DFSP: A new algorithm for a swift computation of formal concept set stability

Ilyes DIMASSI, Amira MOUAKHER and Sadok BEN YAHIA

University of Tunis El Manar, LIPAH, Faculty of Sciences of Tunis, Tunis, Tunisia.
sadok.benyahia@fst.rnu.tn

Abstract. Concept lattices are very useful for the task of knowledge discovery in databases. However, the overwhelming number of drawn formal concepts was always an actual hamper towards their effective use. In the aim of filtering out, such endless lists of formal concepts, the stability metric is the most worth of mention one. In this respect, the stability computation of large concepts has been shown to be infeasible due to exponential number of object sets to be processed. The literature only witnesses approaches for the stability computation that heavily rely on the existence of the Galois lattice. In this paper, we introduce a new efficient algorithm, called DFSP, for computing the stability of a set of formal concepts without having at hand the underlying partial relation. The main thrust of the introduced algorithm stands in the smart detection of non generators and their pruning owe to their fulfilment of monotony property within a given equivalence class. To the best of our knowledge, DFSP is the first algorithm that tackled such tough issue. Carried out experiments showed that DFSP efficiently computes the scalability of very large formal concepts extracted from benchmark datasets of the Data Mining field.
Keywords: Formal concept analysis, stability, generators,pruning, tidset.

1 Introduction and motivations

Concept lattices are very useful for the task of knowledge discovery in databases. However, this field is hampered by the overwhelming size of formal concept lists drawn from even moderately sized contexts. In this respect, the stability index has been shown to be efficient for throwing away "bad" concepts. Nevertheless, the computation of such stability index is very consuming and has been shown to be NP-Complete task. Thus, the FCA community paid much attention to the computation of exact and/or approximative of the stability as could witness the recent publications on such issue [1, 2]. At a glance, the recent proposals of approximations of stability computations unveil the actual complexity of such a task. Roughly speaking, the computation of the stability of a concept $C = (A, B)$ comes back to the exploration to a huge space made of the power set of the extension part. In this space, we have to record all the elements in snugness connection with the corresponding part. Clearly, even for an extent with dozens of objects, it does not exit a primitive type for storing the value of a stability[1].

At a glance, the related work flags out approaches that compute the stability of a set of formal concepts organized through the Galois lattice. Doing so, they start computing

[1] The GMP library, https://gmplib.org/, could be of use, in such a case, to store huge values.

the stability of smallest formal concept (in extent's size terms) and exploit this result for the subsumer concepts until reaching the top formal concept.

In this paper, we introduce a new algorithm, called DFSP, that aims to an efficient straightforward computation of a set of formal concepts. The main thrust of the introduced algorithm stands in the smart detection of non generators and their pruning owe to their fulfilment of monotony property within a given equivalence class. Indeed, we introduce the notion of saturation of non-generators through the detection of the maximal set of a non-generators. Given that each subset of a non generator is also a non generator, the DFSP algorithms sweeps the search space in depth first manner and only stresses on the generators by avoid squandering its efforts on useless non-generators subspaces.

The carried out experiments highlight that DFSP easily handles formal concepts having thousands of objects in their extent part. To the best of our knowledge, DFSP is the first algorithm that handles efficiently and straightforwardly formal concepts for the stability computation.

The remainder of the paper is organized as follows: The next section recalls key notions used throughout this paper as well as the pioneering approaches of the related work. Then, we present in section 3 our algorithm for computing the stability of a set of formal concepts, called DFSP. Section 4 describes the experimental study and the results we obtained. Section 5 concludes the paper and points out our future work.

2 Stability computation: Scrutiny of the Related work

Before scrutinizing the related work that paid attention the stability computation, we provide a simplified definition of some concepts used throughout in this paper, by supposing that the the reader is familiar with FCA basic settings.

Definition 1. (MONOTONIC / ANTI-MONOTONIC CONSTRAINT) *Let Q be a constraint,*

- *Q is anti-monotonic if $\forall I \subseteq \mathcal{I}$, $\forall I_1 \subseteq I$: I fulfils $Q \Rightarrow I_1$ fulfils Q.*
- *Q is monotonic if $\forall\ I \subseteq \mathcal{I}$, $\forall\ I_1 \supseteq I$: I fulfils $Q \Rightarrow I_1$ fulfils Q.*

Definition 2. (EQUIVALENCE CLASS) *An equivalence class is a set of itemsets with same closure (and same image). Let C=(A, B) be a formal concept, for any subset $X \subseteq \mathcal{O}$, $A = X''$ is the largest tidset w.r.t. set inclusion in its equivalence class. Precisely, $A \subseteq \mathcal{O}$ is closed iff $\nexists X$ such as $X \subset A$ with $X' = A'$; $X \subseteq \mathcal{O}$ is a generator iff $\nexists U \subset X$ with $U' = X'$. $G_A = \{X \subseteq A | X' = B\}$ is the set of all generators in the equivalence class.*

Definition 3. (EXTENT FULL SPACE) *Let $\mathcal{K}$ = ($\mathcal{O}$, $\mathcal{I}$, $\mathcal{R}$) be a formal context, $\mathcal{B}(\mathcal{K})$ its concept lattice and $C = (A, B)$ a concept from $\mathcal{B}(\mathcal{K})$ where $|A| = n$. $\mathcal{P}(A) = \{X | X \subseteq A\}$ is the set of all A's subsets and $\mathcal{P}(A) = 2^{|A|}$.*

Stability has been introduced probably for the first time by Kuznetsov [3] and later revisited in [4, 5]. This measure seems to be the most widely used around the FCA community and is applied in numerous applications [6], e.g. biclustering, detection of scientific subcommunities, to cite but a few. Formally, it is defined as follows:

Definition 4. (STABILITY) *The stability index of a given concept describes the proportion of subsets of its objects whose closure is equal to the intent of this concept. This metric reflects the dependency of the intent on particular objects of the extent [4]. Let $\mathcal{K} = (\mathcal{O}, \mathcal{I}, \mathcal{R})$ be a formal context and $C = (A, B)$ a formal concept of $\mathcal{K}$. The stability index, σ, of C is defined as follows:*

$$\sigma(A, B) = \frac{|\{C \subseteq A | C' = B\}|}{2^{|A|}} = \frac{|G_A|}{2^{|A|}} \tag{1}$$

The authors of [7] highlighted that a concept that covers fewer objects is normally less stable than do a concept covering more objects.

However, the main hamper towards its intensive use mainly for large datasets, is the complexity of its computation. In fact, it's shown to be a #P-complete problem [3, 5]. In order to compute it for large concept lattices, several works proposed to use estimates and approximations however others tried to find an exact solution using the concept lattice in computing stability.

Roth et al. [8] paid attention to concept's stability as well as other metrics to reduce the size of large concept lattice. They proposed an exact and polynomial algorithm COMPUTESTABILITY that computes the stability indices for every concept of the lattice using the covering graph of a concept lattice. The algorithm traverses the covering graph from the bottom concept upwards. A concept is processed only after the stability indices of all its sub-concepts have been computed. The main drawback of this algorithm that it is essentially quadratic in the number of concepts in the lattice, which may be prohibitively expensive for large lattices. In addition, this algorithm inputs a Galois lattice and such a requirement could not be easily available for very large datasets may be impractical.

Kuznetsov [5] introduced a polynomial algorithm for computing stability using various methods of estimating scientific hypotheses. This algorithm is considered as optimal in the sense that its time complexity is linear and polynomial in the size of the context. Nevertheless, this approach only gives an approximate assessment about the stability index and could not be efficient in exact studies.

Later, Jay et al. [9] applied the concept of stability and iceberg lattices in social network analysis. They used the stability metric to reduce the complexity of the lattice, by filtering out all unstable concepts w.r.t. a given threshold. In this respect, the authors introduced a new definition of the stability using the equivalence classes. Given a concept (A, B), the stability metric measures the number of elements of G that are in the same equivalence class of A where an equivalence class is defined as follows:

Property 1. Using Definition 4, the authors proved the following property:

$$\sigma(A, B) = 1 - \sum_{X \subset A, X = X''} \sigma(X, X') 2^{|X| - |A|}$$

So, once the lattice concept is given, it is possible to compute quickly the stability of concepts using an ascending algorithm.

Roth et al. [10] proposed an algorithm, based on a polynomial heuristic for computing stability index for all concepts using the concept lattice. This algorithm was quite

good in practical applications so far, but in the worst case its complexity is quadratic in the size of the lattice.

Later, Babin and Kuznetsov [1] also suggested a method for approximating concept stability based on a Monte Carlo approach. Their approximate algorithm can run in reasonable time. In their approach, they specified a new parameter called *stability threshold* to reduce the number of the concepts. The results show that the approximations are better when stability threshold is low.

Recently, Buzmakov et al. [2] introduced an efficient way for finding a "good" assessment of concept stability. The authors combined the *bounding method* [9] as well as the *Monte Carlo method* [1] in a complementary way. Once the stability bounds are computed in the lattice, the method that should be applied is chosen according to the most tight of each them.

The main criticism that can be made about the literature's approaches stands in the fact that they are unable to compute stability of concepts in absence of order relation. In fact, the lattice structure is a *sine qua non* condition to proceed with the computation. Beside that, such computation of concept's stability requires visiting all its sub-concepts, i.e., direct and non direct ones. Clearly, doing so is very greedy in computations and memory usage. Nevertheless, building concept lattice is very far from being an easy task [11] and sometimes impossible. Furthermore, the cost of generating a lattice concept remains high as far as the context is composed of a large number of objects [12] and/or a dense incidence relation.

Thus, we introduce a new efficient algorithm, called DFSP that allows computing the stability for a given set of concepts. The latter doesn't need any partial relation between the concepts. The main thrust of the introduced algorithm stands in the smart detection of non generators and their pruning owe to their fulfilment of monotony property within a given equivalence class.

3 DFSP : Depth First Stability Processor algorithm

Before, thoroughly describing the DFSP algorithm, we start by introducing some useful notations that are used in the remainder.

Let $C = (A, B)$ be a formal concept for which the stability index needs to be processed. DFSP algorithm organises $\mathcal{P}(A)$ according to a tidset prefix tree. Each exploration node in the tree is specific to a tidset and represented by the TSNode data structure. TSNode is a recursive structure that keeps track of useful informations about its associated tidset such as its suffix, itemset and a set of immediate supersets. As for the current tidset, its immediate supersets are themselves represented each by a TSNode instance and so on.

Definition 5. (SUFFIX OF TIDSET) *Let $ts = \{t_1, t_2 \dots t_k\}$ be an ordered sequence of objects and n_{ts} its associated exploration node. $Suff(ts) = t_k$ is the last object in ts. TSNode.s is the member property of the data structure TSNode in which $Suff(ts)$ is maintained ; $n_{ts}.s = t_k$.*

	a	b	c	d	e	f	g	h
1		×					×	
2	×		×				×	
3				×	×	×	×	×
4	×			×	×	×	×	×
5	×				×	×	×	×
6		×				×	×	
7	×					×	×	
8	×				×		×	
9	×	×	×	×	×	×	×	×

Table 1. Formal context

Definition 6. (CHILDREN'S NODE) *TSNode.ss is a member of the TSNode structure that holds a set of TSNode instances. TSNode.ss is the set of a TSNode immediate children.*

Definition 7. (NODE INTENT) *Let ts be a tidset and n_{ts} its associated node. TSNode.is is a TSNode member that holds the tidset image. $n_{ts}.is = ts'$.*

Unlike $n_{ts}.s$ which only holds the tidset suffix $Suff(ts)$, $n_{ts}.is$ integrally maintains ts'.

Example 1. *With respect to the formal context depicted in Table 1, we have $ts = \{123\}$ and n_{123} its associated node, then $n_{123}.s = 3$. In addition, we have : $n_{123}.ss$=$\{n_{1234}$, n_{1235}, n_{1236}, n_{1237}, n_{1238}, $n_{1239}\}$. Besides, we also have $n_{123}.is = ts' = \{123\}' = \{g\}$.*

In the following, we present a detailed description of DFSP algorithm. Let us remind that the main idea of our new approach is to provide a simple and very efficient strategy for computing stability through generators enumeration. The DFSP algorithm, whose pseudo-code is sketched by Algorithm 1, operates mainly as follows:

Initially, the sizes of the extent and the intent are stored respectively into n and m (lines 2,3). The root node is then built and it's $root.s$ and $root.is$ members are set to $\emptyset$ (line 4). Then, the objects a_i of A are scanned in order to build the first level nodes. For each object a_i, a child node is created, its member $child.s$ is set to a_i and its member $child.is$ is set to the intent of a_i (line 6, 7). If the size of the intent a'_i is different from m the size of B, then child is a non generator and is added to $root.ss$ the set of the root node immediate children (c.f lines 8, 9). Then the first level generators count is determined using the generators counting formula (line 10). After that, the exploration of space of tidset through the EXPLORETIDSET function updates gc one last time to obtain the final count of generators (line 11). The stability index is determined when the generators count gc is divided by the overall tidset count (line 12).

In the following part, we will explain the fundamental step of the algorithm which is the recursive exploration of tidset's space.

Algorithm 1: Depth First Stability Processor (DFSP)

Data: **TSNode**
-**TSNode.s**: the tidset suffix
-**TSNode.is**: the tidset intent
-**TSNode.ss**: the set of immediate children nodes.
Input:
-$\mathcal{K} = (\mathcal{O}, \mathcal{I}, \mathcal{R})$: a formal context.
-$C = (A, B)$: a formal concept.
Results:
-$\mathcal{S}$: the stability of C.

```
1  Begin
2    n := |A|;
3    m := |B|;
4    root.i := root.is := ∅;
5    For i = 1 ... n do
6      nchild.s := a_i;
7      nchild.is := a'_i;
8      If |nchild.is|! = m then
9        root.ss∪ = nchild;
10   gc := Σ_{i=|root.ss|}^{n-1} 2^i;
11   gc := gc + EXPLORETIDSET(root.ss, K, m);
12   S := gc / 2^n;
13   Return S;
14 End
```

3.1 Depth First exploration of the Tidset space

The main goal of EXPLORETIDSET which pseudo-code is sketched through Algorithm 2 is counting generators while minimizing as much as possible the visited tidsets. This is achieved by pruning generators and most importantly by detecting "prunable" non generators. The first invocation for EXPLORETIDSET (c.f line 11 of DFSP) is applied on the root node immediate children.

The tidset space exploration pattern The exploration mechanics are straightforward. To harness this process, lets ignore any possible optimisation that leads to nodes pruning. On the first invocation of EXPLORETIDSET in DFSP, ss contains the nodes $\{n_{a1}, \ldots, n_{an}\}$ associated to unitary first level tidsets $\{a1\},\ldots,\{an\}$ for which EXPLORETIDSET builds immediate children as follows: n_{a1a2} the first immediate child of n_{a1} is built by adding the suffix of n_{a2} to n_{a1}. More generally, n_{aiaj} the j^{th} immediate child of n_{ai} (line 12) is obtained by adding to n_{ai} the suffix of $n_{a(i+j)}$ the j^{th} node following n_{ai} (line 15). For the tidset $\{aiaj\}$ associated to n_{aiaj}, only $Suff(\{aiaj\})$ is stored in $n_{aiaj}.s$ (line 16). Since, $\{aiaj\} = \{ai\}|Suff(\{aj\})$ then $\{aiaj\}' = \{ai\}' \bigcap Suff(\{aj\})'$. $\{aiaj\}'$ is stored integrally in $n_{aiaj}.is$ (line 17). After building $n_{ai}.ss$ from n_{ai} followings, EXPLORETIDSET is recursively applied on $n_{ai}.ss$ (line 22) which only makes sense when $|n_{ai}.ss|$ has at least 2 elements (line 21). After process-

ing the n_{ai} subspace, EXPLORETIDSET moves to the next node $n_{a(i+1)}$ (line 12). The last node is n_{an} is not processed as it has no followings.

Counting and pruning generators As described above, the generation process builds a child node by adding an object to its parent node. A child node is therefore always a superset of its parent. Otherwise, it is known that a superset of a generator is also a generator. Therefore, applying the exploration process on a generator node will inevitably produce generators. The exploration branch starting from that node is said to be monotonous and since we are able to count the population induced from that branch we can save processing power by dismissing these nodes (line 8 in DFSP and line 18 in EXPLORETIDSET). Let's find out how it is possible to count generators that are inferred from a given generator without the need of exploring them. Let n_i be the i^{th} node in ss and n_i is a generator. Building the n_i subspace by exploring its immediate children then its children's children and so on recursively is equivalent to generating all possible supersets of the tidset associated to n_i using the suffixes of n_i following nodes $\{n_{(i+1)}.s, \ldots, n_{(i+1)}.s\}$. The count of all generators in n_i branch (including even n_i) is equal $2^{(n-i-1)}$.

We have to also consider supersets of n_i that are not part of n_i branch but rather in n_k branches $\{k >= 1 \text{ and } k < i\}$ the branches of all nodes that precedes n_i in ss. To avoid locating these nodes and simplify calculations, let's virtually move n_i to the start of ss. All supersets of n_i are now confined in n_i branch and the updated count of n_i supersets is $2^{(n-1)}$. Let n_j be another generator in the same cluster. It is important to count all n_j supersets while avoid including elements that are already counted as part of the n_i branch. By virtually moving n_j after n_i in ss and counting all elements in the n_j branch, it is possible to fulfil both conditions. n_j branch count is $2^{(n-2)}$ and the same process is applied to the remaining generators in the cluster. Doing so leads us to the generalized generators counting formula $gc = \sum_{k=|ss|-1}^{|gs|} 2^k$ where $|gs|$ is the generators count and $|ss|$ is cluster size (line 10 in DFSP and line 20 in EXPLORETIDSET).

Detecting non generators monotony The most significant mop up mechanism in DFSP is non generators pruning. In order to also eliminate non generators, EXPLORETIDSET looks for nodes in ss that when combined together, the resulting clique superset will still be a non generator. Those nodes are said to form a *non generator monotonous clique*. Suppose, we're building the branch of a node from this clique. If we use exclusively nodes from the clique, all nodes in the branch are guaranteed to be subsets of the clique superset. Since a subset of a non generator is also a non generator then all branches in the clique will only contain non generators. Nodes in the clique are pushed at the end of the ss set to insure that the generation process will only use nodes from the clique. Nodes outside the clique are moved away to the beginning of ss. Nodes in the clique are not expanded, since no generator could be found in their branches but are still used to build branches outside the clique.

Algorithm 2: EXPLORETIDSET

Input:
-$\mathcal{K} = (\mathcal{O}, \mathcal{I}, \mathcal{R})$: a formal context.
-ss=a set of TSNode siblings.
-m: the size of the intent.
Results:
-gc : the generators count.

```
1  Begin
2    i := ssc := |ss|;
3    ingpc := gc := 0;
4    ingpi := I;
5    While i < 1 do
6      If |ngpi ∩ ss[i].is| = m then
7        MOVETOHEAD(i, ss);
8        ingpc := ingpc + 1;
9      Else
10       i := i − 1;
11       ngpi := ngpi ∩ ss[i].is;
12   For (i = 1 ... ingpc) do
13     nleft := ss[i];
14     For (j = i + 1 ... ssc) do
15       nright := ss[j];
16       nchild.s := nright.s;
17       nchild.is := nleft.is ∩ f(nchild.s);
18       If (|nchild.is|! = m) then
19         nleft.ss∪ := nchild;
20     gc+ = Σ_{k=|nleft.ss|}^{ssc−i−1} 2^k;
21     If (|nleft.ss| > 1) then
22       gc+ = EXPLORETIDSET(nleft.ss, K, m);
23   Return gc;
24 End
```

3.2 Illustrative example

To illustrate our approach, let us consider the formal concept $C_1 = (A_1, B_1)$ from the formal context depicted by Table 1 such that $A_1 = \{3, 4, 5, 6, 7, 9\}$ and $B_1 = \{f, g\}$. As shown in figure 1, the DFSP algorithm operates as follows:

During the first step (1), the root node is created and initialized though the function BUILDTREEROOT (gc=0). Initially, $root.s = \emptyset$ and nodes n_3, n_4, n_5, n_6, n_7 will be created through individual elements of $\{3, 4, 5, 6, 7, 9\}$ (steps (2), (3), (4), (5), (6) and (7)). These nodes are prospective direct children to the root node. Given that all these nodes are non generators, they become in step (8) as effective direct children of root and are decreasingly sorted with respect to their support value. In step (9), non generators forming monotone clique are placed at the end of the list and marked by (*). However,

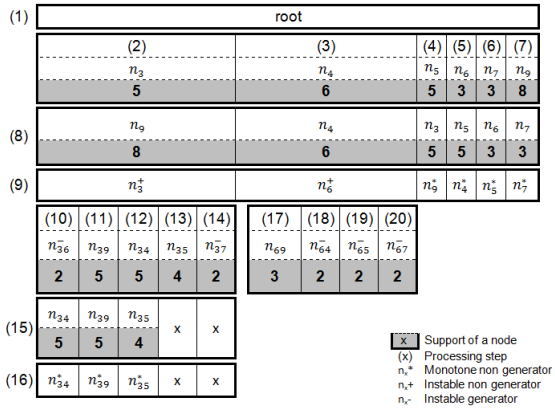

Fig. 1. Illustrative example

instable generators are placed at the beginning of the list and marked by (+). After that, in steps (10), (11), (12), (13) and (14) prospective direct children of node n_3 are created which are, respectively, $n_{36}, n_{39}, n_{34}, n_{35}$ and n_{37}. The count of generators is updated in step (15) ($gc = 2^4+2^3 = 24$). Only nodes n_{34}, n_{35} and n_{39} are left as effective direct children of n_3. The latter are also sorted decreasingly. In step (16), all these effective direct children form a monotone clique and exploration of this branch is stopped. After that, nodes n_{69}, n_{64}, n_{65} and n_{67} are created and count of generators is also updated in step (21) with the tree generators of n_{64}, n_{65} and n_{67} ($gc = gc + 2^3 + 2^2 + 2^1 = 24 + 14 = 38$). Only the node n_{69} is kept in the list of effective direct children of n_6. Indeed, the latter does not fulfil the condition of EXPLORETIDSET to be launched.

4 Experimental results

In this section, we put the focus on the evaluation of the DFSP algorithm by stressing on two complementary aspects : (*i*)Execution time; (*ii*) efficiency of search space pruning. Experiments were carried out on an Intel Xeon PC, CPU E5-2630 2,30 GHz with 16 GB of RAM and Linux system. During the lead experiments, we used some benchmark datasets commonly of extensive use within Data mining. The first three datasets are considered as dense ones, *i.e.*, yielding high number of formal concepts even for a small number of objects and attributes, while the other ones are considered as sparse. The characteristics of these datasets are summarized by Table 2. Thus, for each dataset we report its number of objects, the number of attributes, as well as the number of all formal concepts that may drawn. In addition, we also reported the respective sizes of the smallest and the largest formal concepts (in terms of extent's size). For these considered concepts, we kept track of the number of the actually explored nodes as well as the execution time (the column denoted $|explor.|$).

At a glance, statistics show that the DFSP algorithm is able to process dozens of thousands of objects in a reasonable time. Indeed, the 15596 (respec. 16040) objects

composing the extent of the largest formal concept extracted from the RETAIL (respec. T10I4D100K) dataset are handled in only 27.27 (respec. 68.85) seconds. Even though, the respective cardinalities are close (15596 vs 16040 objects), the difference in execution time is not proportional to this low gap. A preliminary explanation could be the difference in density of both datasets (RETAIL is dense while T10I4D100K is a sparse one). A in-depth study of these performances in connection to the nature of datasets is currently carried out. The most sighting fact is the low number of visited nodes in the associated search space. For example for the MUSHROOM dataset, DFSP algorithm actually handled only 83918 nodes from 2^{1000} potential nodes of the search space, i.e., in numerical terms it comes to only explore infinitely insignificant part equal to 7.8×10^{-297} of the search space. The case of RETAIL and T10I4D100K datasets is also worth of mention. For the respective smallest extracted concepts, DFSP algorithm only explores, $1.14 \times 10^{(-45)}$ and $1.5 \times 10^{(-90)}$ parts of the respective search spaces.

Datasets	**# Attr**	**# Obj**	**# concepts**	**smallest concept**			**largest concept**		
				$\|ext\|$	$\|explor.\|$	**time (sec.)**	$\|ext\|$	$\|explor.\|$	**time (sec.)**
CHESS	75	3196	3316	2630	2362233	0.12	3195	5855899	0.64
MUSHROOM	119	8124	3337	1000	83918	0.10	8124	76749955	11.32
RETAIL	16470	88162	3493	150	164	0.10	15596	64847191	27.27
T10I4D100K	1000	100000	4497	300	306	0.11	6810	19719991	12.77
T40I10D100K	1000	100000	3102	1800	1495324	1.39	16040	92154598	68.85

Table 2. Characteristics of the considered benchmark datasets

These highlights are also confirmed by Figures 2-11. Indeed, Figures 2, 4, 6, 8 and 10 stress on the variation of the Execution time, while Figures 3, 5, 7, 9 and 11 assess what we call the workload which means the efficiency of search space exploration. At a glance, the execution time is in a snugness connection with the reduction of search space, i.e., the variation of the workload has the same tendency as the performance since we consider the visited tidset in the search space as the processing unit. Worth of mention, the performance is rather correlated to the extent's size rather than the exponential nature of the search space.

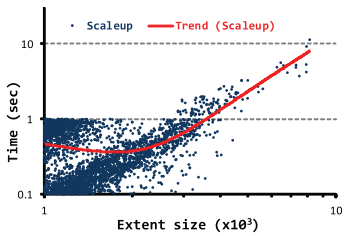

Fig. 2. Mushroom scaleup

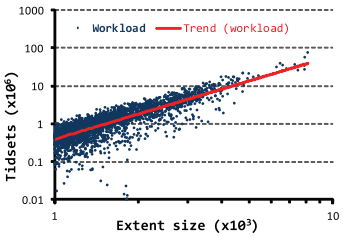

Fig. 3. Mushroom workload

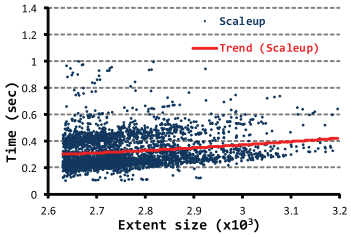

Fig. 4. Chess scaleup

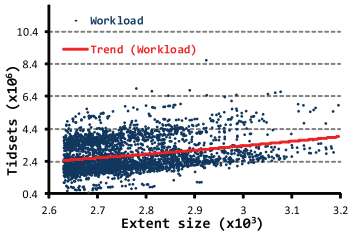

Fig. 5. Chess workload

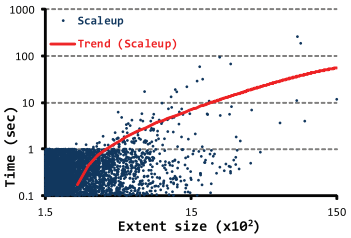

Fig. 6. Retail scaleup

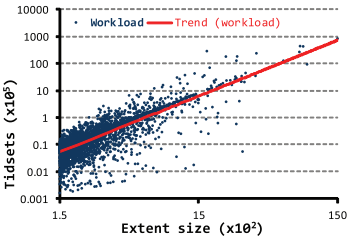

Fig. 7. Retail workload

5 Conclusion and future work

Through the DFSP algorithm, we gaped in the combinatorics of lattices by the showing that most of this sear space could be smartly explored thanks to the saturation of generators. The swift computation of stability encouraged us to integrate the stability as a on-the-fly pruning strategy during mining closed itemsets. We are currently working on a new algorithm for the stability computation given the Galois lattice. The new algorithm only relies on the direct sub-concepts to compute the stability of a concept. Outside the FCA field, the strategy of DFSP would be of benefit for very efficient extraction well known problem of combinatorics : minimal transversals.

References

1. Babin, M.A., Kuznetsov, S.O.: Approximating concept stability. In: Proceedings of the 11th International Conference on Formal Concept Analysis(ICFCA), Dresden, Germany. (2012) 7–15
2. Buzmakov, A., Kuznetsov, S.O., Napoli, A.: Scalable estimates of concept stability. In: Proceedings of the 12th International Conference on Formal Concept Analysis(ICFCA), Cluj-Napoca, Romania. (2014) 157–172

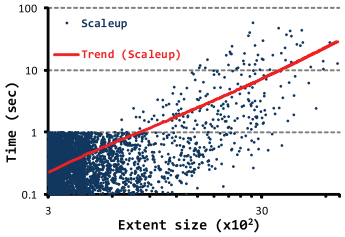

Fig. 8. T10I4D100K scaleup

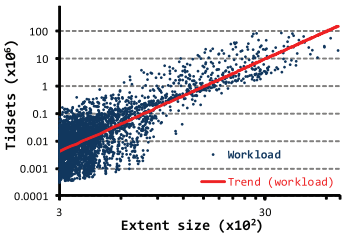

Fig. 9. T10I4D100K workload

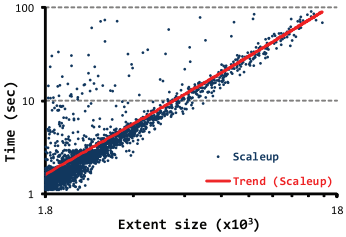

Fig. 10. T40I10D100K scaleup

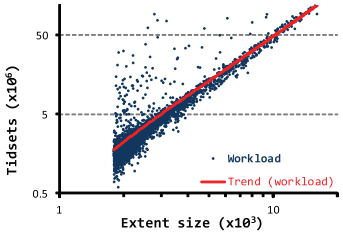

Fig. 11. T40I10D100K workload

3. Kuznetsov, S.O.: Stability as an estimate of the degree of substantiation of hypotheses derived on the basis of operational similarity. Automatic Documentation and Mathematical Linguistics **24** (1990) 62–75
4. Kuznetsov, S.O., Obiedkov, S.A., Roth, C.: Reducing the representation complexity of lattice-based taxonomies. In: Proceedings of the 15th International Conference on Conceptual Structures (ICCS), Sheffield, UK. (2007) 241–254
5. Kuznetsov, S.O.: On stability of a formal concept. Ann. Math. Artif. Intell. **49** (2007) 101–115
6. Buzmakov, A., Kuznetsov, S.O., Napoli, A.: Is concept stability a measure for pattern selection? Procedia Computer Science **31** (2014) 918 – 927
7. Klimushkin, M., Obiedkov, S.A., Roth, C.: Approaches to the selection of relevant concepts in the case of noisy data. In: Proceedings of the 8th International Conference(ICFCA) , Agadir, Morocco. (2010) 255–266
8. Roth, C., Obiedkov, S.A., Kourie, D.G.: Towards concise representation for taxonomies of epistemic communities. In: Proceedings of the 4th International Conference on Concept Lattices and Their Applications (CLA), Hammamet, Tunisia. (2006) 240–255
9. Jay, N., Kohler, F., Napoli, A.: Analysis of social communities with iceberg and stability-based concept lattices. In: Proceedings of the 6th International Conference(ICFCA), Montreal, Canada. (2008) 258–272
10. Roth, C., Obiedkov, S.A., Kourie, D.G.: On succinct representation of knowledge community taxonomies with formal concept analysis. Int. J. Found. Comput. Sci. **19** (2008) 383–404
11. Qiao, S.Y., Wen, S.P., Chen, C.Y., Li, Z.G.: A fast algorithm for building concept lattice. (2003)
12. Demko, C., Bertet, K.: Generation algorithm of a concept lattice with limited object access. In: Proceedings of the 8th International Conference Concept Lattices and Their Applications (CLA), Nancy, France. (2011) 239–250