*Production Systems and Information Engineering, Miskolc, Vol. 1. (2003), pp. 91-106*

# ALGORITHMS FOR BUILDING CONCEPT SETS AND CONCEPT LATTICES

LÁSZLÓ KOVÁCS
Department of Information Technology, University of Miskolc
H-3515 MISKOLC, Hungary
kovacs@iit.uni-miskolc.hu

[Received May 2002 and accepted March 2003]

**Abstract.** In our days there is an increasing interest on the application of concept lattices for data mining, especially for generating association rules. The building of concept lattice consists of two, usually distinct phases. In the first phase the set of concepts is generated. The lattice is built in the second phase from the generated set. The paper gives an overview of the available methods and presents a proposed method for contexts of large size where the full context can not be stored in the main memory and some objects may be repeated in the context several times. The proposed algorithm for concept set generation is a fine-tuned version of the incremental concept set building method. At the end of the paper, the test results for comparing the new method with some known methods are given. The proposed method yields in a significantly better cost value than the other methods under the assumed conditions.

*Keywords:* formal concept analysis, concept lattice, algorithm, cost function

## 1. Introduction

Concept lattices are used in many application areas to represent conceptual hierarchies stored in a hidden form in the underlying data. The field of Formal Concept Analysis [1] introduced in the early 80ies has grown to a powerful theory for data analysis, information retrieval and knowledge discovery. In our days, there is an increasing interest on the application of concept lattices for data mining especially for generating association rules [8]. One of the main characteristics of this application area is the large amount of structured data to be analysed. Beside this area another important application field is the program analysis inside a compiler using concept lattices of very large size. A technical oriented application field of Formal Concept Analysis is the area of production planning where concept lattices are used to partition the products into disjoint groups yielding an optimal processing cost [6]. Since the cost of building a concept lattice is a super-linear function of the corresponding context size, the efficient computing of concept lattices is a very important issue investigated for several years [5].

The building of concept lattices consists of two usually distinct phases. In the first phase the set of concepts is generated. The lattice is built in the second phase from the generated set. We can find proposals in the literature for both variants, i.e. there are proposals addressing only one of the two phases and there are methods for combining these phases into a single algorithm. Based on the analysis of these methods, the cost for both steps is about the same order of magnitude and the asymptotic cost depends on mainly three parameters: the number of objects, the number of attributes and the number of concepts.

The cost is always larger than the product of these parameters. The concept-set generation algorithms have two main variants. The methods of the first group work in batch mode, assuming that every element of the context table is already present. The most widely known member of this group is the Ganter's next closure method. The other group of proposals uses an incremental building mode. In this case, the concept set is updated with new elements if the context is extended with a new object. The Godin's method belongs to this group. Regarding the phase for building the lattice, the proposed approaches are based on the considerations that the lattice should be built up in a top-down (or bottom-up) manner because in this case only the elements of the upper (or lower) neighbourhood are to be localised. The second usual optimisation step is to reduce the set of lattice elements tested during the localisation of the nearest upper or lower neighbour elements.

This paper addresses both of the problems, the generating of concept sets and the building of concept lattices. The proposal is intended to use for contexts of large size where the full context can not be stored in the main memory. According to our assumption, the access to context data is an expensive operation. Another basic feature of the investigated problem area is that the same incoming attribute set may occur several times in the different input objects, i.e. the objects may have the same set of attributes in the context.

## 2. Formal Concept Analysis

This section gives only a brief overview of the basic notations of the theory for *Formal Concept Analysis*. For a more detailed description, see [1].

A *K context* is a triple *K(G,M,I)* where *G* and *M* are sets and *I* is a relation between *G* and *M*. The *G* is called the set of *objects* and *M* is the set of *attributes*. The cross table *T* of a context *K(G,M,I)* is a matrix form description of the *relation I*:

$$t_{ij} = \begin{matrix} 1 \text{ , if } g_i I a_j \text{ and} \\ 0 \text{ otherwise,} \end{matrix} \qquad (1)$$

where $g_i \in G$, $a_j \in M$.

For every $A \subseteq G$, a *derivation operator* is defined:

$$A' = \{ a \in M \mid g\, I\, a \text{ for } \forall\, g \in A \} \qquad (2)$$

and for every $B \subseteq M$

$$B' = \{ g \in G \mid g\, I\, a \text{ for } \forall\, a \in B \}. \qquad (3)$$

The pair *C(A,B)* is a *concept* of the *K* context if

$$\begin{aligned} &- A \subseteq G \\ &- B \subseteq M \\ &- A' = B \\ &- B' = A \end{aligned} \qquad (4)$$

are satisfied. In this case, the $A$ is called the *extent* and $B$ is the *intent* of the $C$ concept. It can be shown that for any $A_i \subseteq G, i \in I$

$$(\cup_{i \in I} A_i)' = \cap_{i \in I} A'_I \tag{5}$$

and similarly for any $Bi \subseteq M, i \in I$

$$(\cup_{i \in I} B_i)' = \cap_{i \in I} B'_I \tag{6}$$

is satisfied.

Considering the $\Phi$ set of all concepts for the $K$ context, an *ordering relation* can be introduced for the concept set in the following way:

$$C_1 \leq C_2 \tag{7}$$

if

$$A_1 \subseteq A_2$$

where $C_1$ and $C_2$ are arbitrary concepts. It can be shown that for every $(C_1,C_2)$ pair of concepts, the following rules hold true:

$$C_1 \wedge C_2 \in \Phi \tag{8}$$

and

$$C_1 \vee C_2 \in \Phi.$$

Based on these features, $(\Phi, \leq)$ is a lattice, called *concept lattice*. According to the Basic Theorem of concept lattices, $(\Phi, \leq)$ is a complete lattice, i.e. the infinum and suprenum exist for every set of concepts. The following rules hold true for every family $(A_i, B_i), i \in I$ of concepts:

$$\begin{aligned} &\vee_{i \in I} (A_i, B_i) = (\cap_{i \in I} A_i, (\cup_{i \in I} B_i)''), \\ &\wedge_{i \in I} (A_i, B_i) = ((\cup_{i \in I} A_i)'', \cap_{i \in I} B_i) \end{aligned} \tag{9}$$

where $A''$ denotes the *closure* of the set $A$ and it is defined as derivation of the derivated set:

$$A'' = (A')'. \tag{10}$$

Using these definitions and rules, some other important and interesting rules may be derived. Some of the derived rules are given in the following list:

$$\begin{aligned} &A_1 \subseteq A_2 \Rightarrow A_2' \subseteq A_1', \\ &A \subseteq (A')', \\ &A' \subseteq ((A')')' \end{aligned} \tag{11}$$

The structure of the concept lattice can be used not only to describe the concepts hidden in the underlying data system, but it shows the generalisation relation among the objects and it can be used for clustering purposes, too. A good description on the related chapters of the lattice theory can be found among others in [2].

## 3. Algorithms for Generating the Concept Set

As for every concept the extent part is determined unambiguously by the intent part, the generation of the intent parts is investigated only. In most data mining applications the intent parts are enough to generate the rules. The rules define a relation, an implication among the attributes, i.e. on the intent parts. The actual support set for the rules is usually not important.

Among the sophisticated concept set generation algorithms the Ganter's next closure algorithm [1] is probably the most widely known method. It is widely accepted by experts, that this algorithm is not only the best known but the most effective one, too [4]. The concepts are generated according to an ordering relation. Based on the indexing of the elements, the lexicographical ordering between the concepts is defined in the following way:

$$A < B \iff \exists\, a_i \in G: A <i\, B \tag{12}$$

where

$$A <i\, B \iff a_i \in B \setminus A\ ,\ A \cap \{a_1,\dots,a_{i-1}\} = B \cap \{a_1,\dots,a_{i-1}\} \tag{13}$$

This method calculates the extent part first, and the intent part is generated from the extent part. The key function element, the next extent routine, tests several extent variants until it finds an appropriate one. The total asymptotic cost of the algorithm is equal to

$$O(CN^2\sigma + CN^2M) \tag{14}$$

where

$C$ : the number of concepts in the concept set, and
$\sigma$ is a cost unit.

Regarding the efficiency of this algorithm and the objectives, some facts should be taken into consideration:

1. the disk IO cost may be very high if $N$ is high;
2. the total cost is proportional to $N^2$, so it will be resulted in high costs for contexts with large number of objects, as it is assumed in our investigation.

One of the main characteristics of the Ganter's algorithm is that it accesses the context table several times during the generation of a concept. As the same context table element is accessed several times it is clear, this method assumes that

a: all parts of the context table are present at the concept set generation;
b: the context table can fit into the memory with low cost access operations.

Based on these assumptions, this method is called a batch method. A different kind of approach is presented by Godin [2]. His proposal is an incremental concept formation method, where the concept set is updated in an incremental manner, i.e. the set is updated when the context table is extended by a new object instance. In this kind of method, every context table element is accessed only once, yielding a minimal IO cost. The building of the concept set in incremental mode is based on the following rule:

> *Every new concept intent after inserting a new object into the context, will be the result of intersecting the attribute set of the new object with some intent set already present in the concept set.*

Godin's method can be used for updating the concept set after insertion of a new object into the context. The algorithm consists of the following main steps. First, the concepts are partitioned into buckets based on the cardinality. Next, the buckets are processed in ascending cardinality order. Every intent in the current bucket is intersected with the intent set of the new object. If the result set is not present in the concept set, it will be added. The cost estimation for the algorithm can be given by

$$O(N\sigma + CNDM). \tag{15}$$

This formula assumes linear existence testing. Linear testing was implemented in the algorithm as testing can be reduced to the subset of the so called marked elements. The marking test can be performed only in linear mode. In the cost estimation formula $D$ denotes the number of elements with a mark. This mark is assigned to the elements generated in the current phase. Comparing this cost function with the cost estimation of the next closure method, we can see that the incremental method will be more efficient if

1: the $\sigma$ cost unit is high;

2: or $N$ is high.

On the other hand, the cost of Godin's method is more sensitive to the $C$ size of the concept set.

Beside these two basic concept set generation algorithms, there are some other proposals in the literature, mainly some kind of optimisation of the basic algorithms. From these papers, only some of the most recent ones will be presented here to demonstrate the computational efficiency of the most up-to-date variants.

In the paper of Hu [3], the concept set generation process is coupled with the calculation of the support value in order to discover association rules from the concept lattice. The concept set building part is based on the incremental method of Godin, thus resulting the same asymptotic calculation cost estimation value:

$$O(N\sigma + CNDM). \tag{16}$$

Another proposal is the Titanic algorithm, presented in [7]. This method uses the support values of the different attribute sets to determine the concept intents. It generates the candidate generator sets in increasing order of the size. A set is called a generator set if its closure is a concept intent and it is minimal, i.e. it does not contain any other generators for the same concept intent. The method processes first the one-attribute-long candidates and after then generates the candidate sets for the next level. At the next level, the length of the intents is increased by one

$$O(NM\sigma + aMCN + a^2C^2M). \quad (17)$$

The algorithm processes not only the concepts, but all of the candidates, thus in the cost estimation formula, $a$ denotes how many times the number of candidates is larger than the number of concepts. This value is always greater than 1. The most costly part of the algorithm is the generation of candidate sets. In this phase, every pair at level $l$ having the same values in the first ($l$-1) attributes will be processed to generate a new candidate set at level $l$.

The proposal of Lindig given in [4], is aimed at not only the generation of the concept set but on the building of the whole concept lattice. If we consider now only the concept set generation part of the algorithm, this method is related to the Ganter's method in many aspects. It assumes a lectical ordering among the concepts and the concepts are processed according to this ordering. The method also generates for every new concept the set of upper neighbour concepts to use this kind of information during the insertion into the concept lattice.

The neighbours of a concept are generated using the closure operation for the candidate neighbour attribute sets. At every call of the neighbour routine the full context table is scanned. The cost estimation of this algorithm is

$$O(Nc\sigma + CN^2M). \quad (18)$$

Thus the asymptotic complexity is the same as for the Ganter's method.

The aim of the investigation was to find an efficient algorithm that can be used for cases with large context size, so the proposals found in the literature were evaluated using the following criteria:

1. the disk IO should be minimal, every context table should be accessed only once and
2. the in-memory operations should be optimised to omit the redundant calculations.

Based upon these selection criteria, the incremental method is the best solution as it has only a linear disk IO cost and not all elements of the context table should be available at the beginning of the concept set building. To achieve a better performance, the objective was to improve the in-memory operations of the existing incremental methods. In the next sections of the paper a fine-tuned version of the incremental concept set building method is presented and the efficiency of the proposed method is also demonstrated with comparison tests. Based upon the test results, we can say that the incremental methods can outperform the batch method in practical applications. This result is in consonance with the results of Godin.

## 4. Fine tuned incremental method

According to the properties of the incremental methods, the context table is generated by adding single objects one by one, after each other. Let's denote the intent part of the concept set built up from the first k objects by

$$L_k$$

The $L_{k+1}$ is constructed from $L_k$ and $a_k$ where $a_k$ denotes the $k$-th object in the input list. The generation of $L_k$ is based on the following considerations that can be proven very easily from the basic properties of the concept lattices, so we omit here the proofs.

Proposition 1.

$$\text{for every } a_k \in G : A(a_k) \in L_k \tag{19}$$

where $A(a)$ denotes the attribute set of object $a$.

Proposition 2.

$$\text{for every } A,B \in L_k \Rightarrow A \cap B \in L_k. \tag{20}$$

Proposition 3.

$$\text{for every } A \in L_{k+1}, A \neq A(a_k), A \notin L_k : \exists\, B \in L_k : A = A(a_{k+1}) \cap B. \tag{21}$$

Based on these propositions, in every iteration loop starting with $L_k$, the $A(a_k)$ can be added first to $L$ and then the intersections of $A(a_k)$ with the elements already present in the $L_k$ are generated and inserted into $L_{k+1}$. Since the number of possible pairs for an intersection is very large, the algorithm has a high cost in testing the intersections. A possibility of cost reduction is provided by the fact that not every pair generates a new concept intent. Most of the pairs yield in an existing intent value. The key point for fine tuning of the incremental algorithm in our proposal is based on the following simple considerations:

Proposition 4.

$$\text{for every } A,B,C \in L_k, A \cap B = 0, C \subseteq A:\ C \cap B = 0. \tag{22}$$

The meaning of this rule is for us the following: if A is disjoint with some B then all of its subsets can be pruned from testing.

Proposition 5.

$$\text{for every } A,B,C \in L_k, A \subseteq B\ , C \subseteq A: C \subseteq B \text{ and } C \cap B = C. \tag{23}$$

Thus, if an intent part is a subset of the tested element, then all its subsets can be eliminated.

Proposition 6.

$$\text{if } \exists\, B \in L_k, A(a_k)) = B \ \text{ then for } \forall\, C \in L_k: C \cap A(a_k) \in L_k. \tag{24}$$

Thus, if an intersect part is presented in the concept set, then the whole testing loop for $A(a_k)$ can be eliminated.

To implement the cost reduction elements into the concept set building algorithm, the following modifications of the basic incremental method were developed:

1. Before the testing loop for the new incoming object, it is tested whether it equals an already existing intent part. The test for existence checking is performed using a B-tree structure, resulting in a test cost of *O(log(C))*.
2. Before the intersects of the elements would be inserted into the concept set, the candidate elements (the results of the intersections) are stored in a hash table, so the sets generated repeatedly can be detected in a cost effective way.
3. To reduce the redundant intersection tests, the elements of the concept set are stored in a special pointer list where the elements containing a given attribute value are connected to each other. The intersection operation should be performed only for elements having at least one common attribute. Thus the intersection test for disjoint elements can be eliminated.
4. To eliminate the insertion testing for intents already present in the concept set, during the intersection phase, a special marker is used in the hash table. In most cases, the existence can be detected in the hash table building phase, before the insertion phase.

Based on these considerations, the structure of the algorithm is

```
L1  loop
      read_next(a)
      find_in_Btree(a)
C1    if not found() then
             update_pointer_chain(a)
             insert_set(a)
L2           foreach X (element of) L(k), X (intersect) A(a) not = 0 do
                Y = X (intersect) A(a)
                insert_hash(Y)
             endfor
L3           foreach X elements in hash do
                if it is not marked then
               insert_set(X)
                endif
             endfor
       endif
    until (no more input)
```

The estimated cost of the proposed algorithm is calculated in the following way.

| | |
|---|---|
| read_next() | *O(σ)* |
| find_in_Btree | *O(Mlog(C))* |
| update_pointer_chain | *O(M)* |
| insert_hash | *O(M)* |
| intersect | *O(M)* |
| L1 loop | it is executed for every object, so the number of iterations is equal to *N* |

| | |
|---|---|
| L2 loop | it is executed for every intent set having common attribute with the new object, the number of iterations is equal to $C'$ where $C' < C$ |
| L3 loop | the insert operation is performed only if the intersection result is not marked, the number of iterations: $C'' << C$ |
| C1 branching | the inner part is executed for objects with a new attribute set, $N' < N$ |

The total cost can be given by

$$O(N\{\sigma + MlogC)\} + N'\{M + MlogC + C'\{M\} + C''\{MlogC\}\}). \tag{25}$$

This expression can be transformed into a simpler form

$$O(N\sigma + NMlogC + N'MlogC + N'C'M + N'C''MlogC) \tag{26}$$

and pruning the non-dominant tags:

$$O(N\sigma + NMlogC + N'MlogC + N'C''MlogC). \tag{27}$$

One of the benefits of this algorithm is that it can reduce the computational cost if the new object has an attribute set contained in the context already. In applications this case may occur often, for example in processing questionnaires where several people may give the same answer. The $C'$ value is the number of sets having intersection with the new object's attribute set. A rough estimation for $C'$ can be given as follows:

Let's denote the length of the attribute set of the objects by $K$. For $K = 1$, the probability that it has no common part with a subset of $M$ is equal to

$$2^{M-1} / 2^{M} = ½ \tag{28}$$

so

$$P_1 = ½. \tag{29}$$

In a similar way we get, that

$$P_i = 1/2^i \tag{30}$$

The number of subsets having length $i$ is equal to

$$\binom{M}{i}. \tag{31}$$

So the probability that an arbitrary subset has no common part with an other subset is

$$P = \Sigma_i \binom{M}{i} P_i / 2^M = (\Sigma_i \binom{M}{i} 1/2^i) / 2^M = (3/4)^M \tag{32}$$

Although, the relative gain of using this intersection pointer list is lower for large M values, the absolute number of testing that can be omitted is large enough to use this kind of optimisation in the applications.

Regarding the $C''$ value, the gain here can be more dominant as the number of pruned sets is much higher. If $c_k$ is the number of concepts after inserting the $k$-th object, then the following holds true:

$$1 = c_1 < c_2 < \ldots < c_N = |C|. \tag{33}$$

The total number of insertion testing without marking is

$$c_1 + c_2 + \quad + c_N. \tag{34}$$

So, the gain of the reduction is equal to the difference

$$c_1 + c_2 + \quad + c_N - |C|. \tag{35}$$

A rough estimation for $c_k$ can be

$$O(k^2) \tag{36}$$

so this reduction step is very important.

## 5. Algorithms for Building Concept Lattices

Let's denote the set of concepts and the ordering relation on this set by $(\Phi, \leq)$. For any arbitrary concept $C$, the upper and lower neighbour can be defined in the following way. An $L \in \Phi$ is a lower neighbour of $C$ if

$$L \leq C \text{ and } !\exists\, X \in \Phi : L \leq X \leq C. \tag{37}$$

The upper neighbour can be defined in a similar way. In a lattice an element may have several upper and lower neighbour elements. We denote the set of lower (upper) neighbours for $C$ by $Low(C)$ and $Upp(C)$. For building the lattice $Low(C)$ and $Upp(C)$ must be known for every $C$.

The naive way to generate $Upp(C)$, $Low(C)$ is to test all of the concept pairs. The main structure of the algorithm for $Upp(C)$ consists of two nested loops to test every concept pair. If one of them is the ancestor of the other then it should be tested whether there is another element being between these two elements. So, this algorithm contains three nested loops and the cost of the execution can be estimated by the following formula:

$$O(C^2 C_u M) \tag{38}$$

where $C_u$ is the number of upper neighbours. Comparing this estimation with the cost values for generating the concept set, we can see that this cost is of the same magnitude or sometimes higher than the cost of the first phase. This short evaluation shows the importance of an optimised lattice building method.

In the literature, we can find several approaches addressing this problem. Only the most recent ones are described here.

The proposal of Ky Hu [3] was published in 1999. The first phase of this method is based on Godin's incremental lattice generation method. The algorithm generates the concepts in increasing cardinality order of the intent part. According to this principle, the parents are generated first, and only after that come the children. This means, that during insertion of a new concept into the lattice only the parent neighbours, the ancestor part of the lattice should be tested. The lower neighbours will be determined during the insertion of the children elements. The concepts of smaller intent size are all tested to find the potential parents. For every potential parent, the set of its lower neighbours is tested whether they are parents of the new concept. If the candidate element is an upper neighbour, then all nodes marked as upper neighbour previously and being an upper neighbour of the tested element should be removed from the set of marked nodes. At the end, the marked elements will constitute the upper neighbour set.

The main optimisation elements presented in [3] are

- only a subset of concepts is tested during the search for potential parents,
- the test for pruning elements marked previously is reduced to a special subset of elements.

The cost estimation of the algorithm can be given by

$$O(CC_nC_aM) \tag{39}$$

where

$C_a$ : number of ancestor nodes,
$C_n$ : number of neighbour nodes.

Another current approach is the algorithm of Lindig presented in [4]. The proposed concept set and lattice building algorithm is related to the Ganter's method in many aspects. The concepts from the concept set are processed in total order to make sure all concepts that are inserted into the lattice are also considered for their neighbours. The lectical order used in Ganter's method is an appropriate ordering. Due to this processing order only the upper neighbour set is needed to be generated here, too. The test for upper neighbours is based on extending the extent part with a new element and performing a closure operation. The cost estimation for this algorithm can be given by

$$O(CN^2M), \tag{40}$$

i.e. it has the same asymptotic cost value as the next closure method has.

The method presented in [4] performs an element-wise update of the lattice according to a linear extension of the lattice order. The lattice extension is done in a top-down manner, starting from the top node and processing the rest of the nodes according to a total order which is a linear extension of the lattice order. At each step the current element is connected to each of its immediate successors in the final lattice. During the building of the lattice a special subset of elements, the so called border elements play an important role. The border of a lattice $\Phi$ is defined as

$$Border(\Phi) = \{ C_i \in \Phi \mid \forall C_i' \in (\Phi \setminus 0), C_i \leq C_i \Rightarrow C_i' = C_i \} \quad (41)$$

where $C_i$ and $C_i$' denote concepts in the lattice. During the insertion of a new $X$ concept, the border will change. The border set always contains the new element, whereas all elements of the old border that are greater than $X$ are dropped out. The cost estimation of this method is

$$O(CC'^2M) \quad (42)$$

where $C'$ denotes the number of elements in the border region.

Based upon the proposals mentioned here, we can see that every optimisation approach is based on the following considerations:

- the lattice should be built up in a top-down (or bottom-up) manner so only the elements of the upper or lower neighbourhood are to be localised;
- the search for elements of *Upp*($C$) or *Low*($C$) are performed on only a subset of the whole lattice.

This kind of optimisation method requires more or less meta-data structures with significant administration cost. In the next section another approach for optimisation of concept lattice building is introduced, that is based on a simple insertion.

## 6. Efficiency Analysis of the simple lattice building algorithm

Let us consider now a simple lattice building algorithm which locates the elements of *Upp*($X$) and *Low*($X$) for an arbitrary $X$ by using a simple top-down or bottom-up lattice scanning method. The search starts at the top (bottom) node traversing the concepts being an ancestor or descendant of $X$. The ancestor nodes having no child with this property are the elements of the neighbourhood. The search for neighbours can be defined in a recursive way:

```
search_upp(Y,X)
 c=0
 foreach C child of Y do
    if C > X then
       search_upp(C,X)
       c++
    endif
 endfor
 if c = 0 then
   Upp(X) = Upp(X) + Y
 endif
```

where $X$ is the new concept's upper neighbour which we are searching for. $Y$ is the tested lattice element.

During the tests with different input orders we became aware of another and more important factor for efficiency, namely the parent-child relationships among the elements. The rule is the following: in the search processes for upper or lower neighbour elements, the cost of lattice traversing depends on the number of nodes to be processed and not on the number of ancestors or descendants. The number of tested nodes is greater than the number of ancestors or descendants as there are a large number of nodes with negative test results. These nodes are located on the border of the ancestors' or descendants' sub-lattice. These elements are children of the ancestors (or parents of the descendants) but they themselves are not ancestors (descendants) of the new element. According to our test results, the cost for processing these border elements can be very high and it depends dominantly on the position and insertion ordering of the $X$ nodes. The aim of our investigation was to find an optimal order of concept insertion yielding a low computational cost.

During the search for upper neighbour nodes, at every ancestor node, all of the lower neighbour elements are tested. A similar statement is true for the search for lower neighbouring nodes. The cost of insertion for an arbitrary $C$ concept is proportional to the number of nodes to be tested in both directions:

$$Cost_C = \Sigma_{n \in SAC}\, NC_n + \Sigma_{n \in SDC}\, NP_n \qquad (43)$$

where

$NC_C$ : the number of lower neighbour nodes at $C$,
$NP_C$ : the number of upper neighbour nodes at $C$,
$SA_C$ : the set of ancestors of $C$,
$SD_C$ : the set of descendants of $C$.

The total cost for building the concept lattice is

$$Cost = \Sigma_{C \in \Phi}\, Cost_C\,. \qquad (44)$$

After inserting a new element into a lattice, the *NC, NP, SA, SD* values may change, so the *NC, NP, SA, SD* parameters are a function of the discrete time value. Let's denote this time value by $i$ which is a simple sequence number, thus we get

$$Cost = \Sigma^{\Phi}_{i=1}(\Sigma_{n \in SAii}\, NC_{n,i} + \Sigma_{n \in Sdii}\, NP_{n,i}) \qquad (45)$$

where *SAni, SDni* denote the set of ancestors (descendants) of the concept $n$ at the time point $i$. Similarly, the $i$ index for *NC* and *NP* denotes the value at the time point $i$. At the end of the building process, the $\Phi$ lattice is built up completely. This resulting lattice does not depend on the insertion order of the concepts. So for every $C \in \Phi$ , *NC, NP, SA, SD* have a given, fixed value. But for any $0 < i < |\Phi|$ point of time, these values may be unknown. Any of the functions may increase or decrease during the building phase.

Summarising these considerations, the rule of optimisation can be formulated as follows:

*The larger the number of elements below (above) an x element is, the longer the number of lower neighbour nodes (upper neighbour nodes) for x should remain on such a low value as is possible.*

This rule implies the following rule that can be implemented easier in the practice:

*The elements with low ancestor (descendant) population should get a new upper (lower) neighbour first.*

To provide a feasible method for this problem, a cost saving heuristic optimisation method based on the previous considerations is introduced.

The proposed heuristic method builds an approximate tree structure for the lattice. This can be considered as a spanning tree. This tree can be generated in an efficient way and the elements are inserted into the lattice in the order based on the hierarchy structure of this tree. The information to build this tree can be gathered during the concept set generation phase. The tree is processed in a top-down traversing and every processed element will be inserted into the lattice according to the order of traversing. The spanning tree is generated in the concept set building stage, during the intersection generation phase. The basic consideration behind the tree construction algorithm is the following. If

$$A = B \cap C$$

then B and C contain A, so B and C are candidate parents of A. The candidate parent concept with minimal length (the number of attributes not present in A is minimal) is selected as the parent element of A in the spanning tree.

## 7. Test Results

To compare the efficiencies of the different approaches, the different lattice building methods were implemented in a test system. The implementation programming language was the Java to create a platform independent solution. In the first phase the different lattice set building algorithms were tested. According to the test results the proposed fine-tuned method provides the best cost values among the tested ones. The next table summarises some results related to two different concept sets.

**Table 1: Experimental test results for concept set generation**

| Method | Elapsed time | Size of the concept set |
|---|---|---|
| naive method | 245 | 3865 |
| Ganter method | 14 | |
| Godin method | 6 | |
| proposed method | 3 | |
| naive method | 795 | 37344 |
| Ganter method | 170 | |
| Godin method | 83 | |
| proposed method | 42 | |

In the second phase the lattice building algorithms were tested. Taking the simple lattice building algorithm presented in the previous section, the proposed approximation tree ordering resulted in a better result than the other methods. It is an interesting experience, that in the case of insertion ordering based on intent size the results are always worse than with random ordering. The next closure method that is also based on special ordering, is also worse than an average random ordering method. The following table shows the computational cost in elapsed time and in number of performed set operations.

**Table 2: Experimental test results for lattice building**

| Method | Elapsed time | Number of operations |
|---|---|---|
| spanning tree order | 10 | 7068255 |
| lattice top-down traversing order | 9 | 6093848 |
| Keyun method's order | 12 | 7136129 |
| normal intersection order | 13 | 8181065 |
| random order | 23 | 17020450 |
| next closure order | 40 | 38784364 |
| increasing set size order | 71 | 49308966 |
| Valtchev method's order | 98 | 56898773 |

In the tests, the size of the concept lattice is between 100 and 12000. All of the contexts were generated randomly.

We should mention that these results are based on the simple insertion algorithm. The methods mentioned in Table 2 are usually based on modified lattice building algorithms which include some kind of heuristic elements, too. In the next closure method, for example, the searching phase for the descendants is omitted as the current incoming element is always the smallest one without any descendants. Eliminating this step, the total cost can be significantly reduced. Thus, the result values in the table are related only to the insertion order, the original lattice building algorithms can provide better cost values. The aim of this investigation was only to analyse the effect of different insertion orders during the lattice building algorithm.

**Acknowledgements**
This work has been supported by the Hungarian Eötvös State Fellowship Grant No.: MÖB 595-1-2001.

## REFERENCES

1. GANTER, B., WILLE, R.: *Formal Concept Analysis*: *Mathematical Foundations*, Springer Verlag, 1999.
2. GODIN, R., MISSAOUI, R., ALAOUI, H.: *Incremental concept formation algorithms based on Galois lattices*, Computational Intelligence, 11(2), 1995, pp. 246-267.
3. HU, K., LU, Y., SHI, C: *Incremental Discovering Association Rules*: *A Concept Lattice Approach*, Proceedings of PAKDD99, Beijing, 1999, pp. 109-113.
4. LINDIG, C.: *Fast Concept Analysis*, Proceedings of the 8th ICCS, Darmstadt, 2000.
5. NOURINE, L., RAYNAUD, O.: *A Fast Algorithm for Building Lattices*, Information Processing Letters, 71, 1999, pp. 197-210.
6. RADELECZKI, S., TÓTH, T.: *Fogalomhálók alkalmazása a csoporttechnológiában*, OTKA kutatási jelentés, Miskolc, Hungary, 2001.

7. STUMME, G. , TAOUIL, R., BASTIDE, Y., PASQUIER, N., LAKHAL, L.: *Fast Computation of Concept Lattices Using Data Mining Techniques,* 7th International Workshop on Knowlegde Representation meets Databases (KRDB 2000), Berlin, 2000.
8. ZAKI, M., OGIHARA, M. *Theoretical Foundations of Association Rules*, Proceedings of 3 rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98), Seattle, Washington, USA, June 1998.
9. KOVACS, L.: *Efficiency Analysis of Building Concept Lattice*, Proceedings of 2nd ISHR on Computational Intelligence, Budapest, 2001.