Expert Systems with Applications 42 (2015) 4474–4481

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

A fast incremental algorithm for constructing concept lattices

Ligeng Zou, Zuping Zhang *, Jun Long

School of Information Science and Engineering, Central South University, Changsha 410083, PR China

ARTICLE INFO

Article history:
Available online 31 January 2015

Keywords:
Formal Concept Analysis
Concept lattice
Incremental algorithm
Lattice construction

ABSTRACT

Incremental algorithms for constructing concept lattices can update a concept lattice according to new objects added to the formal context. In this paper, we propose an efficient incremental algorithm for concept lattice construction. The algorithm, called FastAddIntent, results as a modification of AddIntent in which we improve two fundamental procedures including searching for canonical generators and fixing the covering relation. We describe the algorithm completely, prove correctness of our improvements, discuss time complexity issues, and present an experimental evaluation of its performance and comparison with AddIntent. Theoretical and empirical analyses show the advantages of our algorithm when applied to large or (and) dense formal contexts.

1. Introduction

Formal Concept Analysis (FCA) was introduced by Rudolf Wille in the early 1980s (Ganter & Wille, 1999; Wille, 1982, 2009). It is a method of analysis of object-attribute relational data and knowledge representation. For the last two decades, FCA has been used extensively in various disciplines such as software engineering (Wermelinger, Yu, & Strohmaier, 2009), linguistics (Priss, 2005), information retrieval (Dau, Ducrou, & Eklund, 2008), ontology engineering (Maio et al., 2012), bioinformatics (Amin, Kassim, & Hefny, 2013) and data mining (Poelmans, Elzinga, Viaene, & Dedene, 2010). An extensive overview of FCA-based methods in different application domains is given by Poelmans, Ignatov, Kuznetsov, and Dedene (2013a).

As the underlying core structure of FCA, concept lattices have solid mathematical foundations and also the ability to visualize the incidence relationship between objects and attributes. However, applying FCA methods to large formal contexts could bring many challenges, because concept lattices can grow exponentially large in the worst case and counting the number of all concepts is an NP-complete problem (Babin & Kuznetsov, 2010; Kuznetsov, 2001; Kuznetsov, 2004). Since an open problem of "handling large contexts" was pointed out at the fourth international Conference on Formal Concept Analysis, designing more efficient algorithms for handling large and complex incidence matrices has become a popular research topic (Poelmans et al., 2013b), and among all types of those lattice construction algorithms, incremental algorithms have a unique advantage. The input formal context may not be fixed in a real-life application of FCA, which means we have to update the present lattice or compute a new lattice from scratch. Obviously, computing the corresponding changes only and updating the current lattice should be a better choice in most scenarios, which can be handled by incremental algorithms such as Godin's (Godin, Missaoui, & Alaoui, 1995), Object Intersections (Carpineto & Romano, 2004) and AddIntent (Van Der Merwe, Obiedkov, & Kourie, 2004). It makes using incremental algorithms a very suitable and reasonable option for maintaining an online lattice in applications of FCA.

In this paper, we introduce a new incremental algorithm for constructing concept lattices. The algorithm we propose is a refinement of AddIntent (Kourie, Obiedkov, Watson, & van der Merwe, 2009; Van Der Merwe et al., 2004) in which we improve two fundamental procedures including fixing the lattice order relation and searching for canonical generators. The improvements make the algorithm perform considerably better than AddIntent when applied to relatively large or (and) dense datasets.

The remainder of the paper is composed as follows. In Section 2, we recall some basic definitions and propositions of FCA. Section 3 gives a brief survey of incremental algorithms, then describes our algorithm and shows the correctness of the proposed improvements. In Section 4, we discuss complexity issues. Section 5 presents an experimental evaluation of the performance of the presented algorithm. Our work is concluded in Section 6.

2. Preliminaries

In this section, we introduce basic FCA notions and conventions. All definitions and propositions are assumed they are written by

* Corresponding author. Tel.: +86 073182539925.
E-mail addresses: ligeng-zou@csu.edu.cn (L. Zou), zpzhang@csu.edu.cn (Z. Zhang), dragone7968@163.com (J. Long).

http://dx.doi.org/10.1016/j.eswa.2015.01.044
0957-4174/

Ganter and Wille (1999) which the reader is kindly referred to for a more detailed description.

Definition 1. A *formal context* is a triple of sets $K = (G, M, I)$, where $I \subseteq G \times M$ is a binary relation between G and M. The elements in G and M are called *objects* and *attributes*, respectively. $g I m$ or $(g, m) \in I$ indicates the object g has the attribute m.

A formal context can be represented by a cross table (or matrix) where every row is an object and every column is an attribute. Crosses in the table represent the incidence relation I. An example of a simple formal context is illustrated in Table 1.

Definition 2. For a set of objects $A \subseteq G$ we define the set of common attributes shared by all objects in A as:

$$A^{\uparrow I} = \{m \in M | \forall g \in A, g I m\}.$$

Similarly, for a set of attributes $B \subseteq M$ we define the set of objects that have all attributes in B as:

$$B^{\downarrow I} = \{g \in G | \forall m \in B, g I m\}.$$

Definition 3. A *formal concept* of a formal context $K = (G, M, I)$ is defined as a pair (A, B) where $A \subseteq G$, $B \subseteq M$, $A^{\uparrow I} = B$ and $B^{\downarrow I} = A$. A and B are called the *extent* and the *intent* of the concept (A, B), respectively.

Definition 4. Let (A_1, B_1) and (A_2, B_2) be two formal concepts of a given formal context K. (A_1, B_1) is called a *superconcept* of (A_2, B_2) and (A_2, B_2) is called a *subconcept* of (A_1, B_1) if $A_2 \subseteq A_1$ (or equivalently, $B_1 \subseteq B_2$) which can be denoted by $(A_2, B_2) \leqslant (A_1, B_1)$. The set of all formal concepts of K together with the superconcept-subconcept relation makes a complete lattice that is called the *concept lattice* of the context.

Since the subconcept-superconcept relation is a natural partial order relation, we can simply adopt the definition of neighboring nodes of order theory here.

Definition 5. Let c_1 and c_2 be two concepts of a given formal context K. We say c_1 is a *lower neighbor* (or a *child*) of c_2 and c_2 is an *upper neighbor* (or a *parent*) of c_1, if $c_1 \leqslant c_2$ and there is no other concept c_3 with $c_3 \neq c_1$, $c_3 \neq c_2$ and $c_1 \leqslant c_3 \leqslant c_2$. This relationship (also called the *covering relation*) is denoted by $c_1 \prec c_2$.

Like any other partially ordered sets, concept lattices can be represented by line diagrams (or Hasse diagrams). In a line diagram, only neighboring nodes are connected by edges and c_2 should be above c_1 if $c_1 \prec c_2$. For instance, Fig. 1 is the Hasse diagram of the concept lattice derived from Table 1.

In order to make our proposed improvements clear, we need to provide two elementary propositions here.

Proposition 1. *$K = (G, M, I)$ is a formal context where $A, A_1, A_2 \subseteq G$ are sets of objects and $B, B_1, B_2 \subseteq M$ are sets of attributes. Then,*

(1) $A_1 \subseteq A_2 \Rightarrow A_2^{\downarrow I} \subseteq A_1^{\downarrow I}$,
(2) $B_1 \subseteq B_2 \Rightarrow B_2^{\downarrow I} \subseteq B_1^{\downarrow I}$,

Table 1
Example of a formal context.

	a	b	c	d	e
1	×	×	×	×	×
2	×	×	×		×
3			×	×	
4					×
5	×	×	×	×	

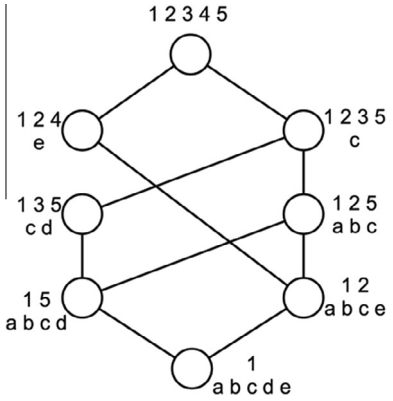

Fig. 1. Concept lattice of the formal context in Table 1.

(3) $A \subseteq A^{\uparrow I \downarrow I}$,
(4) $B \subseteq B^{\downarrow I \uparrow I}$,
(5) $A^{\uparrow I} = A^{\uparrow I \downarrow I \uparrow I}$,
(6) $B^{\downarrow I} = B^{\downarrow I \uparrow I \downarrow I}$,
(7) $A \subseteq B^{\downarrow I} \Leftrightarrow B \subseteq A^{\uparrow I} \Leftrightarrow A \times B \subseteq I$.

Corollary 1. *$A^{\uparrow I \downarrow I}$ is the smallest extent that includes A, and $B^{\downarrow I \uparrow I}$ is the smallest intent that includes B.*

Corollary 2. *$A \subseteq G$ is the extent of a formal concept if and only if $A = A^{\uparrow I \downarrow I}$. Similarly, $B \subseteq M$ is the intent of a formal concept if and only if $B = B^{\downarrow I \uparrow I}$.*

Proposition 2. *T is an index set. For every $t \in T$, let A_t be a set of objects. Then*

$$\left(\bigcup_{t \in T} A_t\right)^{\uparrow I} = \bigcap_{t \in T} A_t^{\uparrow I}.$$

The same statement holds for the sets of attributes too.

3. The FastAddIntent algorithm

3.1. Incremental algorithms

Computing a concept lattice has been widely studied, which leads to the development of many efficient algorithms. In general, we can divide these algorithms into two categories including batch algorithms (Ganter, 2010; Kuznetsov, 1993; Outrata & Vychodil, 2012) and incremental algorithms (Godin et al., 1995; Kourie et al., 2009; Valtchev & Missaoui, 2001; Van Der Merwe et al., 2004). Batch algorithms usually construct the lattice in a bottom-up (or top-down) approach, while incremental algorithms compute the lattice by adding objects (or attributes) of a given context one by one. A significant characteristic of the latter is that any information regarding objects (or attributes) that have not been processed remains unknown to algorithms.

Comparing to batch algorithms, incremental algorithms are more appropriate in real-life applications that work with dynamic datasets and therefore they are of our main interest in this paper. Two of the most representative incremental algorithms are Godin (Godin et al., 1995) and AddIntent (Van Der Merwe et al., 2004). Valtchev, Missaoui, and Godin (2008) adopt a fast variant of the Godin algorithm and integrate it into a very efficient framework for incremental frequent closed itemsets (FCIs) mining. An

improved version of AddIntent can also be found in Lingling, Lei, Anfu, and Funa (2011), in which two fields are added to every concept for locating generated new concepts quickly. The experiments show the improved version has small advantages when the number of objects is not much bigger than the number of attributes (Lingling et al., 2011). The main idea of those algorithms is to modify corresponding concepts and create new concepts if necessary. Based on previous researches, we can describe concepts during the update in terms of the following definition (Van Der Merwe et al., 2004).

Definition 6. Let L and L' be the concept lattice of a given context before and after adding a new object g, respectively. The attribute set of g is denoted by $g^{\uparrow I'}$ instead of $\{g\}^{\uparrow I'}$. Let (A, B) be a formal concept in L'. Then,

(1) (A, B) is a *new* concept if B is not an intent of any concept in L,
(2) (A, B) is a *modified* concept if $B \subseteq g^{\uparrow I'}$,
(3) (A, B) is an *old* concept if it remains unchanged from L to L',
(4) (A, B) is a *generator* of a new concept (X, Y) if $B \cap g^{\uparrow I'} = Y \neq B$; otherwise, it is a *general old* concept.

According to the above definition, we can see that searching for modified concepts and generating new concepts are crucial to incremental algorithms. Modified concepts are easy to identify since their intents are subsets of $g^{\uparrow I'}$, and we only need to add the new object g to their extents. On the other hand, new concepts would require much more effort during processing because they do not exist before inserting the new object. In order to determine which concepts need to be created, incremental algorithms usually utilize the relation between generators and new concepts defined in Definition 6. Every new concept (X, Y) has at least one generator, but there is one unique concept among all generators called the *canonical generator* of (X, Y) (Van Der Merwe et al., 2004). The canonical generator is a superconcept of all generators of (X, Y), which means it is the upper bound. Since each new concept only has one canonical generator, we can create new concepts by identifying corresponding canonical generators. The correspondences between concepts in L and concepts in L' are illustrated in Fig. 2. Circles in Fig. 2 represent different groups of concepts. If the reader wants to know more about these mappings, a comprehensive demonstration is provided by Kriegel (2014).

As one of the fastest incremental algorithms, our selected algorithm AddIntent traverses a concept lattice in a recursive way and takes advantage of Proposition 1 and Proposition 2 in Van Der Merwe et al. (2004) to avoid consideration of general old concepts and non-canonical generators. The main strategy of AddIntent is quite straightforward. First, it finds or creates a concept C with intent equal to $g^{\uparrow I'}$. If C is not new, the algorithm terminates; otherwise, it computes the set of potential upper neighbors of C and calls AddIntent recursively for every concept in that set if necessary. After determining all true upper neighbors of C, fixing of the covering relation in the new Hasse diagram is conducted. A full description of AddIntent is not feasible here, and the reader is kindly referred to (Van Der Merwe et al., 2004) for more detailed information.

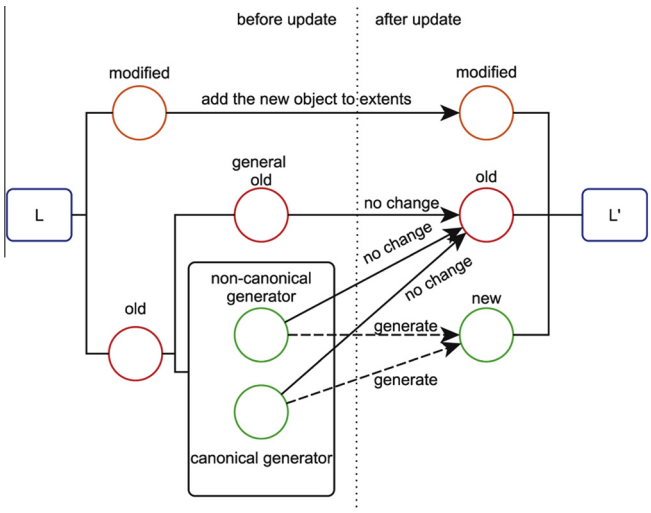

Fig. 2. Correspondences between concepts in L and L'.

3.2. Proposed improvements

Our proposed improvements are based on the selected AddIntent algorithm. In this subsection, we introduce these improvements one by one, and give thorough descriptions of them.

A quick look at the introduction of incremental algorithms in Section 3.1 reveals that canonical generators are of key importance to generate new concepts. To create a new concept whose intent is Y, AddIntent searches for the canonical generator by traversing the Hasse diagram in a bottom-up approach. The procedure first starts from any concept whose intent is a superset of Y, then moves on to one of its upper neighbors whose intent is also a superset of Y. The loop terminates when a concept C has no such neighbor whose intent contains Y, which means C is the canonical generator. AddIntent can find the canonical generator in a diagram graph in at most $O(|G|^2|M|)$ time (Van Der Merwe et al., 2004).

However, traversing the Hasse diagram could become rather time consuming along with the number of objects gradually growing. Additionally, AddIntent goes through the same procedure while Y is already the intent of some concept. Our improvement on how to find a canonical generator is based on the following Proposition 3 and its corollary that demonstrate the essential correspondence between canonical generators and new concepts. To indicate that changes have been made to a formal context, we use I and I' to represent the incidence relation before and after adding a new object, respectively.

Proposition 3. *Given an attribute set $Y \subseteq g^{\uparrow I'}$, Y is the intent of a new concept in L' if and only if $Y^{\downarrow I \uparrow I} \cap g^{\uparrow I'} = Y$ and $Y \neq Y^{\downarrow I \uparrow I}$.*

Proof. Necessity: Y being the intent of a new concept means there is no concept with intent equal to Y in L. Therefore, we directly get $Y \neq Y^{\downarrow I \uparrow I}$ according to Corollary 2.

Since $Y \subseteq g^{\uparrow I'}$, we have $\{g^{\uparrow I'}\}^{\downarrow I'} \subseteq Y^{\downarrow I'}$ based on Proposition 1, which leads us to $g \in Y^{\downarrow I'}$. Now that $Y^{\downarrow I'}$ and $Y^{\downarrow I}$ are the sets of objects sharing Y in L and L' respectively, and the only change made to the context is adding the new object g, we can easily deduce that $Y^{\downarrow I'} = \{g\} \cup Y^{\downarrow I}$. Then we get

$$Y^{\downarrow I'} = Y^{\downarrow I} \cup \{g\} \Rightarrow Y^{\downarrow I' \uparrow I'} = \{Y^{\downarrow I} \cup \{g\}\}^{\uparrow I'} \Rightarrow Y^{\downarrow I' \uparrow I'}$$
$$= Y^{\downarrow I \uparrow I'} \cap g^{\uparrow I'} \quad \text{(Propostition 2)}$$

As the attribute set of the formal context does not change during the update, we can see that every object has the same attribute set before and after update, which implies $(Y^{\downarrow I})^{\uparrow I'} = (Y^{\downarrow I})^{\uparrow I}$. Then we have

$$Y = Y^{\downarrow I' \uparrow I'} \quad \text{(Corollary 2)}$$
$$= Y^{\downarrow I \uparrow I'} \cap g^{\uparrow I'}$$
$$= Y^{\downarrow I \uparrow I} \cap g^{\uparrow I'}$$

Sufficiency: Since $Y \neq Y^{\downarrow I \uparrow I}$, there is no concept with intent equal to Y in L. From the above proof, we learn that $Y \subseteq g^{\uparrow I'}$ implies $Y^{\downarrow I' \uparrow I'} =$

$(Y^{\downarrow I})^{\uparrow I'} \cap g^{\uparrow I'}$, and $(Y^{\downarrow I})^{\uparrow I'} = (Y^{\downarrow I})^{\uparrow I}$. It yields $Y^{\downarrow I'\uparrow I'} = (Y^{\downarrow I})^{\uparrow I} \cap g^{\uparrow I'}$. Given that $Y^{\downarrow I\uparrow I} \cap g^{\uparrow I'} = Y$, we can deduce that $Y^{\downarrow I'\uparrow I'} = Y$. Thus, Y is the intent of a new concept in L'. □

As a consequence of Proposition 3, there must be one generator $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ for every new concept $(Y^{\downarrow I'}, Y)$. According to Corollary 1 and Definition 4, we know $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ is a superconcept of all generators of $(Y^{\downarrow I'}, Y)$, which gives us the following corollary.

Corollary 3. *If $(Y^{\downarrow I'}, Y)$ is a new concept, then $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ is the canonical generator of $(Y^{\downarrow I'}, Y)$.*

The observation of Corollary 3 allows us to compute the canonical generator of a new concept with intent equal to Y in two steps. The first step uses the following formula to get the extent according to Proposition 2:

$$Y^{\downarrow I} = \bigcap_{a \in Y} \{a\}^{\downarrow I} \tag{1}$$

To get the intent, the second step tests if

$$Y^{\downarrow I} \subseteq \{m\}^{\downarrow I} \tag{2}$$

holds, where m is an attribute in M but not in Y. If (2) holds, we know $m \in Y^{\downarrow I\uparrow I}$ based on (7) in Proposition 1. After those two steps, we can get $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ and then use $Y^{\downarrow I\uparrow I}$ as a key to look for the canonical generator in a proper data structure such as a balanced tree and a hash table. The whole procedure is presented in the following algorithm.

Algorithm 1. Procedure *GetClosureConcept*(Y, *generator*, L):

```
1: if L.Find(Y) ≠ ∅ then:
2:   return L.Find(Y)
3: end if
4: closureExtent = ∅
5: for each attribute in Y:
6:   if closureExtent = = ∅ then:
7:     closureExtent = {attribute}^↓I
8:   else:
9:     closureExtent = closureExtent ∩ {attribute}^↓I
10:  end if
11: end for
12: closureIntent = Y
13: for each m in generator.intent-Y:
14:   if closureExtent ⊆ {m}^↓I then:
15:     closureIntent = closureIntent ∪ {m}
16:   end if
17: end for
18: return L.Find(closureIntent)
```

The procedure *GetClosureConcept* accepts three arguments: an attribute set Y, a concept *generator* whose intent contains Y and a concept lattice L. *GetClosureConcept* returns a concept whose intent is equal to $Y^{\downarrow I\uparrow I}$. Note that the method *L*.Find used in Algorithm 1 is just a wrapper of the lookup function provided by those data structures that we use to store concepts.

If Y is already the intent of a concept in L, computing $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ is not necessary. Hence Algorithm 1 first tries to look for the concept with intent equal to Y, and return that concept if succeeds (line 1–3). If the search fails, the extent $Y^{\downarrow I}$ of the canonical generator is computed using (1) (line 4–11). Then, it tests every attribute m in (*generator*.intent-Y) using (2), and m belongs to $Y^{\downarrow I\uparrow I}$ if the test succeeds (line 12–17). In the end, the algorithm looks for $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ using $Y^{\downarrow I\uparrow I}$ as a key, and returns it to the caller.

After creating a new concept, the next step is to find its upper neighbors. For a new concept (X, Y) and its corresponding canonical generator c_g, AddIntent can determine that all upper neighbors of (X, Y) are in the set $C_u = \{(D^{\downarrow I'}, D) \mid D = F \cap Y$, where $(F^{\downarrow I}, F)$ could be any upper neighbor of c_g in $L\}$. It is clear that the cardinality of C_u cannot be greater than the number of upper neighbors of c_g. Since each concept in a concept lattice could only have $|G|$ upper neighbors at most (Van der Merwe, 2006; Van Der Merwe et al., 2004), we can see that $|C_u| \leqslant |G|$. For every $c \in C_u$, AddIntent compares c with every other concept in C_u to determine whether c is an actual upper neighbor of (X, Y). If there exists a concept $c_0 \in C_u$ with $c_0 \leqslant c$ and $c_0 \neq c$, c is eliminated from further consideration. Otherwise, c is a parent of (X, Y).

Like AddIntent, our algorithm uses the same scope limited by C_u to search for actual parents of the new concept (X, Y). However, we do not compare $c \in C_u$ with other elements in C_u. Instead, we take advantage of two straightforward facts to improve the procedure. One of them is that c being an upper neighbor of (X, Y) is equivalent to (X, Y) being a lower neighbor of c, and the other is given by the following Proposition 4 and its corollary.

Proposition 4. *$K = (G, M, I)$ is a formal context. (A, B) is a formal concept of K with $|B| < |M|$. Then every lower neighbor of (A, B) is in the set $C_d = \{(D^{\downarrow I}, D^{\downarrow I\uparrow I}) \mid D \in M_d\}$, where $M_d = \{F \mid B \subset F$ and $|F| = |B|+1\}$.*

Proof. $|B| < |M|$ means that (A, B) has at least one lower neighbor. Assume that (A_1, B_1) is a lower neighbor of (A, B), which implies $B \subset B_1$ and $|B| < |B_1|$. Hence, we have the following two cases.

(i) If $|B|+1 = |B_1|$, then obviously $B_1 \in M_d$, namely $(B_1^{\downarrow I}, B_1^{\downarrow I\uparrow I}) \in C_d$. Thus, from Corollary 2 it follows that $B_1 = B_1^{\downarrow I\uparrow I}$ and $A_1 = B_1^{\downarrow I}$, which leads directly to $(A_1, B_1) \in C_d$.

(ii) If $|B|+1 < |B_1|$, there must exist an attribute set $B_2 \in M_d$ such that $B_2 \subset B_1$. Due to (1) and (2) in Proposition 1, we obtain

$$\begin{aligned} B_2 \subset B_1 &\Rightarrow B_2^{\downarrow I} \supseteq B_1^{\downarrow I} \\ &\Rightarrow B_2^{\downarrow I\uparrow I} \subseteq B_1^{\downarrow I\uparrow I} \\ &\Rightarrow B_2^{\downarrow I\uparrow I} \subseteq B_1 \end{aligned}$$

From the definition of M_d it follows that $B \subset B_2$. Moreover, we know $B_2 \subseteq B_2^{\downarrow I\uparrow I}$ according to (4) in Proposition 1. Putting those two conditions together, we get $B \subset B_2^{\downarrow I\uparrow I}$. As a consequence, we have $B \subset B_2^{\downarrow I\uparrow I} \subseteq B_1$. Since (A_1, B_1) is a lower neighbor of (A, B), there is no concept between them. It shows that $B \subset B_2^{\downarrow I\uparrow I} \subseteq B_1$ holds because $B_2^{\downarrow I\uparrow I} = B_1$. From (5) in Proposition 1 it follows that $B_2^{\downarrow I} = (B_2^{\downarrow I\uparrow I})^{\downarrow I} = B_1^{\downarrow I} = A_1$. Therefore, $(A_1, B_1) = (B_2^{\downarrow I}, B_2^{\downarrow I\uparrow I}) \in C_d$.In conclusion, $(A_1, B_1) \in C_d$ in both cases. Because of the generality of (A_1, B_1), we know every lower neighbor of (A, B) is in C_d. □

It is clear that $|C_d|$ cannot be greater than $|M_d|$. From the definition of M_d it follows that $|M_d| = |M| - |B|$, so $|C_d| \leqslant |M|$. Proposition 4 covers every concept except the bottom concept with intent equal to M. In that case, it is straightforward to see that $C_d = \varnothing$. Therefore, we have the following corollary:

Corollary 4. *The number of lower neighbors of any concept never exceeds $|M|$.*

In order to determine whether a concept $c \in C_u$ is a real upper neighbor of (X, Y), our algorithm compares (X, Y) with every lower neighbor of c. If there is a child c_1 of c that satisfies $(X, Y) \leqslant c_1$ and $(X, Y) \neq c_1$, c could not be a parent of (X, Y). Performing such tests on all elements in C_u would take $O(|G||M|^2)$ time at most according to the result shown in Corollary 4, while AddIntent takes $O(|G|^2|M|)$

time in the worst case to select all actual upper neighbors from C_u. Our improvement is represented by the following recursive procedure.

Algorithm 2: Procedure *FastAddIntent*(*Y*, *generator*, *L*)

```
1: generator = GetClosureConcept(Y, generator, L)
2: if generator.Intent == Y then:
3:   return generator
4: end if
5: GeneratorParents = generator.Parents
6: newParents = ∅
7: for each candidate in GeneratorParents:
8:   if candidate.Intent ⊄ Y then:
9:     candidate = FastAddIntent(candidate.Intent ∩
   Y,candidate,L)
10:  end if
11:  Add candidate to newParents
12: end for
13: newConcept = (generator.Extent,Y)
14: L = L ∪ {newConcept}
15: for each parent in newParents:
16:  isTrueParent = True
17:  chidren = parent.Children
18:  for each child in children:
19:    if child.Intent ⊆ Y then:
20:      isTrueParent = False
21:      break
22:    end if
23:  end for
24:  if isTrueParent then:
25:    Remove links between generator and parent
26:    Set links between newConcept and parent
27:  end if
28: end for
29: Set links between generator and newConcept
30: return newConcept
```

The procedure *FastAddIntent* takes three arguments: an attribute set *Y*, a concept *generator* whose intent contains *Y*, and a concept lattice *L*. If there is a concept with intent equal to *Y* in *L*, *FastAddIntent* returns that concept. Otherwise, a new concept with intent equal to *Y* will be created, added to *L* and returned by the procedure.

First, Algorithm 2 searches for the concept $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ and assigns it to *generator* (line 1). If *Y* equals $Y^{\downarrow I\uparrow I}$, the concept with intent equal to *Y* is already in the lattice and the algorithm terminates (line 2–4). Otherwise, *generator* is the desired canonical generator. Recall that all parents of the new concept is in the set C_u. In Algorithm 2, the set C_u is computed and stored in *newParents* (line 5–12). After that, a concept *newConcept* with intent equal to *Y* is created and added to *L* (line 13–14). In order to find real parents of *newConcept*, we test every candidate in *newParents* (line 15). As we mentioned earlier, we compare *newConcept* with all children of each element in *newParents* (line 17–23) and link those qualified concepts with *newConcept* (line 24–27). In the end, a link between *newConcept* and $(Y^{\downarrow I}, Y^{\downarrow I\uparrow I})$ is established (line 29), and the procedure returns *newConcept* (line 30).

For the convenience of comparison with AddIntent, Algorithm 2 described above does not include the operation of updating extents. Such an operation is performed in the following procedure for constructing the overall lattice:

Algorithm 3: Procedure *CreateLatticeIncrementally*(*G*, *M*, *I*)

```
1: bottomConcept = (∅, M)
2: L = {bottomConcept}
3: for each g in G:
4:   objectConcept = FastAddIntent(g^{↑I'}, bottomConcept, L)
5:   Add g to the extent of objectConcept and all concepts
   above
6: end for
7: return L
```

The procedure *CreateLatticeIncrementally* invokes *FastAddIntent* with each object in *G*. After every call to *FastAddIntent*, a concept *objectConcept* with intent equal to $g^{\uparrow I'}$ is found or generated. If *objectConcept* is a new concept, every new concept above *objectConcept* are created and added to *L* in the procedure *FastAddIntent*. Then we update the extent of *objectConcept* and every concept above.

Even though our proposed algorithm is a refinement of AddIntent, the main structure of AddIntent remains intact. In addition, all procedures in both algorithms can be used interchangeably. Therefore, the correctness of the proposed algorithm follows the correctness of the AddIntent algorithm and the above descriptions of our proposed improvements. The reader is kindly referred to Kourie et al. (2009) and Van der Merwe (2006) for the correctness of the AddIntent algorithm.

4. Complexity issues

Constructing the concept lattice of a formal context (*G*, *M*, *I*) by invoking the *CreateLatticeIncrementally* procedure has a worst-case time complexity bound of $O(|L||G||M|^2)$. We give our main argument about asymptotic time complexity as follows.

The complexity depends on the execution times of the main loop in Algorithm 3 (line 3–6). That loop is basically to call the *FastAddIntent* procedure, which is executed every time we introduce a new object. But taking a closer look at Algorithm 2 reveals that line 5–30 are only used for generating new concepts, while adding new objects to a formal context does not result in removal of concepts already in *L* (Outrata, 2013). That means the process of creating a new concept can only be performed once for each concept in *L*. Recall that the number of lower neighbors of any concept never exceeds $|M|$ and every concept could have $|G|$ upper neighbors at most. Therefore, during the construction of *L* the *FastAddIntent* procedure takes at most $T_0 + O(|L||G||M|^2)$ time, where T_0 is the execution time of the *GetClosureConcept* procedure.

Analysis of the *GetClosureConcept* procedure is similar to *FastAddIntent*. The process of computing canonical generators only takes place when new concepts are about to be created. In other cases, if a concept with intent equal to *Y* is already in *L*, *GetClosureConcept* would find and return that concept. To quickly find a concept, a balanced tree, a trie (Fredkin, 1960) or a hash table is usually used to store concepts (Krajca & Vychodil, 2009). Here, we assume a balanced tree is implemented. That gives us $O(|M|^2)$ complexity of searching for a concept using an attribute set as a key (including comparisons between keys), because the total number of concepts never exceeds $2^{|M|}$ (Albano & Do Lago, 2014). Hence, the overall time complexity of *GetClosureConcept* during the construction of *L* can also be estimated as $O(|L||G||M|^2)$.

We substitute $O(|L||G||M|^2)$ for T_0 to get a total complexity of $O(|L||G||M|^2)$ as stated above. Comparing to AddIntent whose complexity bound is $O(|L||G|^2|M|)$, FastAddIntent is more suitable for

large formal contexts because $|G|$ is usually much bigger than $|M|$ in those cases. The experimental results in Section 5 are also consistent with our theoretical analysis. Note that if we just want to build a lattice from scratch (instead of updating one) in which case either G or M can be considered as fixed, AddIntent can be applied to the transposed data matrix to achieve the same complexity that our proposed algorithm has.

5. Experimental evaluation

We have implemented our algorithm and AddIntent in pure Python (version 3.4). For the sake of comparison, we use two different versions of FastAddIntent for the tests. One version employs an AVL tree (AdelsonVelskii & Landis, 1963) to store concepts, and the other employs a hash table. The experiments were run on an idle 64-bit system (Intel Core i3–4130, 3.4 GHz, 4 GB RAM).

All tests are divided into two groups. In the first group of experiments, we have used contexts that are randomly generated datasets with different fill ratio (i.e. $|I|/|G||M|$). These datasets have 50 attributes, but the number of objects varies, and each object may have different number of attributes.

Fig. 3 depicts the running time of FastAddIntent and AddIntent on random datasets with low density (10% fill ratio). When $|G|$ is not big enough, the difference between FastAddIntent and AddIntent is also small. With $|G|$ growing, FastAddIntent using a hash table gradually takes the lead while FastAddIntent using an AVL tree has small advantages over AddIntent.

Fig. 4 illustrates the running time of FastAddIntent and AddIntent on random datasets with relatively medium density (25% fill ratio). The performance gap between FastAddIntent and AddIntent is much bigger in Fig. 4 than that in Fig. 3, and FastAddIntent takes the lead earlier when $|G|$ is around 700.

Fig. 5 displays the running time of FastAddIntent and AddIntent on random datasets with relatively high density (40% fill ratio). Due to lattices of dense contexts consuming the memory particularly fast, we can only perform tests when $|G|$ is not too big. But duly note two facts in Fig. 5: comparing to AddIntent, FastAddIntent with an AVL tree takes the lead when $|G|$ is only about 300 which is much smaller than those in previous tests; and FastAddIntent with a hash table significantly outperforms AddIntent at almost all test points.

The second set of experiments were done with several real benchmark datasets selected from the UCI Machine Learning Repository (Bache & Lichman, 2013). Since most of those datasets are many-valued contexts, we have used a formal context creator named FcaBedrock (Andrews & Orphanides, 2010) to scale them into one-valued contexts. As a consequence, some of our scaled datasets and those in other publications may have different attributes and concepts. Table 2 gives the results for running time (in seconds) of FastAddIntent and AddIntent along with basic information on used datasets.

From Table 2 we can find the performance of FastAddIntent is comparable to the performance of AddIntent when $|G|$ is relatively small. But when $|G|$ becomes the main factor or contexts are dense,

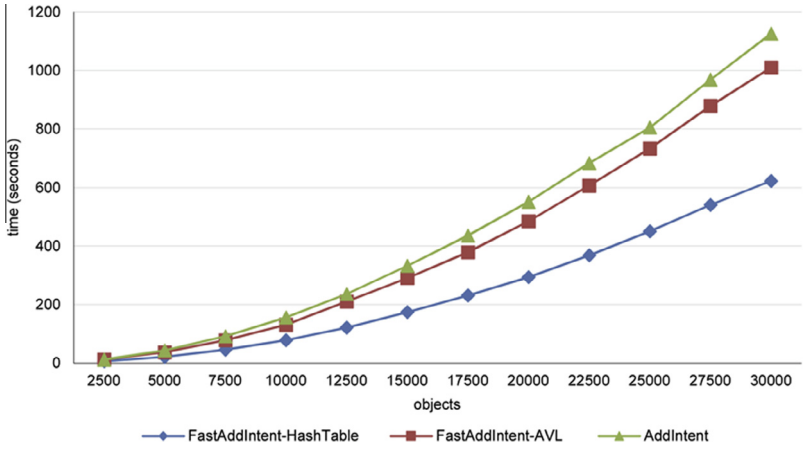

Fig. 3. Results for random datasets with low density.

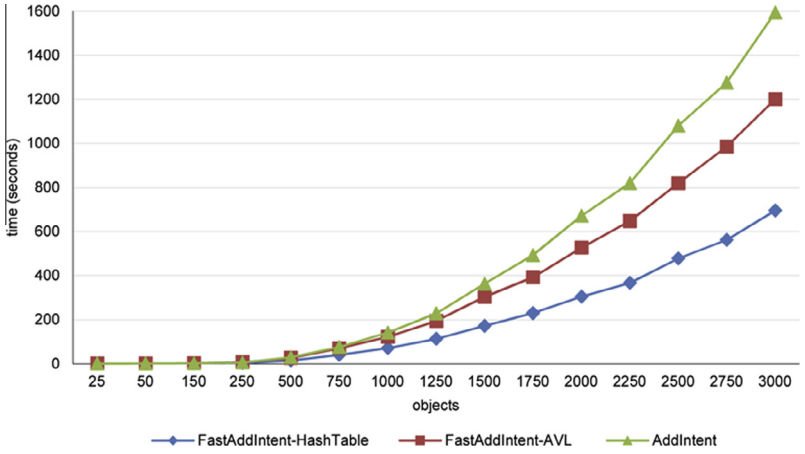

Fig. 4. Results for random datasets with medium density.

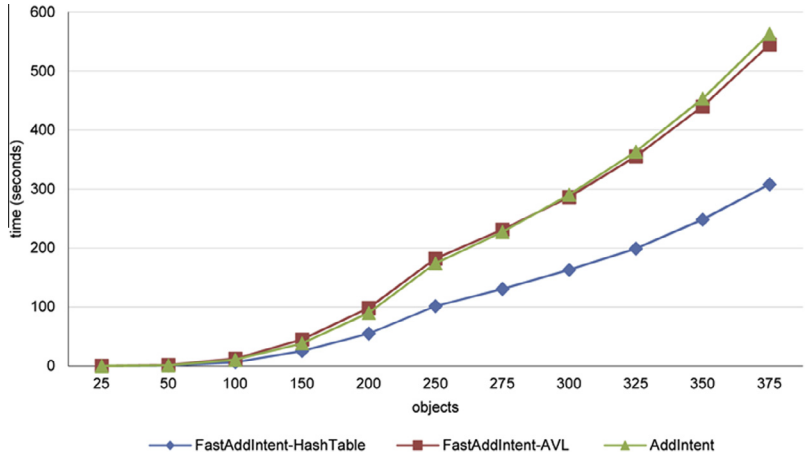

Fig. 5. Results for random datasets with high density.

Table 2
Performance (in seconds) on real datasets.

Dataset	Size	Fill ratio	#concepts in total	FastAddIntent-HashTable	FastAddIntent-AVL	AddIntent
SPECT	267 × 23	33.25%	21550	1.123	2.044	1.061
Sola-flare	1389 × 49	26.53%	28742	3.869	5.413	4.072
Car	1728 × 25	28.00%	12640	2.980	7.051	9.407
Adult	48842 × 124	7.14%	40894	54.210	57.861	42.167
Mushroom	8124 × 119	19.33%	238710	180.727	293.781	505.225
Nursery	12960 × 32	28.13%	183079	272.924	531.433	871.921

both versions of FastAddIntent have obvious advantages. Particularly, one may find both new versions have worse performance when applied to the "adult" dataset. The explanation is actually quite simple. As the complexity $O(|L||G||M|^2)$ suggests, our proposed algorithm depends heavily on the cardinality of M. In addition, Fig. 3 shows that $|G|$ needs to be hundreds of times greater than $|M|$ for our algorithm to have advantages when a dataset has a low fill ratio. The "adult" dataset has the lowest fill ratio (7.14%) and the highest number of attributes (i.e. $|M| = 124$) among all real datasets we tested. Hence, it is still reasonable for our algorithm to be slower than AddIntent when applied to the "adult" dataset, even the dataset has the highest number of objects (i.e. $|G| = 48842$).

6. Conclusions

In real-life practices, alteration of a formal context is inevitable in most cases. It would bring changes to the corresponding concept lattice, which can be tackled by incremental algorithms. In this paper, We have proposed an algorithm called FastAddIntent for constructing concept lattices incrementally. The algorithm is derived from AddIntent by introducing a new approach to search for canonical generators and improving the process of updating the covering relation.

We have proved the correctness of our improvements and given theoretical bases for them. The complexity analysis predicts that our algorithm would perform better than AddIntent while the number of objects continues growing. Experimental evaluation has also shown consistence with our theoretical analysis. When we use an AVL tree to store concepts in FastAddIntent, it has considerably advantages over AddIntent if a dataset is relatively dense, and we can see that it would take a large number of objects for FastAddIntent to take the lead if a dataset is sparse. When a hash table is used to save concepts in FastAddIntent, our algorithm performs significantly better than AddIntent at almost all test points, and it can outperform its competitors rather quick even if a dataset is sparse. The results for synthetic datasets also suggest that both versions of our proposed algorithm will catch up and surpass AddIntent at some point. More importantly, the gap between AddIntent and our algorithm grows along with increase in $|G|$. Both theoretical analysis and performance tests encourage us to use FastAddIntent as a better alternative especially when we apply FCA methods to contexts with a relatively large number of objects or (and) high density.

The topics for future research include:

1. A more extensive set of experiments could be conducted to find out how both new versions perform when building a lattice with more than two million concepts.
2. A parallelization of the presented algorithm (or the AddIntent algorithm).
3. Further theoretical and experimental study of the relationship between the density of a dataset and the performance of our proposed algorithm.
4. Experimental comparisons with classical non-incremental algorithms.

Acknowledgement

This research has been supported by the National Natural Science Foundation of China (NSFC Grant No. 61379109) and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20120162110077).

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.eswa.2015.01.044.

References

AdelsonVelskii, M. & Landis, E. M. (1963). An algorithm for the organization of information.

Albano, A., & Do Lago, A. P. (2014). A convexity upper bound for the number of maximal bicliques of a bipartite graph. *Discrete Applied Mathematics, 165*, 12–24.

Amin, I. I., Kassim, S. K. & Hefny, H. A. (2013). Using formal concept analysis for mining hyomethylated genes among breast cancer tumors subtypes. In *2013 International conference on advances in computing, communications and informatics (ICACCI)* (pp. 521–526).

Andrews, S. & Orphanides, C. (2010). FcaBedrock, a formal context creator. In *Conceptual structures: from information to intelligence* (pp. 181–184). Springer.

Babin, M. A. & Kuznetsov, S. O. (2010). Recognizing Pseudo-intents is coNP-complete. In *CLA* (pp. 294–301).

Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml.

Carpineto, C., & Romano, G. (2004). *Concept data analysis: Theory and applications*. John Wiley & Sons.

Dau, F., Ducrou, J., & Eklund, P. (2008). Concept similarity and related categories in SearchSleuth. In *Conceptual structures: Knowledge visualization and reasoning* (pp. 255–268). Springer.

Fredkin, E. (1960). Trie memory. *Communications of the ACM, 3*(9), 490–499.

Ganter, B. (2010). *Two basic algorithms in concept analysis*. Springer.

Ganter, B. & Wille, R. (1999). Formal concept analysis: Mathematical foundations 10.

Godin, R., Missaoui, R., & Alaoui, H. (1995). Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence, 11*(2), 246–267.

Kourie, D. G., Obiedkov, S., Watson, B. W., & van der Merwe, D. (2009). An incremental algorithm to construct a lattice of set intersections. *Science of Computer Programming, 74*(3), 128–142.

Krajca, P., & Vychodil, V. (2009). Comparison of data structures for computing formal concepts. In *Modeling Decisions for Artificial Intelligence* (pp. 114–125). Springer.

Kriegel, F. (2014). Incremental computation of concept diagrams. *Studia Univ. Babes-Bolyai, Informatica, 59*(1), 1–15.

Kuznetsov, S. O. (1993). A fast algorithm for computing all intersections of objects in a finite semi-lattice. *Automatic Documentation and Mathematical Linguistics, 27*(5), 11–21.

Kuznetsov, S. O. (2001). On computing the size of a lattice and related decision problems. *Order, 18*(4), 313–321.

Kuznetsov, S. O. (2004). On the intractability of computing the duquenne- guigues base. *Journal of Universal Computer Science, 10*(8), 927–933.

Lingling, L., Lei, Z., Anfu, Z. & Funa, Z. (2011). An improved addintent algorithm for building concept lattice. In *(ICICIP), 2011 2nd International Conference on Intelligent Control and Information Processing Vol. 1* (pp. 161–165).

Maio, C. D., Fenza, G., Gaeta, M., Loia, V., Orciuoli, F., & Senatore, S. (2012). RSS-based e-learning recommendations exploiting fuzzy FCA for Knowledge Modeling. *Applied Soft Computing, 12*(1), 113–124.

Outrata, J. (2013). A lattice-free concept lattice update algorithm based on* CbO★. *CLA, 2013*, 261–274.

Outrata, J., & Vychodil, V. (2012). Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data. *Information Sciences, 185*(1), 114–127.

Poelmans, J., Elzinga, P., Viaene, S., & Dedene, G. (2010). Formal concept analysis in knowledge discovery: A survey. In *Conceptual Structures: From Information to Intelligence* (pp. 139–153). Springer.

Poelmans, J., Ignatov, D. I., Kuznetsov, S. O., & Dedene, G. (2013a). Formal concept analysis in knowledge processing: A survey on applications. *Expert Systems with Applications, 40*(16), 6538–6560.

Poelmans, J., Kuznetsov, S. O., Ignatov, D. I., Dedene, G., Elzinga, P., & Viaene, S. (2013b). Formal concept analysis in knowledge processing: A survey on models and techniques. *Expert Systems with Applications, 40*(16), 6601–6623.

Priss, U. (2005). Linguistic applications of formal concept analysis. In *Formal Concept Analysis* (pp. 149–160). Springer.

Valtchev, P., & Missaoui, R. (2001). Building concept (Galois) lattices from parts: generalizing the incremental methods. In *Conceptual structures: Broadening the base* (pp. 290–303). Springer.

Valtchev, P., Missaoui, R., & Godin, R. (2008). A framework for incremental generation of closed itemsets. *Discrete Applied Mathematics, 156*(6), 924–949.

Van der Merwe, F. J. (2006). *Constructing concept lattices and compressed pseudo-lattices* 10.

Van Der Merwe, D., Obiedkov, S., & Kourie, D. (2004). AddIntent: A new incremental algorithm for constructing concept lattices. In *Concept Lattices* (pp. 372–385). Springer.

Wermelinger, M., Yu, Y. & Strohmaier, M. (2009). Using formal concept analysis to construct and visualise hierarchies of socio-technical relations. In *31st International Conference on Software Engineering-Companion Volume, 2009. ICSE-Companion 2009.* (pp. 327–330).

Wille, R. (1982). Restructuring lattice theory: An approach based on hierarchies of concepts. In *Ordered Sets* (pp. 445–470). Springer.

Wille, R. (2009). *Restructuring lattice theory: An approach based on hierarchies of concepts*. Springer.