J Supercomput
https://doi.org/10.1007/s11227-018-2453-z

Minimal generators, an affordable approach by means of massive computation

F. Benito-Picazo[1] · P. Cordero[1] · M. Enciso[1] · A. Mora[1]

Abstract Closed sets and minimal generators are fundamental elements to build a complete knowledge representation in formal concept analysis. The enumeration of all the closed sets and their minimal generators from a set of rules or implications constitutes a complex problem, drawing an exponential cost. Even for small datasets, such representation can demand an exhaustive management of the information stored as attribute implications. In this work, we tackle this problem by merging two strategies. On the one hand, we design a pruning, strongly based on logic properties, to drastically reduce the search space of the method. On the other hand, we consider a parallelization of the problem leading to a massive computation by means of a map-reduce like paradigm. In this study we have characterized the type of search space reductions suitable for parallelization. Also, we have analyzed different situations to provide an orientation of the resources (number of cores) needed for both the parallel architecture and the size of the problem in the splitting stage to take advantage in the map stage.

This work is partially supported by Project TIN2017-89023-P of the Science and Innovation Ministry of Spain, co-funded by the EU Regional Development (ERDF).

✉ A. Mora
amora@ctima.uma.es

F. Benito-Picazo
fbenito@lcc.uma.es

P. Cordero
pcordero@uma.es

M. Enciso
enciso@lcc.uma.es

[1] Universidad de Málaga, Málaga, Spain

Published online: 04 June 2018

Keywords Minimal generators · Formal concept analysis · Parallel methods · Logic

1 Introduction

Knowledge representation and reasoning are the two main pillars of artificial intelligence (AI). The growing interest in AI arisen in several areas beyond computer science is firmly rooted in the recent techniques to extract knowledge from the data and to be efficiently managed. One outstanding framework, based on a strong theoretical basis and also providing executable methods, is the Formal Concept Analysis (FCA) introduced by Ganter and Wille [11]. In FCA, data are stored in a table, representing a binary relation between G (a set of objects) and M (a set of attributes). FCA is not an approximate approach since it pursuits to capture, manage and analyze the complete knowledge from the information. This feature has a direct impact on the cost of the methods developed to extract and manage the information.

In this framework, two alternative knowledge representations are used: concept lattice and implications. The second one can be viewed as if-then rules already introduced in other areas, dressed with different clothes. Thus, in relational databases [4] they are named Functional Dependencies and in FCA they are named Implications [11]. All these notions capture the same idea (a pretty intuitive one): when some premise occurs, then a conclusion holds.

Since the size of the concept lattice (in the worst case) is $2^{\min(|G|,|M|)}$, the computational cost of the methods to build it has been considered as a limitation for the application of FCA. Algorithms to infer the concept lattice from the dataset were deeply studied and compared in [15]. Moreover, the extraction of the complete set of implications (basis) from a data set also presents an exponential behavior [5]. In both problems, the density of the dataset plays a major role in the performance of the execution of the method. To improve the computational behavior, in [14] the authors introduce the notion of redundant attributes to avoid the inclusion of such attributes in the extraction of the implications. Recently, in [8] a wide range of parallel methods to solve this second problem has been presented. These works motivate the need to combine some kind of reduction in the search space and a parallel execution to solve these complex problems.

In the two problems already mentioned, the input is the dataset and the output is its knowledge in terms of implications (basis) or attribute closed sets (concept lattice). However, here we deal with a complementary problem that allows to connect both knowledge representations: the enumeration of all the closed sets from a given set of implications. Moreover, we propose a method to produce not only all closed sets but also, for each of them, their canonical representations, named minimal generators.

Minimal generators are interesting not only from a theoretical point of view. In recent works, their computation is the core for solving other problems. The significance of minimal generators is well emphasized in the survey of Poelmans et.al. [21]: "Minimal generators have some nice properties: they have relatively small size and they form an order ideal". Qu et al. [22] pointed out that decision implications involve to know all minimal generators.

Furthermore, they have been used as a key point to build basis, which constitutes a compact representation of the knowledge allowing a better performance of the reasoning methods based on rules. Missaoui et al. [17,18] present the use of minimal generators to compute basis involving positive and negative attributes whose premises are minimal generators. All the mentioned authors have considered the dataset as the input of the problem, namely, the minimal generators and closed sets are inferred from the plain data. In [10,20] some methods for solving this problem have been proposed.

In this work, we consider implications as elements to describe the information and we design a method to enumerate all closed sets and their minimal generators from this information, and not from the dataset. This is a complementary problem to the extraction of minimal generators from the dataset and, as far as we know, no previous work has been developed. In summary, we propose a method to transform the knowledge in terms of implications, into a more organized form so that a complete representation of the closed sets of attributes (and their minimal generators) is obtained. This complete and precise specification allows a further fast management of the semantics of the information contained in the dataset.

Our starting point is [6], where a logic-based method based on SL_{FD}, a sound and complete logic for implications was introduced. In that work, we present the MinGen algorithm which works by traversing the set of implications and applying a set of inference rules, building a search tree space. This shape of the search space limits its execution for medium-sized problems, because of the overwhelming resources of the sequential MinGen algorithm. Although the search space is suitable for a parallel extension of the method, some further research has to be carried out. On the one hand, we first consider the integration of some strategies to reduce the search space. In this work, we present two approaches, MinGenRd and GenMinGen, providing a single-level or a multi-level strategy, respectively. We also show how the MinGenRd is suitable for parallelization whereas GenMinGen is not. On the other hand, we approach a parallelization of the MinGenRd, rendering the MinGenRdPar algorithm. This new method is driven by a like-Map-Reduce strategy.

In addition to the MinGenRdPar method, we also provide a set of experiments in different situations to characterize the resources needed for the parallel execution and to establish a threshold in the splitting stage. In both issues, our goal is to look for an efficient execution of the method. We remark that we have used a random generation of synthetic problems and also real-world problems. Our intention is to show that our approach is valid in both situations.

The paper is organized as follows: after presenting the needed background in Sect. 2, we revisit in Sect. 3 our starting point: a previous logic-based method to enumerate all minimal generators and closed sets (`MinGen`). Then the reduction strategy by eliminating redundant branches is described in Sect. 4, introducing the MinGenRd and GenMinGen methods. Later, in Sect. 5, we design a parallel version of the MinGenRd algorithm, named MinGenRdPar, running on a massive computation architecture. We provide some stats of its performance together with an illustrative execution of the parallel algorithm over a real (and well-known) problem. We will end up with Conclusions and Future Works. Figure 1 illustrates the content of the paper.

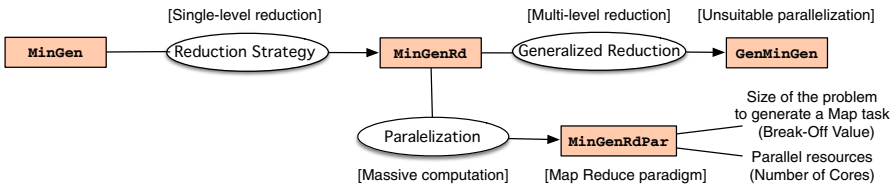

Fig. 1 Illustration of the content of the work

2 Preliminaries

Simplification Logic [7] is introduced by describing its four pillars: its *language*, its *semantics*, its *axiomatic system* and its *automated reasoning method*. Let M be a finite set of symbols (called *attributes*). The language over M is defined as

$$\mathcal{L}_M := \{A \to B \mid A, B \subseteq M\}$$

Formulas $A \to B$ are called (attribute) *implications* and the sets A and B are called *premise* and *conclusion* of the implication, respectively. Sets of implications are called *implicational systems*. In order to simplify the notation we omit the brackets in premises and conclusions and write their elements by juxtaposition. Thus, for instance, $ab \to cde$ denotes $\{a, b\} \to \{c, d, e\}$.

For introducing the semantics we use the notion of *closure operator*. For a more detailed description of closure operator see [23]. A closure operator on M is a mapping $c\colon 2^M \to 2^M$ that is *extensive*, *isotone* and *idempotent*. The fix-points for c are called *closed sets*. Closure operators are strongly connected to knowledge representation in different areas [1,3,9,12,16].

Since we use the logic as an executable tool, we have to substitute the semantic interpretation and inference for an efficient symbolic management. Thus, we introduce an axiomatic system considering reflexivity as axiom scheme

$$\texttt{[Ref]} \quad \frac{}{A \cup B \to A}$$

together with the following inference rules, called *fragmentation*, *composition* and *simplification*, respectively:

$$\texttt{[Frag]} \ \frac{A \to B \cup C}{A \to B} \qquad \texttt{[Comp]} \ \frac{A \to B,\ C \to D}{A \cup C \to B \cup D} \qquad \texttt{[Simp]} \ \frac{A \to B,\ C \to D}{A \cup (C \setminus B) \to D}$$

This axiomatic system is *sound* and *complete* (both semantic and syntactic derivations coincide). This result allows us to design automated reasoning methods. These methods can be approached by using a new closure operator called *syntactic closure*: give an implicational system $\Sigma \subseteq \mathcal{L}_M$, a set $X \subseteq M$ is said to be closed w.r.t Σ if $A \subseteq X$ implies $B \subseteq X$ for all $A \to B \in \Sigma$. Since M is closed w.r.t Σ and any intersection of closed sets is closed, we can define the following closure operator:

$$(\,)^+_\Sigma\colon 2^M \to 2^M \qquad X^+_\Sigma = \bigcap \{Y \subseteq M \mid Y \text{ is closed w.r.t } \Sigma \text{ and } X \subseteq Y\}$$

The automated reasoning method in Simplification Logic is based on the Deduction Theorem and three syntactic equivalences which allow us to transform the set of

implications with no semantic loss, i.e., the meaning of the knowledge representation is preserved.

Theorem 1 (Deduction Theorem) *Let* $A \rightarrow B \in \mathcal{L}_M$ *and* $\Sigma \subseteq \mathcal{L}_M$. *Then,*

$$\Sigma \vdash A \rightarrow B \quad \textit{iff} \quad B \subseteq A^+_\Sigma \quad \textit{iff} \quad \{\varnothing \rightarrow A\} \cup \Sigma \vdash \{\varnothing \rightarrow B\}$$

The following corollary characterizes the closed sets built with the syntactic closure as a maximum set enclosing the information of implications.

Corollary 1 *Let* $\Sigma \subseteq \mathcal{L}_M$. *For all* $X \subseteq M$, *one has* $X^+_\Sigma = \max\{Y \subseteq M \mid \Sigma \vdash X \rightarrow Y\}$.

In [19] we present a novel algorithm to compute closures using Simplification Logic. Given a set $A \subseteq M$, to compute A^+_Σ, the formula $\varnothing \rightarrow A$ is added to Σ and used as a seed by the reasoning method by using the equivalences provided by the previous proposition. Specifically, the algorithm uses the following equivalences:

- **Eq. I**: If $B \subseteq A$ then $\{\emptyset \rightarrow A, B \rightarrow C\} \equiv \{\emptyset \rightarrow A \cup C\}$.
- **Eq. II**: If $C \subseteq A$ then $\{\emptyset \rightarrow A, B \rightarrow C\} \equiv \{\emptyset \rightarrow A\}$.
- **Eq. III**: Otherwise $\{\emptyset \rightarrow A, B \rightarrow C\} \equiv \{\emptyset \rightarrow A, B \smallsetminus A \rightarrow C \smallsetminus A\}$.

One outstanding characteristic of our algorithm, named as Function `Cls`, is that, besides the attribute set corresponding to the closure, it also renders a set of implications corresponding to the complementary knowledge that describes the information which is not within the closure. This new set could be further treated in an iterative process, as our Minimal Generator algorithm will do. An illustrative execution of this function is shown in Example 1.

Example 1 Let $\Sigma = \{a \rightarrow c, bc \rightarrow d, c \rightarrow ae, d \rightarrow e\}$ and $A = \{c, e\}$. The algorithm `Cls` returns $A^+ = \{a, c, e\}$ and the following table summarizes its trace:

Guide	Σ
$\emptyset \rightarrow ce$	$a \rightarrow c$ $bc \rightarrow d$ $c \rightarrow ae$ $d \rightarrow e$
$\emptyset \rightarrow ce$	$a \rightarrow \not e$ $b\not c \rightarrow d\not c \rightarrow a\not e$ $d \rightarrow \not e$
$\emptyset \rightarrow ace$	$b \rightarrow d$

Therefore, $\texttt{Cls}(\{c, e\}, \{a \rightarrow c, bc \rightarrow d, c \rightarrow ae, d \rightarrow e\}) = (\{a, c, e\}, \{b \rightarrow d\})$.

An implicational system is said to be *complete for a closure operator* if it captures all the knowledge relating to that operator, that is to say, the implicational system expresses in the language of logic all knowledge relative to it. Therefore, when an implicational system Σ is complete for a closure operator c, its syntactic closure coincides with c, i.e., $X^+_\Sigma = c(X)$ for all $X \subseteq M$.

To efficiently manage all this knowledge, we characterize it with all the closed sets and their minimal generators. That is, given a set of implications Σ, closed sets characterize attribute sets with maximal meaning and, looking for a better integration in applications, in addition to the closed sets it would be very appreciated to provide a

compact representation of these closed sets. In other words, enumerate for each closed set those subsets that generate them. The following definition comes to formalize these kind of subsets.

Definition 1 Let Σ be a set of implications, $()^+_\Sigma \colon 2^M \to 2^M$ be its closure operator and $C \subseteq M$ a closed set, i.e., $(C)^+_\Sigma = C$. The set $A \subseteq M$ is said to be a minimal generator (mingen) for C if $(A)^+_\Sigma = C$ and, for all $X \subseteq A$, if $(X)^+_\Sigma = C$ then $X = A$.

As we mentioned before, the generation of the implicational system associated with a closure operator is a hard problem and the reverse one has also an exponential behavior. Thus, here we deal with the definition of a method to solve this problem (Sect. 3) and also with the design of its efficient implementation (Sect. 5), particularly approaching the problem by using parallelism.

3 Minimal generators method

In [6], the Simplification Logic was used as the tool to find all the minimal generators (mingens) from a set of implications. The method applies the Function `Cls` to guide the search of new minimal generator candidates. Specifically, given a set of attributes M and an implicational system Σ, the algorithm renders a mapping $mg_\Sigma \colon 2^M \to 2^{2^M}$ that satisfies the following:

$\forall X, Y \subseteq M$, $X \in mg_\Sigma(C)$ iff C is closed for $(\,)^+_\Sigma$ and X is a mingen for C.

Example 2 For the implicational system introduced in Example 1, the mapping mg_Σ is described as follows:

X	$\varnothing$	b	e	be	de	ace	bde	$acde$	$abcde$
$mg_\Sigma(X)$	$\varnothing$	b	e	be	d	a	bd	ad	ab
						c		cd	bc

Otherwise, X is not closed and $mg_\Sigma(X) = \varnothing$. Notice that $\varnothing$ is closed and $mg_\Sigma(\varnothing) = \{\varnothing\}$, i.e., $\varnothing$ is a minimal generator of the closed set $\varnothing$.

The algorithms we introduce in this section need to use the following operation for this kind of mappings. Given two mappings $mg_1, mg_2 \colon 2^M \to 2^{2^M}$, the mapping $mg_1 \sqcup mg_2 \colon 2^M \to 2^{2^M}$ is defined as

$$(mg_1 \sqcup mg_2)(X) = \texttt{minimals}(mg_1(X) \cup mg_2(X)) \quad \forall X \subseteq M$$

Thus, $Y \in (mg_1 \sqcup mg_2)(X)$ if and only if Y is a minimal set of $mg_1(X) \cup mg_2(X)$ in $(2^M, \subseteq)$, i.e., $Y \in mg_1(X) \cup mg_2(X)$ and there does not exist another attribute set $Z \in mg_1(X) \cup mg_2(X)$ such that $Z \subsetneq Y$.

We already have all the tools needed to define the Minimal Generator method. It was introduced in [6] and here we describe it as Function MinGen depicted below, together with an illustrative example of its application (see Example 3).

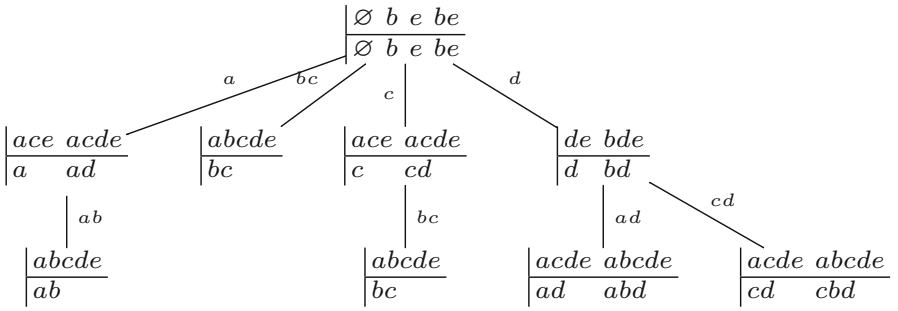

Fig. 2 Search tree for MinGen in Example 3

Function MinGen(M,Label,Guide,Σ)

input : M, the set of all attributes;
Label, an auxiliar set to build a minimal generator;
Guide, an auxiliar set to build a closed set;
Σ, an implicational system on M;
output: The mapping mg_Σ
begin
  foreach $X \subseteq M$ **do**
    $mg_\Sigma(X) := \varnothing$ (Guide,Σ):=$\mathtt{Cls}$(Guide,Σ);
  $M := M \setminus$Guide;
  Premises := $\{A \subseteq M \mid A \to B \in \Sigma$ for some $B \subseteq M\}$;
  ClosedSets:= $\{X \subseteq M \mid A \not\subseteq X$ for all $A \in$ Premises$\}$;
  foreach $X \in$ ClosedSets **do**
    mg_Σ(Guide $\cup X$) := {Label $\cup X$}
  foreach $A \in$ Premises **do**
    $mg_\Sigma := mg_\Sigma \sqcup$ MinGen(M, Label $\cup A$, Guide $\cup A$, Σ)
  return mg_Σ

Example 3 For the implicational system introduced in Example 1, the Function MinGen ($abcde$, $\varnothing$, $\varnothing$, $\{a \to c, bc \to d, c \to ae, d \to e\}$) returns

X	$\varnothing$	b	e	be	ace	$acde$	$abcde$	de	bde
$mg_\Sigma(X)$	$\varnothing$	b	e	be	a	ad	ab	d	bd
					c	cd	bc		

The search tree is shown in Fig. 2.

4 Reducing the search space in the MinGen method

In this subsection, we present a further MinGen method corresponding to a significantly enriched version of the original one. We have integrated a reduction to avoid the generation of redundant minimal generators and closed sets. The aim of this pruning is to identify redundant branches in the search space to avoid their exploration. This reduction must only be carried out if we can ensure that all the information regarding minimal generators has been collected in other branches. In Function MinGenRd this reduction strategy is implemented in line #1. Thus, to ensure that the information generated in one branch is superfluous, we design a prune based on set inclusion involving

all the nodes at the same level. We illustrate how this technique works in the following example:

Example 4 Given the search tree previously introduced in Example 3, the Function MinGenRd $(abcde, \varnothing, \varnothing, \{a \to c, bc \to d, c \to ae, d \to e\})$ applies a reduction strategy avoiding to open the branch whose label is a superset of another edge at the same level. Particularly, the branch labeled bc in Fig. 2 will not be opened because it is not minimal in the set $\{a, bc, c, d\}$.

Function MinGenRd(M,Label,Guide,Σ)

output: The mapping mg_Σ
begin
  foreach $X \subseteq M$ **do**
    $mg_\Sigma(X) := \varnothing$ (Guide,Σ):=Cls(Guide,Σ);
  $M := M \setminus$Guide;
[#1] Premises := Minimals$\{A \subseteq M \mid A \to B \in \Sigma$ for some $B \subseteq M\}$;
  ClosedSets:= $\{X \subseteq M \mid A \nsubseteq X$ for all $A \in$ Premises$\}$;
  foreach $X \in$ ClosedSets **do**
    mg_Σ(Guide $\cup X$) := {Label $\cup X$}
  foreach $A \in$ Premises **do**
    $mg_\Sigma := mg_\Sigma \sqcup$ MinGenRd(M, Label $\cup A$, Guide $\cup A$, Σ)
  return mg_Σ

4.1 A generalization of the reduction strategy

In this section, we propose a generalization of the reduction strategy by considering the subset inclusion test not only with the nodes at the same level, but with all the minimal generators computed before the opening of each branch. One step further is to take advantage of the minimal generators already computed to increase the number of branches not needed to be opened. As Function GenMinGen shows, we consider this generalized pruned in #1, and then, the list of minimal premises to be considered in each stage is built in #2.

Function GenMinGen(M,Label,Guide,Σ,MinGenList)

output: The mapping mg_Σ
begin
  foreach $X \subseteq M$ **do**
    $mg_\Sigma(X) := \varnothing$ (Guide,Σ):=Cls(Guide,Σ);
  $M := M \setminus$Guide;
  Premises := Minimals$\{A \subseteq M \mid A \to B \in \Sigma$ for some $B \subseteq M\}$;
  ClosedSets:= $\{X \subseteq M \mid A \nsubseteq X$ for all $A \in$ Premises$\}$;
  foreach $X \in$ ClosedSets **do**
    mg_Σ(Guide $\cup X$) := {Label $\cup X$}
  foreach $A \in$ Premises **do**
[#1] **if** *there no exists* $Y \in$ MinGenList *such that* $Y \subseteq A$ **then**
      $mg_\Sigma := mg_\Sigma \sqcup$ GenMinGen(M, Label $\cup A$, Guide $\cup A$, Σ, MinGenList)
[#2] **add** A **to** MinGenList;
  return (mg_Σ)

Notice that in Fig. 2 the branches labeled with *ad* and *cd* will now not be opened because in the previous level labels *a* and *c* were (respectively) opened. In the output of Function GenMinGen the minimal generators *ad* and *cd* appears in previous branches whereas *abd* and *cbd* are not computed because they are not really minimal generators (see the enumeration closed sets and minimal generators for this problem in Example 3).

4.2 Testing the performance of the reducing techniques

Once we have presented the original Minimal Generator method (MinGen) along with the two approaches (MinGenRd and GenMinGen) and an illustration of their search spaces has been shown, we present now a global comparison of the performance achieved by each of them.

For that matter, we have developed the corresponding implementations to apply the methods to a battery of sets of implications randomly generated. To evaluate this comparison, two different metrics are applied: (1) the execution time of the algorithm and (2) the number of nodes in the search tree built by the method. The reason for this selection becomes reasonable as follows. The execution time arises as the classical measure to test the performance, but it is always hardly linked to the resources we are working with. So, we add the number of nodes of the tree as a metric to compare these algorithms since it could better guide us to put forward an argument to decide which algorithm is the best as it is an architecture-independent value.

In addition, due to the intrinsic nature of execution time, it is also imperative to state that every experiment shown throughout this document has been repeated several times, so we can now write down the most reliable average values w.r.t. the execution time metric.

In the experiments, the hardware configuration used is: Intel(R) Core(TM) i7-6700HQ CPU 2.60 GHz, 8 Gb RAM memory, running over Windows 10. We have generated a battery of different inputs to be used in the sequential implementation to show the improvements of the methods, that is:

- A synthetic test. Contains 5 testing files with 50 implications built using 50 possible different attributes. The implications are randomly generated.
- A real dataset. We focused on MovieLens datasets[1] which have been widely used in education, research, and industry [13]. In this spirit, we chose the MovieLens10M dataset which is a dataset totally accessible through MovieLens web page and contains huge information about movies, genres, ratings, users, etc. From all this information, we are going to generate a table with 10.681 rows, where we face every movie with all the possible genres included in the dataset. The extraction of implications from the dataset renders a set of 19 attributes (genres) and a set of 245 implications.

Having said that, the results of the experiments are shown in Fig. 3. Giving an overall view of the results, it can be clearly seen how the reduction strategies significantly reduce both the number of nodes and the execution times. Indeed, experiment

[1] https://grouplens.org/datasets/movielens/.

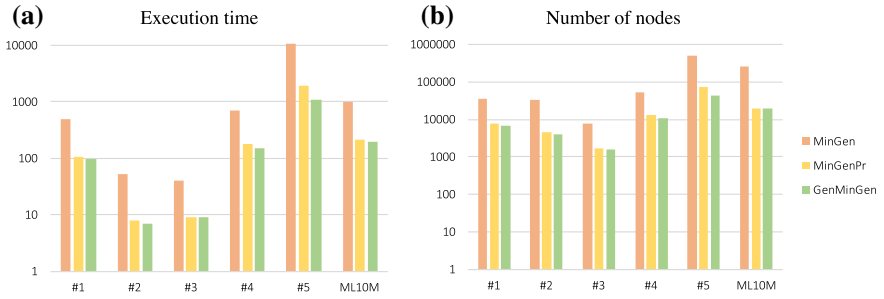

Fig. 3 Execution times (**a**) and number of nodes (**b**) results for the sequential experiments. Notice the logarithmic scale applied to the axes to better view the results

over 'sequential-5' file shows how both metrics have been drastically reduced. The execution time has been reduced from more than 10.000 s to less than 2.000 s. Besides, it is also worth to mention the reduction of the number of nodes which implies less resources needed in terms of memory and storage. Overall, like it was meant to happen, MinGen is surpassed by MinGenRd, and GenMinGen improves the latter as well.

Although GenMinGen has stated to have a better performance than MinGenRd (and both of them better than MinGen), the former is not suitable for parallelization. This is because it demands communication among all the subproblems previously obtained. This breaks our parallel philosophy, avoiding the use of massive computation. For this reason, we have studied a parallelization of the MinGenRd method.

5 Parallel computation of minimal generators

We have already shown the improvements reached by the reduction strategies within the minimal generators methods. However, once it comes the case that we want to use these methods over larger inputs, we can figure out in the light of the results obtained, that execution times of the sequential methods would go too far to be easily handled. Nonetheless, taking into account that every branch of the tree created by the methods constitutes a problem on its own, we can think about resolving them simultaneously and, later, combining the partial solutions to get the final output. This natural view of the problem as a parallel execution, provided by our logic-based methods, leads us to develop a new and parallel version of the minimal generators method.

Before going further, we introduce that the supercomputing resources and architecture that have been used to run the following parallel experiments are those provided by the Supercomputing and Bioinnovation Center of the University of Málaga.[2] In particular, we have developed each experiment using 32 nodes cluster SL230, counting on 16 cores and 64GB RAM memory and 7 nodes cluster DL980, counting on 80 cores and 2 TB RAM memory. Communications are carried out on Infiniband Net FDR and QDR. These cores are reserved just for our use so we can assess reliable results regarding execution times.

[2] https://www.scbi.uma.es/.

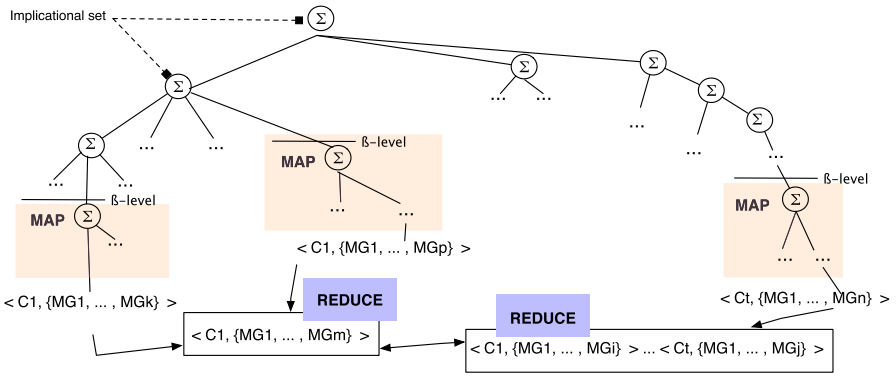

Fig. 4 Illustration of the map-reduce paradigm used in the design of MinGenRdPar

5.1 Parallel algorithm

We have developed a parallel implementation of the methods following the framework shown by the authors in [2]. In summary, it takes advantage of massive computation following the map-reduce paradigm. This parallel implementation of the methods performs in two stages. First one is the splitting stage. There, a single core builds the search space until it reaches a certain depth level of the tree. The goal is to split the original problem into several problems that can be treated by multiples cores, playing the role of a "Map" procedure. It goes this way until the size of the current node becomes lower than a given size (β) w.r.t the number of implications. From that moment on, each of these subproblems is resolved in parallel using multiple cores until we reach the leaves of the tree. Finally, a single core is used to compose the global result by merging the different outputs of each subproblems into a final one. This becomes necessary in order to eliminate redundancies to obtain the *minimal* generators. It is in this stage where the Reduce procedure acts as a filter to avoid redundant information in the global output. The architecture of our approach is depicted in Fig. 4.

In addition, we show the pseudocode of our parallel $MinGenRd$ algorithm in Function MinGenRdPar. We have emphasized the main stages of our implementation that likely correspond to the Map-Reduce paradigm.

The β level, named BOV (Break-Off Value) in the algorithm, is a critical parameter since it is in charge of deciding when to split the original problem into subproblems to manage them in parallel, generating a new Map task. As we will broadly explain later in Sect. 5.3, a hard experimental study has been needed to determinate the most suitable value for BOV.

Function MinGenRdPar(M,Label,Guide,Σ,BOV)

output: The mapping mg_Σ
begin
  foreach $X \subseteq M$ **do** $mg_\Sigma(X) := \varnothing$ (Guide,Σ):=Cls(Guide,Σ);
  $M := M \setminus$Guide;
  Premises := Minimals$\{A \subseteq M \mid A \to B \in \Sigma$ for some $B \subseteq M\}$;
  ClosedSets:= $\{X \subseteq M \mid A \nsubseteq X$ for all $A \in$ Premises$\}$;
  foreach $X \in$ ClosedSets **do** mg_Σ(Guide $\cup X$) := {Label $\cup X$}
  If $\mid \Sigma \mid \leq BOV$ **then do in parallel**
    // [MAP]
    foreach $A \in$ Premises **do**
      // [REDUCE]
      $mg_\Sigma := mg_\Sigma \sqcup$ MinGenRd(M, Label $\cup A$, Guide $\cup A$, Σ)
    return mg_Σ
  else
    // [Splitting stage]
    foreach $A_k \in$ Premises **do**
      $mg^k_\Sigma =$ MinGenRdPar(M, Label $\cup A_k$, Guide $\cup A_k$, Σ, BOV)
    forall the mg^k_Σ **do**
      // [REDUCE]
      $mg_\Sigma := mg_\Sigma \sqcup mg^k_\Sigma$;
    return mg_Σ

5.2 Sequential versus parallel experiments

This time, we have carried out an experiment to test the performance of the parallel implementation MinGenRdPar versus the sequential implementation MinGenRd. The input files we are going to use will count on numbers raised up to 150 attributes and 150 implications. Even with these numbers, three times higher than the sequential experiment in Sect. 4.2, parallel version obtains results in an admissible time as we can notice on the results given in Table 1. Looking closely, Table 1 collects the following information from left to right: (1) identifier of the input file and the method used to resolve, (2) execution time of the sequential version, (3) number of subproblems generated by the splitting stage of the parallel implementation, (4) execution time of the splitting stage, (5) execution time of parallel resolution, (6) execution time of the whole process, (7) number of nodes of the generated tree and (8) number of minimal generators obtained.

In conclusion, thanks to the application of the parallel implementation, we have been able to reduce the execution times from hours and days (experiments over #-{2,3,7,8,10}-sequential) down to just a few minutes. In other words, parallel computation improves the execution times even when dealing with big-sized problems.

5.3 Estimation of the BOV

There is a crucial aspect we have to keep in mind in the first stage of the parallel implementation, i.e., the splitting. We need to decide when a subproblem has to be generated (Map procedure) to be simultaneously executed. To this end, we are going to

Table 1 Comparison between both sequential and parallel versions of MinGenRd algorithm applied to big-sized problems

Problem	Seq.time (s)	Subp	Split$_t$ (s)	Parallel$_t$ (s)	Total$_t$ (s)	Nodes	MinGens
#1	***43***	11	3	1	*4*	374	216
#2	***17.352***	347	220	55	*275*	54.375	6.273
#3	***33.338***	822	2.338	251	*2.589*	68.531	6.529
#4	***4.612***	344	350	97	*447*	25.477	2.478
#5	***1.585***	168	432	30	*462*	12.522	1.159
#6	***1.653***	79	35	7	*42*	8.110	1.436
#7	***107.238***	1.754	958	242	*1.200*	262.621	9.113
#8	***61.381***	966	253	188	*441*	257.267	5.538
#9	***372***	24	7	2	*9*	1.726	683
#10	***7.484***	277	186	65	*251*	45.962	2.969

use a value, denominated BOV, which represents the cardinal of the set of implications of the current node, since we have empirically observed that as long as this cardinal grows, the longer the branch of the tree uses to reach. The BOV is computed as a percentage of the size of the input that has to be estimated to balance the work done by each core.

Yet, deciding the BOV is a decisive task of the current investigation due to the following difficulties. On the one hand, if we decide to stop at a level near to the root of the tree by selecting a low BOV (i.e., a high number of implications), we are certainly reducing the execution time of the splitting stage and only a few subproblems would be created. Accordingly, since the tree would not have been able to spread out yet, then we will not have enough material to be managed in parallel using different cores. On the other hand, if we stop the split in a depth level far from the root, the splitting process will surely create a large amount of subproblems but its execution time would surely grow up. This is why we have selected a BOV empirically, as it is really difficult to get the right value by just analyzing the input theoretically. However, after making a lot of experiments, several aspects worth to be mentioned and this section stands to this effect.

We have hitherto established a BOV of 140 (recall that our number of implications is 150) within the experiments as it has shown to be the best one to leverage parallelism based on our experience. That is to say, we are choosing a BOV that corresponds to the $\approx$ 93.33% of the original set of implications. However, in order to explore this fact, we have repeated our parallel experiments using different BOVs. We take lower BOVs that makes the splitting stage to go deeper into the tree, specifically, we will use both 130 and 100 BOVs, i.e., the $\approx$ 86 and $\approx$ 66% of the whole original set of implications, respectively. This particular selection of values is not made by chance but with the aim of conveying several situations we have found after carrying out plenty of experiments which we come to analyze next. Results concerning execution times are gathered in Table 2.

Table 2 Experiments using different BOV within the parallel version of MinGenRd algorithm applied to big-sized problems

Problem	Subp	Split$_t$ (s)	Parallel$_t$ (s)	Total$_t$ (s)	BOV (%)
#1	0	44	–	**44**	**66.67**
	0	41	–	***41***	***86.67***
	11	3	1	*4*	*93.33*
#2	0	18.533	–	**18.533**	**66.67**
	885	8.532	58	***8.590***	***86.67***
	347	220	55	*275*	*93.33*
#3	0	33.377	–	**33.377**	**66.67**
	0	33.285	–	***33.285***	***86.67***
	822	2.338	251	*2.589*	*93.33*
#4	0	4.858	–	**4.858**	**66.67**
	308	3.703	22	***3.725***	***86.67***
	344	350	97	*447*	*93.33*
#5	0	1.601	–	**1.601**	**66.67**
	0	1.547	–	***1.547***	***86.67***
	168	432	30	*462*	*93.33*
#6	0	772	–	**772**	**66.67**
	144	492	11	***503***	***86.67***
	79	35	7	*42*	*93.33*
#7	0	167.451	–	**167.451**	**66.67**
	5.412	96.433	295	***96.728***	***86.67***
	1.754	958	242	*1.200*	*93.33*
#8	0	75.060	–	**75.060**	**66.67**
	5.344	41.404	375	***41.779***	***86.67***
	966	253	188	*441*	*93.33*
#9	0	82	–	**82**	**66.67**
	24	42	2	***44***	***86.67***
	24	7	2	*9*	*93.33*
#10	0	9.438	–	**9.438**	**66.67**
	697	6.569	50	***6.619***	***86.67***
	277	186	65	**251**	**93.33**

There are several outcomes within that table. First and foremost, except for problem #1 where there exist a subtle difference, all the problems behave worse regarding the execution time when we delay the splitting point. To find out an explanation, we put our attention in the number of generated subproblems where four different behaviors arise when we vary the BOV. We proceed to enumerate each of them with the support of the results obtained in Table 2.

Let $BOV_1 > BOV_2$, and let Sp_1, Sp_2 the number of generated subproblems associated with each of them. So, there might be situations where:

- $Sp_1 < Sp_2$. The algorithm has more scope to expand the tree and generate more subproblems. In this way, parallelism would take advantage but the time needed to split scupper the global execution time. Problems #{2,6,7,8,10} reflect this situation.
- $Sp_1 > Sp_2$. It can happen that even going deeper into the tree we obtained less nodes, i.e., less subproblems. This is because there are branches that end before reaching the BOV and they are resolved within the split stage, so the execution time grows higher. We can see this case in problem #4.
- $Sp_1 \neq 0 \wedge Sp_2 = 0$. Following on from the previous point, we can fall in an extreme situation where no subproblems are generated within the split stage because every branch of the tree ends up before the splitting point. This situation may be considered as the worst case since the parallel implementation performs the same as the sequential one. This happens for every problem when using a $BOV \approx 66\%$ and also in problems #{1,3,5} even with a $BOV \approx 86\%$.
- $Sp_1 = Sp_2$. This reflects an entangled situation in which lower levels of the tree may count on the same number of nodes than the higher ones. The point is that when we go deeper in the tree we may be generating more nodes as the tree expands and so the number of subproblems would grow up, but also, it may be several branches that end before going deeper and so it reduces the number of nodes. Therefore, with this additions and subtractions, the global calculation of subproblems can end up in a draw with different BOVs as shown in problem #9. But that's not all. Problem #9 indeed shows the same number of subproblems for $BOV \approx 93.33\%$ and $BOV \approx 86\%$, however, execution time is not the same, actually, it is worse for a $BOV \approx 86\%$ since we have stretched on the split stage.

As we have already mentioned before and after the analysis shown in this part, it becomes clear that trying to infer the best BOV for an experiment is not an easy issue so far as it depends on many possible situations we can face when dealing with the minimal generators enumeration problem.

5.4 Estimation of the suitable number of cores

Up to now, we can think it is just a matter of resources that we can increase the input size and still get results within a reasonable time. However, it is not so obvious and this subsection is dedicated to analyze this fact. For this purpose, the big-sized problems used before in Sect. 5.2 will be tested again by using 16, 32, 48, 64 and 80 cores each time. Results are shown in Table 3.

This is not up to expectations results. Although resources are better now, results do not come along. It can be seen that, we do improve the performance by using more cores (e.g., from 16 to 64 cores), however, there comes a moment where no profit is obtained; results are pretty similar for 48, 64 and 80 cores. This is due to the fact that there are several branches in the tree (maybe just one) that take so long to finish and then, no matter how much we improve the resources, the global time remains almost the same. The problem is that we cannot predict for the time being whether a branch will result in a long or a short one. This situation stalls our first intentions of blindly growing the amount of resources and shows up that, the number of cores for

Table 3 Execution times (in seconds) results obtained when increasing the number of cores applied

Problem	Number of cores				
	16	32	48	64	80
#1	5	4	*3*	*3*	*3*
#2	310	275	199	*190*	197
#3	2.742	2.589	2.122	2.078	*2.075*
#4	512	447	444	*440*	446
#5	598	462	*416*	445	442
#6	50	42	*34*	35	*34*
#7	1.457	1.200	1.078	*1.010*	1.012
#8	499	441	432	430	*425*
#9	9	9	*7*	*7*	*7*
#10	267	251	196	*194*	195

Table 4 Parallel minimal generator algorithm applied to a real-world dataset

Problem and Method	Subp	Split_t (s)	Parallel_t (s)	Total_t (s)	Nodes	MinGens
Mushrooms-parallel	224	152	9	161	81.363	17.127

this problem can be established to 64 as an optimal value, achieving a balance between resources needed and benefits obtained.

5.5 Real-world dataset experiment

To properly finalize this section, we bring now the results obtained when applying our algorithm to a real-world dataset. In particular, we put our focus in the Mushroom Data Set[3] accessible from the website of the University of California, Irvine (UCI).[4] Basically, this dataset includes descriptions of hypothetical samples corresponding to 8.124 species of gilled mushrooms using 22 attributes. On this dataset, an adaptation is made in order to convert multi-valued information into binary information. As a result, we will manage a dataset with 126 attributes. The number of implication inferred from this dataset is 1.587. We remark that with this number of implications, sequential version of the algorithm does not finish.

To carry out this experiment we use the conclusions reached by the previous sections. Therefore, we will use 64 different cores and a $BOV \approx 93.33\%$, i.e., $BOV = 1.481$, as they seem to be the best values to proceed. The results of this experiment are shown in Table 4 where it clearly arises how our algorithm fulfills when dealing with a big-sized and real-world dataset.

[3] https://archive.ics.uci.edu/ml/datasets/mushroom.

[4] http://archive.ics.uci.edu/ml/.

6 Conclusions and future works

Enumerating all the closed sets and their minimal generators is a hard problem, but essential in several areas and an opportunity to show the benefits of FCA for real applications. Such information can be obtained from a dataset or from a set of implications. In this work we focus on this second problem, which have been approached in a lesser extent. In order to face this task, this work presents an efficient reduction of the search space technique to improve the performance of minimal generator enumeration. The new method has been designed to fit the Map-Reduce architecture. Here is where parallel computation comes to make it possible for us to deal with such amount of information.

In particular, we have designed two new methods (MinGenRd and GenMinGen) implementing pruning strategies to reduce the search space. The empirical study proves how these methods surpass the previous one in the literature (MinGen). We also establish the adequacy of MinGenRd on being implemented following a parallel implementation by means of the Map-Reduce paradigm. Therefore, as the main contribution of this work, we propose the parallel implementation of MinGenRd method. The empirical study proves the very significative improvement achieved w.r.t the original sequential version. The parallel methods to compute minimal generators can make really usable these methods in practical applications.

To properly characterize the parallel approach, we have developed two battery of experiments: (1) we have established the BOV threshold, a parameter included in the algorithm to effectively apply the Map procedure taking the most of this paradigm and, (2) we have established the number of cores need to execute this method, thus delimiting the resources of the parallel architecture.

As future works, we plan to study how to use different software and hardware resources to achieve a parallel implementation of the best sequential method, that we have proposed in this paper, GenMinGen method. Such implementation requires a complex design to store the previously generated minimal generators without interfering the parallel execution.

A deeper study of the best values to be considered as BOV to maximize the results will be also developed, such as taking into account another parameters beyond the implication set cardinality.

Acknowledgements The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Supercomputing and Bioinnovation Center of the University of Málaga - Andalucía Tech (SCBI), particularly to Dr. Rafael Larrosa and Dr. Darío Guerrero. We also want to mention the orientation provided by Dr. José Antonio Onieva to identify the properties of our algorithm for a better classification of its design.

References

1. Armstrong WW (1974) Dependency structures of data base relationships. In: IFIP Congress, pp 580–583
2. Benito-Picazo F, Cordero P, Enciso M, Mora A (2017) Reducing the search space by closure and simplification paradigms. J Supercomput 73(1):75–87

3. Buchi JR, Siefkes D (1990) Finite automata, their algebras and grammars. Springer, New York Inc, Secaucus
4. Codd EF (1970) A relational model of data for large shared data banks. Commun ACM 13(6):377–387
5. Cohen E, Datar M, Fujiwara S, Gionis A, Indyk P, Motwani R, Ullman JD, Yang C (2001) Finding interesting associations without support pruning. IEEE Trans Knowl Data Eng 13(1):64–78
6. Cordero P, Enciso M, Mora A, Ojeda-Aciego M (2012) Computing minimal generators from implications: a logic-guided approach. In Szathmary L, Priss U (eds) Proceedings of the Ninth International Conference on Concept Lattices and Their Applications, Fuengirola (Málaga), Spain, October 11–14, 2012, volume 972 of CEUR Workshop Proceedings, pp 187–198. CEUR-WS.org
7. Cordero P, Mora A, Enciso M, de Guzmán IP (2002) SLFD logic: elimination of data redundancy in knowledge representation. Lect Notes Comput Sci 2527:141–150
8. de Moraes NRM, Dias SM, Freitas HC, Zárate LE (2016) Parallelization of the next closure algorithm for generating the minimum set of implication rules. Artif Intell Res 5(2):40–54
9. Doignon J, Falmagne J (1998) Knowledge spaces. Springer, Berlin
10. Dong GZ, Jiang CY, Pei J, Li JY, Wong L (2005) Mining succinct systems of minimal generators of formal concepts. Proc Database Syst Adv Appl 3453:175–187
11. Ganter B, Wille R (1999) Formal concept analysis: mathematical foundations. Springer, Berlin
12. Guigues JL, Duquenne V (1986) Famille minimale d'implications informatives résultant d'un tableau de données binaires. Math Sci Hum 24(95):5–18
13. Harper FM, Konstan JA (2015) The movielens datasets: history and context. ACM Trans Interact Intell Syst 5(4):19:1–19:19
14. Hu X, Wei X, Wang D, Li P (2007) A parallel algorithm to construct concept lattice. In Lei J (ed) Proceedings of the Fourth International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2007, 24–27 August 2007, Haikou, Hainan, China, vol 2, pp 119–123. IEEE Computer Society
15. Kuznetsov SO, Obiedkov SA (2002) Comparing performance of algorithms for generating concept lattices. J Exp Theor Artif Intell 14(2–3):189–216
16. Maier D (1983) The theory of relational databases. Computer Science Press, Rockville
17. Missaoui R, Nourine L, Renaud Y (2010) An inference system for exhaustive generation of mixed and purely negative implications from purely positive ones. In: CEUR Workshop Proceedings, vol 672, pp 271–282
18. Missaoui R, Nourine L, Renaud Y (2012) Computing implications with negation from a formal context. Fundam Inf 115(4):357–375
19. Mora A, Enciso M, Cordero P, Fortes I (2012) Closure via functional dependence simplification. Int J Comput Math 89(4):510–526
20. Nishio N, Mutoh A, Inuzuka N (2012) On computing minimal generators in multi-relational data mining with respect to 0-subsumption. In: CEUR Workshop Proceedings, vol 975, pp 50–55
21. Poelmans J, Ignatov DI, Kuznetsov SO, Dedene G (2013) Formal concept analysis in knowledge processing: a survey on applications. Expert Syst Appl 40(16):6538–6560
22. Qu K, Zhai Y, Liang J, Chen M (2007) Study of decision implications based on formal concept analysis. Int J Gen Syst 36(2):147–156
23. Rodríguez-Lorenzo E, Cordero P, Enciso M, Mora Á (2017) Canonical dichotomous direct bases. Inf Sci 376:39–53