesting issues to the computer science research community. The traditional approaches to information handling are ineffective in the new context: they are mainly devoted to the management of highly structured information, like relational databases, whereas Web data are semistructured and encoded using different formats (HTML, XML, and so on).

In such context, we address the problem of clustering structurally similar Web documents, and in particular XML documents. This problem has several interesting applications, related, e.g., to the management of Web data. For example, the detection of structural similarities among documents can help in solving the problem of recognizing different sources providing the same kind of information [2], or in the structural analysis of a Web site.

In this paper we propose a novel methodology for clustering XML documents, focusing on the notion of *XML cluster representative*, i.e., a prototype XML document subsuming the most relevant features of the set of XML documents within the cluster. In particular, we devise a technique to compute a representative of a set of XML documents, which is capable of capturing all the structural specificities within the represented documents. To this purpose, the notion of *structural matching* between the trees associated to two XML documents is exploited. Structural matchings allow to both identify the structural similarities between two XML documents and to build a representative around these similarities. We also investigate the exploitation of *merging* and *pruning* strategies for refining XML document trees into effective cluster representatives.

1 Preliminaries

Depending on the specific application domain, the notion of tree matching can be defined in a variety of ways. Here it is exploited in the context of XML trees, i.e. trees resulting from a hierarchy of XML tags, to the purpose of highlighting *common skeletons*. Given two XML documents, a common skeleton is a substructure belonging to both the XML trees: precisely, it is defined as a collection of tags which all exhibit the same name, depth level and parent node. Some definitions at the basis of our approach are provided next.

A tree t is a tuple $t = \langle r_t, V_t, E_t, \delta_t \rangle$ where $V_t \subseteq \mathbb{N}$ is the set of nodes, $E_t \subseteq V_t \times V_t$ is the set of edges, r_t is the root node of t, and $\delta_t : V_t \mapsto \Sigma$ is a

* This work was partially supported by the National Research Council project SP2: "Strumenti, ambienti e applicazioni innovative per la società dell'informazione - Legge 449/97-99".

node labelling function where Σ is an alphabet of node labels. In addition, we denote by $depth_t(v)$ the depth of a node v in t.

Definition 1 (meaningful matching). *Given two XML trees t_1 and t_2, and $v \in V_{t_1}$, $w \in V_{t_2}$, a* meaningful matching *(henceforth called $m(v, w)$) holds if the following conditions inductively hold: $v = r_{t_1}$, $w = r_{t_2}$ and $\delta_{t_1}(v) = \delta_{t_2}(w)$; otherwise, $\delta_{t_1}(v) = \delta_{t_2}(w)$, $depth_{t_1}(v) = depth_{t_2}(w)$ and $m(a, b)$ holds, where $(a, v) \in E_{t_1}$ and $(b, w) \in E_{t_2}$.*

Note that, in general, multiple matchings may occur when a node in a tree has a meaningful matching with more than one node of a different tree. Formally, given two trees t_1 and t_2, a node $v \in V_{t_1}$ has a *multiple matching* if $\exists w', w'' \in V_{t_2}$ s. t. both $m(v, w')$ and $m(v, w'')$ hold.

The definition below introduces a criterion exploited to overcome the multiple matching problem. The idea is avoiding to take into account those meaningful matchings which are not effectively indicative of strong structural similarities. The relevance of a meaningful matching is evaluated through the weighting function $w_m : V \times V \mapsto \mathbb{N}$, for a set V of nodes.

Definition 2 (match weighting function). *Given two nodes $v \in V_{t_1}$ and $w \in V_{t_2}$, the weight associated with $m(v, w)$ is computed as: $w_m(v, w) = 0$ if $\delta_{t_1}(v) \neq \delta_{t_2}(w)$, $w_m(v, w) = 1 + sum_{v,w}$ otherwise.*

where $sum_{v,w}$ is recursively defined as the sum of the relevance degrees associated to the best matchings between the descendants of v and w. Function w_m weights the prominence of a meaningful matching in such a way that the higher its value, the more relevant the matching itself.

The set of meaningful matchings represent the common paths between two trees. A tree containing only these common paths is defined as a *matching tree*, since it resembles the intersection of the original trees.

Definition 3 (matching tree). *A tree t is a* matching tree *(henceforth called t_m) for two trees t_1, t_2 if $\forall v \in V_t$, $\exists! v_1 \in V_{t_1}$ and $\exists! v_2 \in V_{t_2}$ s. t. $m(v_1, v_2)$ holds. A matching tree t_m is* optimal *if its size is maximal, i.e., if for each matching tree $u_m \neq t_m$, we have $|t_m| \geq |u_m|$.*

2 Problem Statement

Suitable clustering algorithms for semistructured documents were extensively studied in the current literature [5]. Hierarchical methods are widely known as providing clusters with a better quality, especially for text datasets [1, 6]. Our *XRep* algorithm is an adaptation of the agglomerative hierarchical algorithm to our problem. Initially each XML tree is placed in its own cluster, and a matrix containing the pair-wise tree distance is computed. Next, the algorithm walks into an iterative step in which the least dissimilar clusters (evaluated on the basis of their cluster representatives) are merged. As a consequence, the distance matrix is updated to reflect this merge operation. The overall process is stopped when an optimal partition is reached.

We address the problem of clustering XML documents in a parametric way w.r.t. the concepts of distance measure and cluster representative. Viewed in

this respect, we need to investigate the conditions for the optimality of a cluster representative, once a suitable distance measure is given.

The notion of proximity, between two patterns drawn from the same feature space, is essential to the definition of a cluster. We mainly focus on suitably adapting a distance measure originally conceived to deal with strings, namely *Edit distance*. Edit distance between trees is computed as the minimum-cost sequence of operations required to convert one given tree to another [7, 4].

Intuitively, a *representative* of a cluster of XML documents is an XML document which effectively reflects all the structural contents within the cluster. The core of the *XRep* algorithm is the computation of the XML cluster representative through three main steps: *i*) (*Tree matching*) first, the information related to all meaningful matchings between the XML trees are suitably combined in order to build an optimal matching tree; *ii*) (*Tree merging*) the optimal matching tree is successively grown into a merge tree by adding to it the uncommon substructures within the original trees; *iii*) (*Pruning of merge tree*) finally, the merge tree is pruned according to a strategy which aims at removing the least frequent nodes: this allows to obtain a representative as an XML tree.

3 Cluster Representative

3.1 XML tree matching

We propose a dynamic-programming algorithm for building the optimal matching tree from two XML trees. The idea is that, at each level, all meaningful matchings are detected and weighted (by means of function w_m). Weights take into account the relevance of meaningful matchings and, as a consequence, allow to overcome multiple matchings. The bottom-up approach to the construction of the optimal matching tree consists of two steps: *i*) matching matrix computation, *ii*) removal of multiple matchings.

Matching matrix computation. Given two XML trees $t_1 = \langle r_{t_1}, V_{t_1}, E_{t_1}, \delta_{t_1} \rangle$ and $t_2 = \langle r_{t_2}, V_{t_2}, E_{t_2}, \delta_{t_2} \rangle$, all meaningful matchings among nodes at the same level within both trees are captured by a matching matrix M_m. M_m has $|V_{t_1}|$ rows and $|V_{t_2}|$ columns. For each level k, a sub-matrix $M_m(k)$ collects the meaningful matchings among the sets of $|V_{t_1}(k)|$ and $|V_{t_2}(k)|$ nodes at level k respectively belonging to t_1 and t_2. The computation starts from the last level of the XML trees and iterates until the roots r_{t_1} and r_{t_2} are reached. At the generic iteration, it verifies if there are nodes, at the same level in both t_1 and t_2, which exhibit meaningful matchings. In these cases, the weight corresponding to each such meaningful matching becomes the value of the entry associated to the matching nodes.

Removal of multiple matchings. A multiple matching is denoted by the presence of at least two non-zero values in some row and/or column of M_m. A node $v_i \in V_{t_1}$, which corresponds to the i-th row in M_m, can match with a number of nodes belonging to t_2. Let $J_{v_i} = \{j_1, \ldots, j_h\}$ be the set of column indexes corresponding to these nodes. Formally, v_i exhibits a multiple matching if $|J_{v_i}| > 1$. Multiple matchings can be eliminated by simply choosing the best matching for each node $v_i \in V_{t_1}$: such a matching corresponds to the column

index $j^*_{v_i} = \arg\max_{j_1,\ldots,j_h}\{M_m(i,j_1),\ldots,M_m(i,j_h)\}$. This technique allows the construction of a *marking vector* $V_m = \{j^*_{v_1},\ldots,j^*_{v_n}\}$, whose generic i-th entry indicates the best matching node in t_2 for v_i (both nodes are at the same level).

An optimal matching tree is efficiently built from vector V_m: it suffices that all nodes $v_i \in V_{t_1}$ such that $V_m[i] = -1$ are removed from t_1. These nodes exhibit no meaningful matchings and, as a consequence, are not taken into account.

3.2 Merging of XML trees

This step aims at building an approximation of the actual cluster representative. Starting from t_2 and the above vector V_m, a merge tree $t_{1,2}$ is suitably formed in order to include nodes which reveal to be either common or uncommon to both trees t_1 and t_2: each node is taken into account at most once. Given a node v_i in t_1 with no meaningful matchings, if the parent node of v_i matches with a node $w_j \in V_{t_2}$, then v_i appears in $t_{1,2}$ as a child of w_j.

Both the techniques for building an optimal matching tree and a merge tree are conceived to work with only two XML documents. We show that this does not determine a loss of generality, since merge trees are associative and, as a consequence, can be associated even to clusters consisting of more than two XML documents. A proof of the associativity of the merging process w.r.t. Edit distance is illustrated in [3].

3.3 Representative computation

Pruning is at the basis of the refining process which turns a merge tree into an effective cluster representative. Precisely, leaf nodes are inspected for removal from the merge tree one at a time. Any performed cut minimizes the distance between the refined merge tree and the original document trees in the cluster. The removal of a leaf node from the refined merge tree is reiterated until its distance from the original cluster trees cannot be further decreased. The pruning technique for building a cluster representative is parametric w.r.t. the notion of distance.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press Books. Addison Wesley, 1999.
2. S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.
3. F. De Francesca, G. Gordano, R. Ortale, and A. Tagarelli. A general framework for XML document clustering. Technical report, n.8, ICAR-CNR, 2003.
4. Carsten Isert. The editing distance between trees. Technical report, Ferienakademie, for course 2: Bume: Algorithmik Und Kombinatorik, Italy, 1999.
5. A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
6. M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. In *Proc. ACM SIGKDD Workshop on Text Mining*, 2000.
7. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

Graph Mining Approaches for the Discovery of Web Communities

Tsuyoshi Murata[1,2]

[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan
tmurata@nii.ac.jp
http://research.nii.ac.jp/~tmurata
[2] Japan Science and Technology Corporation
Yoyogi Community Building, 1-11-2, Yoyogi, Shibuya-ku,
Tokyo, 151-0053 Japan

Abstract. Finding related Web pages is important for assisting users' information retrieval from the Web. In general, related Web pages are densely connected with each other by hyperlinks, and graph mining approaches are applicable for discovering such clusters of related Web pages, which are called Web communities. Among the research of Web structure mining based on the graph structure of hyperlinks, discovery of Web communities is one of the important research topics. In this paper, recent approaches for the discovery of Web communities are introduced, and requirements for graph mining algorithms suitable for the discovery of Web communities are discussed.

1 Introduction

At the time the author is writing this paper, Google indexes more than 3 billion Web pages in the world. The goal of Web mining is to utilize this huge Web network. The Web can be regarded as a graph if we regard each Web page as a vertex and each hyperlink as an edge. Web structure mining is based on the graph structure of hyperlinks, and is one of the important research topics that graph mining algorithms are really required. There are several goals for Web structure mining, such as ranking important Web pages [4][8], discovery of Web communities [3][5], analysis of the Web graph from macroscopic point of view [2], and modeling and simulating the process of Web graph generation [1]. Among these, discovery of Web communities (clusters of related Web pages whose hyperlinks are densely connected) is important in order to assist users' information retrieval from the Web. However, applying graph mining algorithms to the Web is not simple since it is huge and is growing. There are some requirements for graph mining algorithms in order to handle Web data, such as partiality of input data and robustness for missing data.

This paper introduces some methods for the discovery of Web communities as an application of graph mining in order to clarify their characteristics. Requirements to graph mining algorithms for handling Web data are also discussed.

2 Methods for Discovering Web Communities

There are two main approaches for the discovery of Web communities; 1) search of fixed-size graph structure from Web snapshot data, and 2) decomposition of given Web graph into densely connected components. Both are explained below, followed by our own approach.

2.1 Search of fixed-size graph structure

For example, Web pages of aircraft enthusiasts often have hyperlinks to the companies of aircraft manufacturers. Hyperlinks of these pages (enthusiasts and companies) compose a bipartite graph and all of these pages are closely related. Kumar's trawling [5] is based on an assumption that Web pages constituting a bipartite graph are regarded as an indication of Web community sharing common interest. In his experiments, bipartite graph structures are enumerated by applying a priori algorithm. In addition to that, randomly selected samples are investigated by manual inspection. Its results show that most of the pages constituting a bipartite graph are actually closely related.

2.2 Decomposition of Web graph into densely connected components

In general, hyperlinks of related Web pages are densely connected with each other rather than others. Flake [3] applies maximum-flow minimum-cut theorem of network flow theory in order to discover densely connected components, which can be regarded as Web communities. His approach is often explained by the following metaphor: if edges are water pipes and vertices are pipe junctions, the maximum flow problem tells us how much water we can move from one junction to another, and the maximum flow is proved to be identical to minimum cut. Therefore, if you know the maximum flow between two points, you also know what edges you would have to remove to completely disconnect the same two points, which are called cut set. The approach accepts some Web pages as seeds of target Web community, and finds cut set that disconnect a component containing given seed pages.

2.3 Search of bipartite graphs based on data acquired from a search engine

A search engine can be regarded as a resource for Web data acquisition. The author proposed a method for discovering Web communities from the data acquired from a search engine [6][7]. Our method is similar to Kumar's one since both search bipartite graph structures. However, they are different in the following points:

1. Search of bipartite graphs from partial Web data without using Web snapshot data

Previous approaches of Web community discovery require relatively large-scale Web snapshot data. However, collecting Web data and maintaining them is not an

easy task. It is pointed out that the difference between Web snapshot data used for mining and actual Web data may cause the discovery of outdated Web communities [5]. Major search engines contain much updated Web data and they can be used for Web data acquisition in order to achieve relatively new Web communities. Some of the search engines allow users to access contained data, such as Google API.

2. Acquisition of backlinks from a search engine in order to follow hyperlink backward

Although most users use search engines in order to find Web pages about some keywords, a search engine enables us to follow hyperlinks backward. By attaching some option (such as "link:") to input URL, Web pages that contain hyperlinks to input URLs can be searched, which are called backlinks. Since hyperlinks to related Web pages often co-occur, backlink search enables us to find related Web pages.

Fig. 1. Outline of our method for discovering Web communities

Fig.1 shows the outline of our method for Web community discovery. Our goal is to discover a bipartite graph containing some given URLs. At first, some URLs regarding specific topic (such as baseball or Macintosh) are given as initial centers, and fans which co-refer all of the centers are searched by backlink search on a search engine (step 1). HTML files of the searched fans are acquired through the internet, and all the hyperlinks contained in the files are extracted. The hyperlinks are sorted in the order of frequency. Since hyperlinks to related Web pages often co-occur, the top-ranking hyperlink of the sorted result is expected to point to a page whose contents are closely related to the contents of centers. Therefore, the URL of the page is added as a new member of centers (step 2). By using newly generated centers, the above steps are repeated in order to find more centers (step 3).

Although this method is quite simple, it succeeds in discovering many related Web pages. Experimental results show that 19.8 related centers are actually discovered from given 5 seed URLs on average [7].

3 Web Community Discovery as an Application of Graph Mining

As an application of graph mining algorithms, the following requirements should be considered for the discovery of Web communities:

1. Partiality of input data: Nobody can collect data of the whole Web. Algorithms for the discovery of Web communities need to handle partial Web data. Suitable strategies for collecting Web data have to be considered.
2. Quantities of input data: On the contrary to the above, Web data are still huge even though they are partial. Capabilities for handling large-scale data are required for Web community discovery methods.
3. Qualities of input data: Depending on the network conditions, some of the Web pages may not be accessible. Robustness for missing or noisy data is necessary for Web community discovery.
4. Various structure of Web communities: Although Kumar regards a bipartite graph as a characteristic structure for Web communities, there might be other characteristic graph structures. Search algorithms for specific graph structure, such as clique or bipartite graph, are important. However, they are not enough for discovering real complicated Web communities.
5. Post processing of discovered Web communities: When fixed graph structure is searched from given Web data, many overlapping graphs will be found. Post processing of discovered Web communities such as clustering or labeling is necessary to assist users' understanding.
6. Interactive discovery of Web communities: Discovered Web communities are not always satisfactory to users since there are several criteria for "relatedness" among Web pages. It is preferable if users can control the strategies for searching Web communities by giving examples or negative examples.

References

1. Barabasi, A.-L., "LINKED – The New Science of Networks", Perseus Publishing, 2002.
2. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: "Graph Structure in the Web: Experiments and models", Proc. of the 9th WWW Conference, pp.309-320, 2000.
3. G. W. Flake, S. Lawrence, C. L. Giles, F. M. Coetzee: "Self-Organization and Identification of Web Communities", IEEE Computer, Vol.35, No.3, pp.66-71, 2002.
4. J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins: "The Web as a Graph: Measurements, Models, and Methods", Proc. of COCOON'99, LNCS 1627, pp.1-17, 1999.
5. R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins: "Trawling the Web for Emerging Cyber-Communities", Proc. of the 8th WWW Conference, 1999.
6. T. Murata: "Discovery of Web Communities Based on the Co-occurrence of References", Proc. of DS2000, LNAI 1967, pp.65-75, Springer, 2000.
7. T. Murata: "Finding Related Web Pages Based on Connectivity Information from a Search Engine", Poster Proc. of 10th WWW conference, pp.18-19, 2001.
8. L. Page, S. Brin, R. Motwani, T. Winograd.: "The PageRank Citation Ranking: Bringing Order to the Web", Online manuscript, http://www-db.stanford.edu/~backrub/pagerank sub.ps, 1998.