AddIntent: A New Incremental Algorithm for Constructing Concept Lattices

Dean van der Merwe[1], Sergei Obiedkov[2], and Derrick Kourie[1]

[1] Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa Deon.vd.Merwe@bentleywest.com, DKourie@cs.up.ac.za
[2] Institute für Algebra, TU Dresden, 01062 Dresden, Germany s-obj@yandex.ru

Abstract. An incremental concept lattice construction algorithm, called AddIntent, is proposed. In experimental comparison, AddIntent outperformed a selection of other published algorithms for most types of contexts and was close to the most efficient algorithm in other cases. The current best estimate for the algorithm's upper bound complexity to construct a concept lattice L whose context has a set of objects G, each of which possesses at most $\max(|g'|)$ attributes, is $O(|L||G|^2 max(|g'|))$.

1 Introduction

In Formal Concept Analysis (FCA) [1, 2], the problem of generating the set of all concepts of a formal context and constructing the diagram graph (covering relation) of the concept lattice has been well studied [3–7]. See [8] for an overview and comparison. Since lattices can grow exponentially large, leading to correspondingly large computational requirements, efficient algorithms for the construction of concept lattices are of key importance.

In this text, we introduce a new lattice construction algorithm, called AddIntent, an earlier version of which appeared in [9] under the name AddAtom. The algorithm produces not only the concept set, but also the diagram graph. Being incremental, it relies on the graph constructed from the first objects of the context to integrate the next object into the lattice. Therefore, its use is most appropriate in those applications that require both the concept set and diagram graph, for example, in applications related to information retrieval and document browsing [10]. Nevertheless, experiments show that sometimes it takes other algorithms longer merely to construct the concept set of a context than it takes AddIntent to construct the entire diagram graph of the same context.

The upper bound complexity estimate of the algorithm is linear in the lattice size modulo some factor polynomial in the input size. Although the current estimate of the polynomial factor is higher than that of the lowest known for lattice construction algorithms (i.e., that of [11]), experimental comparison indicates that, in practice, AddIntent performs well in constructing lattices from many different contexts. It outperforms other algorithms in most cases and may therefore be considered as the best candidate for a universal lattice construction algorithm for diagram graphs amongst the algorithms considered in this study.

2 Main Definitions

This section defines the basic terminology and notation of Formal Concept Analysis [1].

Definition 1. *A* formal context *is a triple of sets (G, M, I), where G is called a set of objects, M is called a set of attributes, and $I \subseteq G \times M$. The notation gIm indicates that $(g, m) \in I$ and denotes the fact that the object g possesses the attribute m.*

Definition 2. *For $A \subseteq G$ and $B \subseteq M$:*
$A' = \{m \in M | \forall g \in A(gIm)\}$ *and* $B' = \{g \in G | \forall m \in M(gIm)\}$.
The operator $''$ is a closure operator (to be precise, $''$ is a homonymous denotation of two closure operators: $2^G \to 2^G$ and $2^M \to 2^M$.

Definition 3. *A formal concept of a formal context (G, M, I) is a pair (A, B), where $A \subseteq G, B \subseteq M$, $A' = B$, and $B' = A$. The set A is called the* extent*, and the set B is called the* intent *of the concept (A, B).*

For $g \in G$ and $m \in M$, $\{g\}'$ is denoted by g' and called an object intent*, and $\{m\}'$ is denoted by m' and called an* attribute extent*.*

Definition 4. *For a context (G, M, I), a concept $X = (A, B)$ is* less general than *or* equal to *a concept $Y = (C, D)$ (or $X \leq Y$) if $A \subseteq C$ or, equivalently, $D \subseteq B$.*

Definition 5. *For two concepts X and Y, if $X \leq Y$ and there is no concept Z with $Z \neq X$, $Z \neq Y$, $X \leq Z \leq Y$, the concept X is called a* lower neighbour *(or a* child*) of Y and Y is called an* upper neighbour *(or a* parent*) of X. This relationship is denoted by $X \prec Y$.*

Definition 6. *We call the (directed) graph of the relation $\prec$ a* diagram graph*. A plane embedding of a diagram graph where a concept has larger vertical coordinate than that of any of its lower neighbors is called a* line *(*Hasse*)* diagram*.*

3 The AddIntent Algorithm

In this section, we define the AddIntent algorithm and describe the basic strategy it follows. Readers are referred to [12] for a more detailed discussion.

AddIntent is an incremental algorithm, i.e., it takes as input the lattice L_i produced by the first i objects of the context and inserts the next object g to generate a new lattice L_{i+1}. As observed in [5, 7] lattice construction can be described in terms of four sets of concepts: *modified concepts*, *generator concepts*, *new concepts* and *old concepts*.

A concept $(C, D) \in L_{i+1}$ is *new* if D is not an intent of any concept in L_i. We call a concept $(A, B) \in L_i$ *modified* if $B \subseteq g'$ since g has to be added to its extent in L_{i+1}. Otherwise, $B \cap g' = D \neq B$ for some concept $(C, D) \in L_{i+1}$.

If (C, D) is a new concept, then (A, B) is called a *generator* of (C, D); if not, (A, B) is called *old*.

Every new concept (C, D) has at least one generator, but it may have several. The (unique) most general of these generators is called the *canonical generator* of (C, D). The remaining generators of (C, D) are naturally called its *non-canonical* generators. Obviously, there is a one-to-one correspondence between canonical generators and new concepts.

The key problem of incremental construction is how to identify all modified concepts (in order to add g to their extents) and all canonical generators of new concepts (in order to generate every new concept exactly once). Efficient algorithms will spend as little effort as possible searching through the unmodified and non-canonical generators. AddIntent approaches this problem by traversing the diagram graph of L_i in a recursive fashion.

There are several ways to identify canonical generators. The algorithm in [5] processes the list of concepts in L_i starting with the most general ones. In processing a concept (A, B), it generates the intent $B \cap g'$ and then looks through the set of new concepts produced so far (and arranged in "buckets" according to the cardinality of their intents) to see whether such an intent is already there. The algorithm of Norris [6], in processing a concept (A, B), checks whether $B \cap g' \subseteq h'$ for any $h \in Gi \backslash A$ (assuming that G_i is the set of objects processed up to that moment); if so, then A is not the maximal extent, and hence (A, B) is not the most general concept capable of generating $B \cap g'$. What both algorithms ignore are the following facts:

Proposition 1. *If (B', B) is a canonical generator of a new concept (F', F), while (D', D) is a non-canonical generator of (F', F) – in this case, $B \subset D$ – then any concept (H', H) such that $H \subset D$ and $H \not\subset B$ is neither modified nor is it a canonical generator of any new concept.*

Proposition 2. *If (D', D) is an old concept and $D \cap g' = B$ – in this case, $(B', B) \in L_i$ is modified – then any concept (H', H) such that $H \subset D$ and $H \not\subset B$ is neither modified nor is is a canonical generator of any new concept.*

Therefore, there is no need to process such concepts (H', H) in the search of canonical generators and modified concepts. Since AddIntent maintains the diagram graph (which explicitly orders concepts from most to least general) it can exclude these concepts from further consideration by simply traversing the diagram graph in a bottom-up fashion. Having found a non-canonical generator, AddIntent uses the diagram graph to find the canonical generator of the same concept. It then works only with concepts above that canonical generator, ignoring all other concepts above the non-canonical generator. The canonical generator can be found in a diagram graph in at most $O(|G|^2|M|)$ time.

These ideas are expanded in [12] where the notions of the *approximate intent representative* and *exact intent representative* are introduced in the context of so-called compressed pseudo-lattices.

Using parameter names to imply types the *AddIntent* function is defined as follows:

```
01: Function AddIntent(intent, GeneratorConcept, L)
02:   GeneratorConcept = GetMaximalConcept(intent,
                                  GeneratorConcept, L)
03:   If GeneratorConcept.Intent = intent
04:     Return GeneratorConcept
05:   End If
06:   GeneratorParents := GetParents(GeneratorConcept, L)
07:   NewParents = ∅
08:   For each Candidate in GeneratorParents
09:     If Candidate.Intent ⊄ intent
10:       Candidate := AddIntent(Candidate.Intent ∩ intent,
                                  Candidate, L)
11:     End If
12:     addParent := true
13:     For each Parent in NewParents
14:       If Candidate.Intent ⊆ Parent.Intent
15:         addParent := false
16:         Exit For
17:       Else If Parent.Intent ⊆ Candidate.Intent
18:         Remove Parent from NewParents
19:       End If
20:     End For
21:     If addParent
22:       Add Candidate to NewParents
23:     End If
24:   End For
25:   NewConcept := (GeneratorConcept.Extent, intent)
26:   L := L ∪ {NewConcept}
27:   For each Parent in NewParents
28:     RemoveLink(Parent, GeneratorConcept, L)
29:     SetLink(Parent, NewConcept, L)
30:   End For
31:   SetLink(NewConcept, GeneratorConcept, L)
32:   Return NewConcept
```

The parameters of the function *AddIntent*(*intent*, *GeneratorConcept*, L) are the *intent* of a new concept to be placed into the concept lattice L and a precomputed *GeneratorConcept*, such that *intent* is a subset of the intent of *GeneratorConcept*. *AddIntent* returns a concept whose intent corresponds to *intent* – a new concept will be created if there was no such concept before or an existing one will be returned otherwise.

First, the algorithm finds the most general concept whose intent is a superset of *intent* (line 02) and assigns it to *GeneratorConcept*. If the intent of this concept is equal to *intent*, then the desired concept is already in the lattice and the algorithm terminates (line 04). Otherwise, *GeneratorConcept* is the canon-

ical generator of the new concept, which has to be created and linked to other concepts in the diagram graph.

To find the parents of the new concept in the diagram graph, we examine all parents of *GeneratorConcept* (lines 08-24). If the intent of such a parent, called *Candidate*, is a subset of *intent*, then *Candidate* is modified. Otherwise, a recursive call to AddIntent ensures that the lattice contains a concept whose intent is equal to the intersection of *intent* and the intent of *Candidate*. This concept is assigned to *Candidate* (line 10). Then, *Candidate* is added to the (initially empty) *NewParents* list if it is minimal among its current elements (that is, has a maximal intent). At the same time, if some concept in *NewParents* is more general than *Candidate*, this concept is removed from the list (line 18). Thus, the *NewParents* list always contains incomparable (w.r.t. being more general) concepts. Moreover, in the end, it contains precisely the parents of *NewConcept* that is to be inserted (line 25). Proposition 3 below provides basis for the outlined procedure and Corollary 1 shows a way to optimize it (there is no need to test modified candidates for being minimal):

Proposition 3. *(F', F), then the parents of (F', F) in L_{i+1} are exactly the least general concepts from the set $\{(D', D) | D = F \cap H$ for some parent (H', H) of (B', B) in $L_i\}$.*

Corollary 1. *If (B', B) is the canonical generator of (F', F), then every modified parent of (B', B) in L_i is a parent of (F', F) in L_{i+1}.*

Thus, having processed the parents of *GeneratorConcept*, we create *NewConcept* with intent equal to *intent* and link it to concepts in the *NewParents* list, taking care to remove existing links between these concepts and *GeneratorConcept* (lines 27-30). Finally, *NewConcept* is set to be an upper neighbor of *GeneratorConcept* (line 31), and the *AddIntent* function returns *NewConcept*.

Note that the *AddIntent* function as described above does not update extents. Such an update is performed by the calling procedure constructing the lattice (shown below), but it can be easily integrated into *AddIntent* as well.

```
01: Procedure CreateLatticeIcrementally (G, M, I)
02:   BottomConcept := (∅, M)
03:    L := {BottomConcept}
04:   For each g in G
05:     ObjectConcept = AddIntent(g', BottomConcept, L)
06:     Add g to the extent of ObjectConcept and all concepts
          above
07:   End For
```

After adding an object of the context to the lattice via a call to *AddIntent*, the procedure updates the extents of the concepts above and including the newly generated *ObjectConcept*.

```
01: Function GetMaximalConcept(intent, GeneratorConcept, L)
02:   parentIsMaximal := true
```

```
03:   While parentIsMaximal
04:     parentIsMaximal := false
05:     Parents := GetParents(GeneratorConcept, L)
06:      For each Parent in Parents
07:       If intent ⊆ Parent.Intent
08:         GeneratorConcept := Parent
09:         parentIsMaximal := true
10:         Exit For
11:       End If
12:     End For
13:   Return GeneratorConcept
```

In the *GetMaximalConcept* function, we use the diagram graph to find the most general concept, whose intent is a superset of the input *intent*. If this concept is not *GeneratorConcept*, then it is among its predecessors; therefore, it is sufficient to examine the predecessors of *GeneratorConcept*. If a more general concept is found (line 8), no further processing of *Parents* is required and this more general concept can be tested in the next iteration of the *While* loop.

The algorithms described above admit a number of optimizations, which are discussed in [12] but are not included here due to space limitations. For example, the number of set operations in the *GetMaximalConcept* function can be reduced by pre-computing the number of attributes of g' that each concept in L has in common with g' since $|intent| = |Parent.Intent \cap g'|$ implies $intent \subseteq Parent.Intent$ (line 07). By first considering the candidate concepts whose intents yield the largest intersection with the new object intent, unnecessary traversal of the lattice can be avoided.

A worst-case time complexity bound of $O(|L||G|^3|M|)$ for the above version of the algorithm (i.e., the *CreateLatticeIcrementally* procedure) is derived in [13] (where $|L|$ is the number of concepts in the resulting lattice). A detailed discussion is not possible here due to space limitations, but the main argument is as follows. The complexity depends on the total number of invocations of the *AddIntent* function. The same intent can be passed as a parameter of *AddIntent* several times, but, clearly, the execution of *AddIntent* will go past line 05, i.e., past the call to *GetMaximalConcept*, at most once for every intent. Thus, for convenience, we may assume that *GetMaximalConcept* is called before the invocation of the *AddIntent* function, which, in its turn, is called only if the intent of the returned "maximal concept" is different from *intent*. In this setting, *AddIntent* would be called at most once for every intent of the lattice through the insertion of all objects. Since the length of the *GeneratorParents* list never exceeds $|G|$ and the complexity of *GetMaximalConcept* (as defined here) is bounded by $O(|G|^2|M|)$, the complexity of a single invocation of *AddIntent* (without a recursive call) can be estimated as $O(|G|^3|M|)$, which leads us to the total complexity of $O(|L||G|^3|M|)$ as stated above.

By introducing the optimizations referred to, this bound can be improved to $O(|L||G|^2.\max(|g'|)$ where $\max(|g'|)$ is the maximum number of attributes of any object in the context [12]. For the purpose of direct comparison with other

algorithms and since $|g'| < |M|$, we may replace $|g'|$ with $|M|$. A slightly less sharp complexity bound for the optimized algorithm is therefore $O(|L||G|^2|M|)$. The lowest available upper complexity bound of a lattice construction algorithm is that of Nourine [11] with $O((|G|+|M|)|G||L|)$ which is one factor better than the current estimate for the proposed algorithm. However, the experiments in Section 5 suggest that the latter is not as sharp as it might be, and that these bounds are not always good predictors of empirical performance.

4 An Example

Fig. 1. shows the relevant part of a concept lattice before the insertion of an object g with $g' =$ {a, b, d, e, f, g, h}. Parts of the lattice that are not shown are indicated with lines ending in small solid circles – these concepts will not be considered by AddIntent due to the focused way it searches (Propositions 1–3). Concepts are named for reference purposes and only the concept intents are shown. Concepts shaded in grey are modified concepts whilst those in black are canonical generators. Non-canonical generators are filled in with diagonal lines whilst old concepts are not shaded.

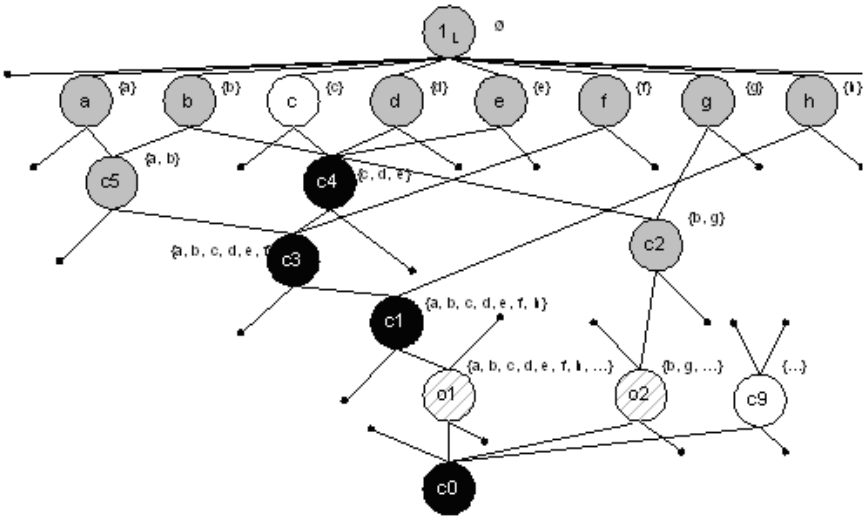

Fig. 1. Part of a concept lattice before inserting g with $g' =$ {a, b, d, e, f, g, h}.

During the initial call *AddIntent*({a, b, d, e, f, g, h}, c_0, L), the concept c_0 is already maximal with respect to *intent* and it remains the *GeneratorConcept*. The object concept o_1 will be considered as the first *Candidate* (line 8). Since it is not modified, the intersection {a, b, d, e, f, h} will be formed and a concept with such intent will be searched for or added by the recursive call *AddIntent*({a, b, d,

e, f, h}, o_1, L) (line 10). After all subsequent recursive calls have unwound, the concept c_8 will be returned and added to *NewParents* before iterating through the rest of c_0's parents. Fig. 2 contains a table that traces important variables during all the recursive calls. The level of the recursion is indicated by the first two columns. Within a recursive call, several candidates can be considered; they are shown in the third column. The last column corresponds to the state of the *NewParents* list before the current *Candidate* is considered.

Intent	**Generator (line2)**	**Candidate (line 8)**	**Candidate.Intent ∩ Intent**	**NewParents**
{a, b, d, e, f, g, h}	c_0	o_1	{a, b, d, e, f, h}	∅
{a , b, d, e, f, h}	c_1	c_3	{a, b, d, e, f}	∅
{a, b, d, e, f}	c_3	c_5	{a, b}	∅
		c_4	{d, e}	{c_5}
{d, e}	c_4	c	∅	∅
∅	1_L	–	–	–
{d, e}	c_4	d	{d}	{1_L}
		e	{e}	{d, e}
Create c_6: intent {d, e}; upper neighbors: d, e; lower neighbors: c_4				
{a, b, d, e, f}	c_3	f	{f}	{c5, c6, f}
Create c_7: intent {a, b, d, e, f}; upper neighbors: c_5, c_6, f; lower neighbors: c_3				
{a, b, d, e, f, h}	c_1	h	{h}	{c_7}
Create c_8: intent {a, b, d, e, f, h}; upper neighbors: c_7, h; lower neighbors: c_1				
{a, b, d, e, f, g, h}	c_0	o_2	{b, g}	{c_8}
{b, g}	c_2	–	–	–
Create c_{10} (g): intent {a, b, d, e, f, g, h}; upper neighbors: c_8, c_2; lower neighbors: c_0				

Fig. 2. Trace of important variables during the insertion of g into the lattice in Fig. 1.

Concept c_0 has additional upper neighbors (e.g., c_9) not included in the trace above. Since these have no attributes in their intents in common with g', there will be exactly one recursive call of *AddIntent* for every such neighbor, and every such call will immediately get us to the top concept (line 2). The resulting lattice is shown in Fig. 3.

5 Experimental Comparison

In [8], a large number of lattice construction algorithms have been studied both theoretically and experimentally. We used the same implementations of the algorithms, and the results reported here may be seen as an extension of the work. Our experimental comparison shows that AddIntent performs quite well in constructing lattices from many types of datasets as compared to other incremental and batch algorithms and is often the best performer.

The charts below show the running time for AddIntent, as well as for several other algorithms (only the most efficient or the most popular ones were included):

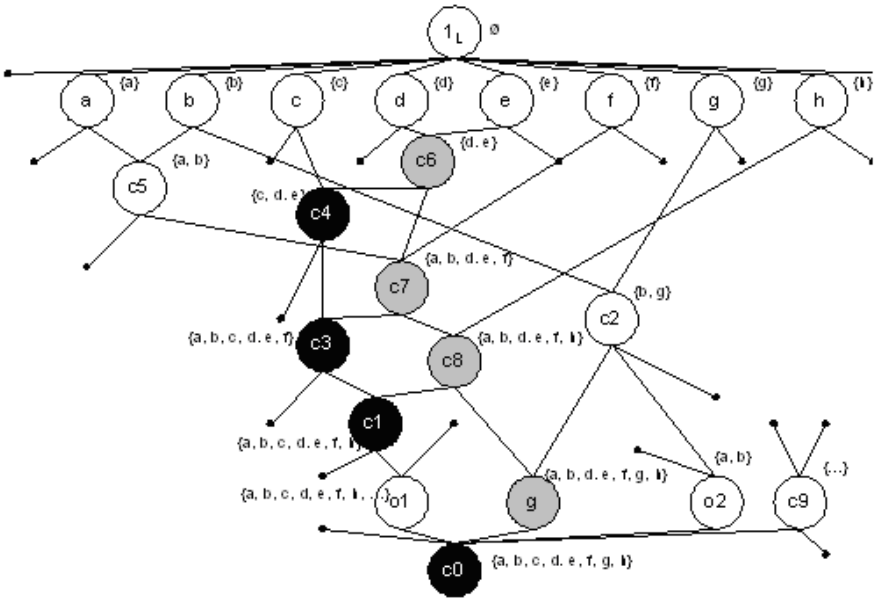

Fig. 3. The relevant part of the lattice resulting from inserting g with $g' = \{a, b, d, e, f, g, h\}$ into the lattice in Fig. 1.

Norris [6], **NextClosure** [4], a version of **Bordat** [3] from [8], **Godin** [5], and **Nourine** [11]. Some of the algorithms were modified, since, in the original version, they produced only the concept set without the diagram graph. These modifications are described in [8].

Tests were performed on a Pentium 4 - 2GHz with 1 Gigabyte RAM using both randomly generated contexts and real datasets. In all random contexts, the total number of attributes is 100, and the number of objects varies, which leads to lattices of different sizes. The number in the name of a random context denotes its density (in percent), which, in this case, is the number of attributes per object $|g'|$ (all objects of such a context have the same number of attributes). In the following charts, we plot the time against the lattice size. Points on the curves denote an increment in the size of the object set.

For the sake of consistency, the AddIntent implementation used in the comparison is that of the algorithm described in Section 3 and does not include the optimizations referred to at the end of Section 3.

On random contexts with very low density (4%), the Bordat algorithm performs the best with AddIntent coming second. According to our other experiments, the algorithm proposed in [14] is even more suitable for very sparse contexts.

On random contexts with relatively modest density (25%), the AddIntent algorithm performs the best with Godin and Bordat coming next.

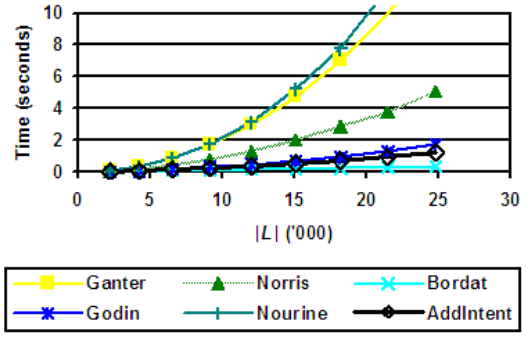

Fig. 4. Results for the Rnd-4 dataset: $|M| = 100$, $|g'| = 4$, $|G|$ varies from 100 to 900, the incremental step being 100 objects.

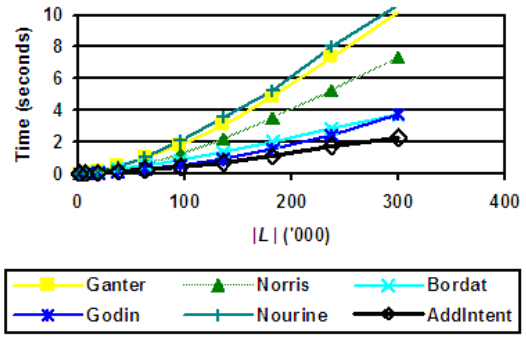

Fig. 5. Results for the Rnd-25 dataset: $|M| = 100$, $|g'| = 25$, $|G|$ varies from 10 to 100, the incremental step being 10 objects.

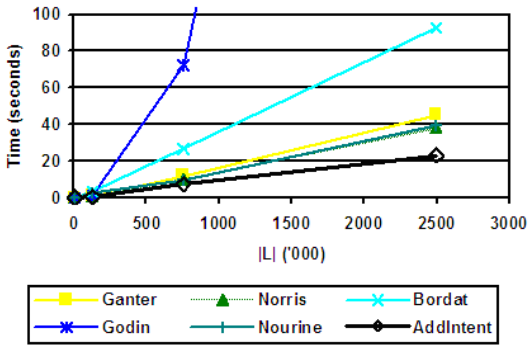

Fig. 6. Results for the Rnd-50 dataset: $|M| = 100$, $|g'| = 50$, $|G|$ varies from 10 to 40, the incremental step being 10 objects.

On random contexts with relatively high density (50%), AddIntent is still the best performer with Norris second and Nourine a close third.

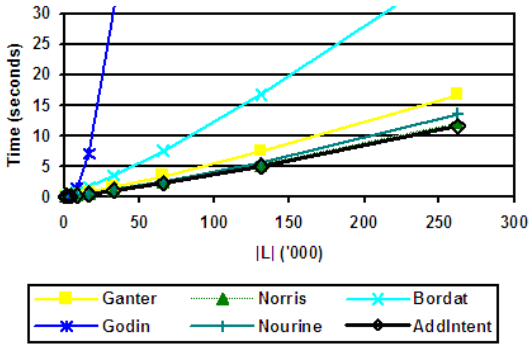

Fig. 7. Results for contexts of the form $(G, G, \neq)$, which give rise to Boolean lattices.

In artificial contexts that create Boolean lattices, AddIntent is again among the best algorithms and, in fact, the best of the six algorithms presented here with Norris a close runner-up. In fact, our results indicate that the fastest algorithm on such contexts is [15], but the performance of AddIntent is comparable to the performance of [15]. However, these contexts are primarily of theoretical interest, as they constitute the worst case with an exponential number of concepts and, thus, exemplify the worst-case complexity.

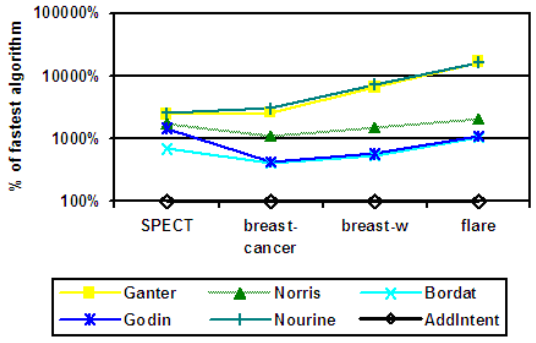

Fig. 8. Results for real datasets. The time spent by AddIntent is assumed to be 100%

Fig 8. shows the results of several tests performed using real datasets from the UCI repository [16]. SPECT (Single Proton Emission Computed Tomography) is a real dataset that contains 267 objects and 23 attributes, generating a lattice with 21550 concepts. The remaining datasets (Breast Cancer[3], Wisconsin Breast Cancer, and Solar Flare databases) are given in the form of many-valued tables and the QuDA program [17, 18] was used to scale them into one-valued contexts.

It is interesting to note that the performance gap between AddIntent and the nearest best performers is even bigger than with artificial contexts (note that a logarithmic scale is used in this particular chart). Furthermore, even algorithms building only the concept set were considerably slower than AddIntent on these datasets. Thus, on real databases, AddIntent seems to be the fastest algorithm among all available algorithms generating the concept set with- or without the diagram graph. Further research is however necessary to give more substance to this claim.

6 Related and further work

AddIntent was first implemented in 1996 in the context of so-called compressed pseudo-lattices [12, 19] and subsequently refined to the version given here. To the best of our knowledge, the most closely related lattice construction algorithm among those ever published is the recent one from [20] (designated "Algorithm 5" in the text). It uses a similar methodology based on Propositions 1 and 2 to avoid the consideration of old concepts and non-canonical generators in the lattice in a bottom-up search for canonical generators and modified concepts. This algorithm relies on a two-phased iterative approach to first discover canonical generators and in the second phase update the lattice. AddIntent combines these two phases. In addition, the algorithm in [20] uses a stack and *tries* as utility data structures facilitating lookup of concepts (as opposed to AddIntent's use of recursion and the diagram graph itself). Direct comparison of this algorithm and AddIntent (and its optimizations) has not yet been conducted and is a topic of further research. Since both algorithms follow the same main strategy, one can expect that their behavior is also similar, in particular, as compared to the behavior of other algorithms. A deeper comparison may reveal potential trade-offs between specific issues in which the algorithms differ and may suggest further implementation improvements. We would like to thank the reviewers of this paper for pointing out the resemblance between our approach and the one in [20].

7 Conclusions

In [8], it has been shown that the choice of an algorithm should be based on the properties of the context such as its size and density, which are indeed good

[3] This breast cancer domain was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data

predictors for randomly generated contexts. From the results presented above, one can see that various algorithms perform differently across various context densities, whereas the performance of AddIntent is always at least acceptable, if not the best. In cases when this algorithm is not the fastest one (for example, on sparse contexts where it is inferior to Bordat), its speed is comparable with that of the best performer. In other situations, it is often superior to other algorithms. Therefore, AddIntent is a good candidate for a universal algorithm generating the diagram graph.

Certainly, the construction of the diagram graph requires much computational effort; hence, algorithms generating only the concept set, are in general, faster than AddIntent. However, as already mentioned, the situation with real datasets is apparently different. In real data, unlike in randomly generated contexts, we can expect specific patterns, and a good algorithm should be able to take advantage of them. If further experiments confirm the efficiency of AddIntent over other algorithms on real datasets, this can be a step forward in lattice construction, since real datasets are the ones that matter after all.

A wider study is needed to understand how dependencies in real data can be exploited to get better performance. The proposed workshop on FCA algorithms [21] may provide the necessary infrastructure for this study.

References

1. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag, Berlin Heidelberg New York (1999)
2. Wille, R.: Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. In Rival, I., ed.: Ordered Sets. Reidel, Dordrecht–Boston (1982) 445–470
3. Bordat, J.: Calcul pratique du treillis de Galois d'une correspondance. Math. Sci. Hum. **96** (1986) 31–47
4. Ganter, B.: Two Basic Algorithms in Concept Analysis. FB4-Preprint No. 831, TH Darmstadt (1984)
5. Godin, R., Missaoui, R., Alaoui H.: Incremental Concept Formation Algorithms Based on Galois Lattices. Computation Intelligence **11** (1995) 246–267
6. Norris, E.: An Algorithm for Computing the Maximal Rectangles in a Binary Relation. Revue Roumaine de Mathématiques Pures et Appliquées **23** (1978) 243–250
7. Valtchev, P., Missaoui, R.: Building Concept (Galois) Lattices from Parts: Generalizing the Incremental Methods. In: Lecture Notes in Artificial Intelligence. Volume 2120., Berlin Heidelberg New York, Springer-Verlag (2001) 290–303
8. Kuznetsov, S., Obiedkov, S.: Comparing Performance of Algorithms for Generating Concept Lattices. J. Experimental and Theoretical Artificial Intelligence **14** (2002) 189–216
9. Van Der Merwe, F., Kourie, D.: AddAtom: an Incremental Algorithm for Constructing Concept- and Concept Sublattices. Technical report of the Department of Computer Science, University of Pretoria, South Africa (2002)
10. Carpineto, C., Romano, G.: A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. Machine Learning **24** (1996) 95–122

11. Nourine, L., Raynaud, O.: A fast algorithm for building lattices. Information Processing Letters **71** (1999) 199–204
12. Van Der Merwe, F.: Constructing Concept Lattices and Compressed Pseudo-Lattices. M.Sc. dissertation, University of Pretoria, South Africa (2003)
13. Obiedkov, S.: Algorithms and Methods of Lattice Theory and Their Application in Machine Learning. PhD thesis, Russian State University for the Humanities (2003)
14. Ferré, S.: Incremental Concept Formation Made More Efficient by the Use of Associative Concepts. INRIA Research Report no 4569 (2002)
15. Valtchev, P., Missaoui, R., Lebrun, P.: A Partition-Based Approach towards Building Galois (Concept) Lattices. Rapport de recherche no. 2000-08, Département d'Informatique, UQAM, Montréal, Canada (2000)
16. Blake, C., Merz, C.: UCI Repository of Machine Learning Databases. [http://www.ics.uci.edu/˜mlearn/MLRepository.html], Irvine, CA: University of California, Department of Information and Computer Science (1998)
17. Grigoriev, P., Yevtushenko, S.: Elements of an Agile Discovery Environment. In: Lecture Notes in Artificial Intelligence. Volume 2843., Berlin Heidelberg New York, Springer-Verlag (2003) 309–316
18. Grigoriev, P., Yevtushenko, S., Grieser, G.: QuDA, a Data Miner's Discovery Environment. Technical Report AIDA–03–06, TU Darmstadt [http://www.intellektik.informatik.tu-darmstadt.de/˜peter/QuDA.pdf] (2003)
19. Van Der Merwe, F., Kourie, D.: Compressed Pseudo-Lattices. J. Expt. Theor. Artif. Intell. **14** (2002) 229–254
20. Valtchev, P., Missaoui, R., Godin, R., Meridji, M.: Generating Frequent Itemsets Incrementally: Two Novel Approaches Based on Galois Lattice Theory. J. Expt. Theor. Artif. Intell. **14** (2002) 115–142
21. Mailing list on FCA-algorithms:
http://www.aifb.uni-karlsruhe.de/mailman/listinfo/fca-algo.
Internet site of the Workshop on Algorithms of Formal Concept Analysis:
http://kvo.itee.uq.edu.au/twiki/bin/view/Tockit/AlgoWorkshop.