Ann Math Artif Intell
DOI 10.1007/s10472-013-9372-8

A fast compound algorithm for mining generators, closed itemsets, and computing links between equivalence classes

Laszlo Szathmary · Petko Valtchev · Amedeo Napoli · Robert Godin · Alix Boc · Vladimir Makarenkov

Abstract In pattern mining and association rule mining, there is a variety of algorithms for mining frequent closed itemsets (FCIs) and frequent generators (FGs), whereas a smaller part further involves the precedence relation between FCIs. The interplay of these three constructs and their joint computation have been studied within the formal concept analysis (FCA) field yet none of the proposed algorithms is scalable. In frequent pattern mining, at least one suite of efficient algorithms has been designed that exploits basically the same ideas and follows the same overall computational schema. Based on an in-depth analysis of the aforementioned interplay that is rooted in a fundamental duality from hypergraph theory, we propose a new schema that should enable for a more parsimonious computation. We exemplify

L. Szathmary (✉)
Faculty of Informatics, Department of IT, University of Debrecen,
4010 Debrecen, Pf. 12, Hungary
e-mail: Szathmary.L@gmail.com

P. Valtchev · R. Godin · A. Boc · V. Makarenkov
Dépt. d'Informatique, UQAM, Université de Montréal, C.P. 8888, Succ. Centre-Ville,
Montréal H3C 3P8, QC, Canada

P. Valtchev
e-mail: valtchev.petko@uqam.ca

R. Godin
e-mail: godin.robert@uqam.ca

A. Boc
e-mail: boc.alix@uqam.ca

V. Makarenkov
e-mail: makarenkov.vladimir@uqam.ca

A. Napoli
LORIA (CNRS - Inria NGE - Université de Lorraine) BP 239,
54506 Vandœuvre-lès-Nancy, France
e-mail: Amedeo.Napoli@loria.fr

the new schema in the design of *Snow-Touch*, a concrete FCI/FG/precedence miner that reuses an existing algorithm, *Charm*, for mining FCIs, and completes it with two original methods for mining FGs and precedence, respectively. The performance of *Snow-Touch* and of its closest competitor, *Charm-L*, were experimentally compared using a large variety of datasets. The outcome of the experimental study suggests that our method outperforms *Charm-L* on dense data while on sparse one the trend is reversed. Furthermore, we demonstrate the usefulness of our method and the new schema through an application to the analysis of a genome dataset. The initial results reported here confirm the capacity of the method to focus on significant associations.

Keywords Pattern mining · Formal concept analysis · Frequent itemset · Frequent generator · Hypergraph · Compound algorithm

Mathematics Subject Classifications (2010) 68Txx · 68T10 · 62H30 · 91C20

1 Introduction

Pattern mining and association discovery [1] in data mining (DM) are aimed at pinpointing the most frequent patterns of *items*, or *itemsets*, and the strongest *associations* between items that are hidden in a large database of *transactions*. Pattern/association rule mining has been successfully applied to such diverse tasks as market basket analysis, fault prediction in telecoms, bank fraud detections, etc. The main challenge of the task is the potentially huge size of its output that is only partially trimmed by the classical quality metrics of *support* and *confidence*. Indeed, a large number of the qualifying rules and patterns brings no additional information that is not embedded in a different rule. To get rid of these redundancies, mining methods focus on *bases*, which are reduced yet lossless representations of the entire family of associations/patterns.

Popular rule bases include the *minimal non-redundant association rules* [2] and the so called *informative basis* (see a list in [3]). Most of the existing bases either have been formulated within the formal concept analysis (FCA) field [4] or involve structures that themselves have. For instance, each of the minimal non-redundant association rules, is made of a *frequent generator* (FG) as a premise and a *frequent closed itemset* (FCI) as a conclusion. Hence its construction requires the knowledge of all FCIs and their respective FGs. Furthermore, the *informative basis* involves, beside FGs and FCIs, the (inclusion-induced) precedence links between FCIs. FCIs and FGs play symmetric role in the Boolean lattice of all itemsets (i.e. the search space for pattern mining): they are, respectively, the maximal and the minimal elements of the equivalence classes induced by the function mapping itemsets to the transactions they appear in. Precedence, in turn, stems from the semi-lattice of all FCIs.

We investigate the computation of iceberg lattices, i.e., FCIs plus precedence, together with the belonging FGs. In the DM literature, a number of methods exist that target FCIs by first discovering the respective FGs, e.g. the levelwise FCI miners *A-Close* [5] and *Titanic* [6]. More recently, a number of extensions of the popular FCI miner *Charm* [7] have been published that output two or all three of the above components. The basic one, *Charm-L* [8], produces FCIs with precedence links, whereas two further versions thereof would produce the FGs as well (see [9, 10]).

In the FCA field, where the typical datasets or formal contexts, are much smaller than a reasonable transaction database, the computational emphasis has been initially put on the construction of the entire closed itemset family (alias concept intents), thus disregarding the frequency aspect of concepts. Moreover, generators have not been explicitly targeted right from the beginning. Historically, the first method whose output combines closures, generators and precedence has been presented in [11] yet this fact is dug into a different terminology and a somewhat incomplete result (see explanations below). The earliest method to explicitly target all three components that we are aware of is to be found in [12] while an improvement was published in [13]. From a DM point of view, all FCA-inspired methods have a common drawback: they scale poorly on large datasets due to repeated scans of the entire database (either for closure computations or as part of an incremental restructuring approach). In contrast, *Charm-L* exploits a *vertical* encoding of the database that helps mitigate the impact of the large object, alias transaction, set on cost.

Finding all FCIs in a dataset and discovering the precedence relation between them, one can construct the so-called Hasse diagram of a concept lattice (see Fig. 1c), where FCIs represent the maximal elements of the concepts (nodes), w.r.t. set inclusion. Concepts in a lattice can be decorated with generators that are minimal elements of the nodes. If a support threshold is set up, then non-frequent nodes of a lattice can be removed, which results in an iceberg lattice. Thus, an iceberg lattice is

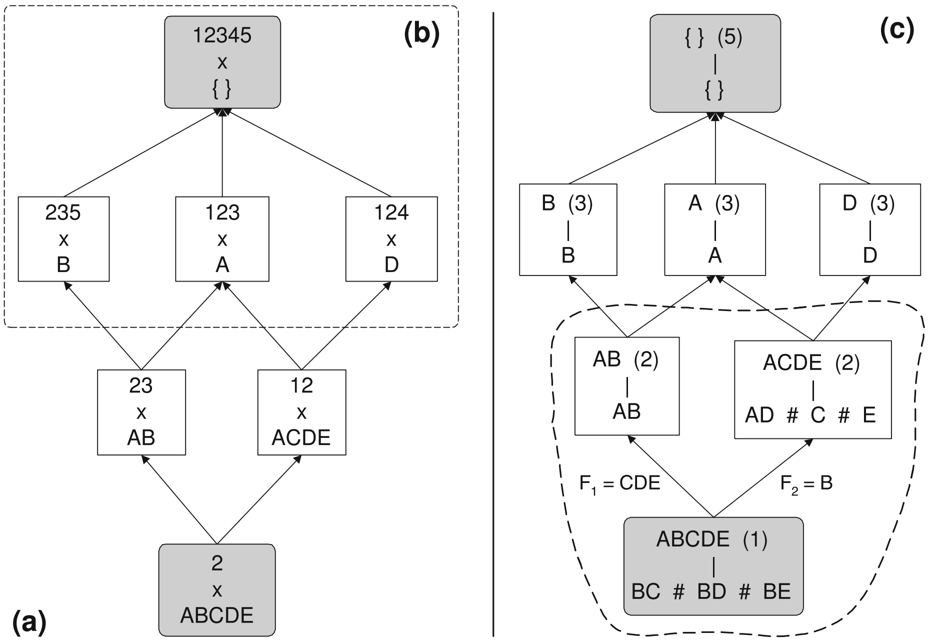


Fig. 1 Concept lattices of dataset $\mathcal{D}$. **a** The entire concept lattice. **b** An iceberg part of (**a**) with *min_supp* = 3 (indicated by a *dashed rectangle*). **c** The concept lattice with generators drawn within their respective nodes

a semi-lattice composed of frequent nodes only (see Fig. 1b). An iceberg lattice, in addition to FCIs, also has precedence relations.

The FGs are actually minimal transversals of a hypergraph defined on top of their FCI and its neighbors. This fact was first mentioned by Pfaltz in Theorem 1, yet under a different terminology (see below), and was extensively exploited in the design of the algorithm in [12]. Later on, the same results were employed to ground the FG computation in [10]. When both strategies for FG computing are compared, the transversal-based one seems to be substantially more efficient, at least in the case of *Charm-L* and its extensions. Moreover, the efficiency of the latter algorithm w.r.t. the FCA methods is rather appealing, yet the size of the datasets being on a constantly increasing course, it is crucial to regularly overcome the existing methods with quicker ones. It is therefore legitimate to look for possible improvements, both in the overall computing schema of *Charm-L* and in the individual techniques used.

Our analysis shows that, however efficient the method is, its design is not nearly optimal. Indeed, while we feel that FCIs and FGs mechanisms leave little space of improvement, precedence computation is less so as it benefits from no particular insight. As it is carried out at any individual FCI discovery and performs searches through potentially the entire supporting tree structure, many FCIs from distant parts of the search space may get compared. We therefore felt that there is space for improvement, e.g., by bringing in techniques that operate locally. An appealing research track seems to lay in the exploration of the important duality that exists between a simple hypergraph and its transversal hypergraph. We show that this allows the computation dependencies between individual tasks to be inverted, i.e. compute precedence links of an FCI from its FGs rather than the other way round (and thus define a new overall workflow). To clarify this point, we chose to start our investigation by a less intertwined algorithmic schema, i.e. by a modular design so that each individual task could be targeted by the best among a pool of alternative methods.

Actually, the present paper is an updated and extended version of two preceding papers [14, 15]. Here, we describe a first step in our study, *Snow-Touch*, which is composed of three individual and independent methods, one per task, whose inputs/outputs have been wired w.r.t. our new schema. In fact, our method relies on the aforementioned *Charm* algorithm for mining FCIs whereas the vertical miner *Talky-G* is responsible for FGs. *Charm* and *Talky-G* compose a compound miner, *Touch* [15], which also comprises an FGs-to-FCIs matching mechanism. The final step is performed by the *Snow* method [14], which extracts the precedence links from FCIs and FGs. Here, in addition to [14, 15], we assemble the algorithms introduced in these two papers in a compound and modular algorithm. Moreover, we propose a series of experiments showing the merits and the efficiency of our approach.

To verify the purposefulness of the new design schema and its straightforward realization, we studied the performance of a Java implementation of *Snow-Touch* on a wide range of datasets. As a comparison basis, we took the natural competitor of our method, *Charm-L* (author's version in C++). In our tests *Snow-Touch* outperformed the concurrent method on all dense datasets. This was not readily anticipated as our initial modular design brought some computational overhead, e.g. the extra matching step between FGs and FCIs. Moreover, we experimented *Snow-Touch* with real-world data: we applied it to a genomic dataset describing gene distributions among known species of prokaryotes. As the first results of a larger-scale analytic study (described below) indicate, our method can be useful and enables a quick focus on meaningful gene associations.

The rest of the paper is as follows: background on pattern mining and concept analysis is provided in Section 2. Problem statement is given in Section 3. Section 4 presents the different components of the compound algorithm *Snow-Touch*, namely *Talky-G* (Section 4.1, including an overview of vertical algorithms too), *Touch* (Section 4.2), and *Snow* (Section 4.3, including basic concepts of hypergraphs too). Related work is provided in Section 5. Experimental evaluations are provided in Section 6. Conclusions and future work directions are given in Section 7.

2 Preliminaries

In the following, we recall basic concepts from frequent pattern mining and formal concept analysis (FCA). The vocabulary and notations come from the dedicated literature but, whenever necessary, parallels are drawn to support the comprehension.

The following 5×5 sample dataset: $\mathcal{D} = \{(1,\ ACDE), (2,\ ABCDE), (3,\ AB), (4,\ D), (5,\ B)\}$ will be used as a running example. Henceforth, we refer to it as dataset $\boldsymbol{\mathcal{D}}$.

We consider a set of *objects* or *transactions* $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$, a set of *attributes* or *items* $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, and a relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$. A set of items is called an *itemset*. Each transaction has a unique identifier (*tid*), and a set of transactions is called a *tidset*. The tidset of all transactions sharing a given itemset X is its *image*, denoted $t(X)$. For instance, the image of $\{A, B\}$ in $\mathcal{D}$ is $\{2, 3\}$, i.e., $t(AB) = 23$ in our separator-free notation for sets. The *length* of an itemset is its cardinality, whereas an itemset of length k is called a k-itemset. The (absolute) *support* of an itemset X, denoted by $supp(X)$, is the size of its image, i.e. $supp(X) = |t(X)|$. An itemset X is called *frequent*, if its support is not less than a given *minimum support* (denoted by *min_supp*), i.e. $supp(X) \geq min\_supp$. An equivalence relation is induced by t on the power-set of items $\wp(\mathcal{A})$: equivalent itemsets share the same image ($X \cong Z$ iff $t(X) = t(Z)$). Consider the equivalence class of X, denoted $[X]$, and its extremal elements w.r.t. set inclusion. $[X]$ knowingly admits a unique maximum (a *closed* itemset), and a set of minimal elements (*generator* itemsets). The following definition thereof exploits the monotony of *supp* upon $\subseteq$ within $\wp(\mathcal{A})$:

Definition 1 An itemset X is *closed* (*a generator*) if it has no proper superset (subset) with the same support.

A *closure* operator underlies the set of closed itemsets; it assigns to X the maximum of $[X]$ (denoted by $\gamma(X)$). Naturally, $X = \gamma(X)$ for closed X. Generators, a.k.a. *key-sets* in database theory, represent a special case of free-sets [16]. For instance, in our dataset $\mathcal{D}$, B and C are generators, with closures B and $ACDE$, respectively (see Fig. 1c).

In [7], a subsumption relation is defined as well: X *subsumes* Z, iff $X \supset Z$ and $supp(X) = supp(Z)$. By Definition 1, if Z *subsumes* X, then Z cannot be a generator.

The following property, which is part of the folklore in the domain, generalizes this observation. It basically states that the generator family forms a downset within the Boolean lattice $\langle \wp(\mathcal{A}), \subseteq \rangle$:

Property 1 Given $X \subseteq \mathcal{A}$, if X is a generator, then $\forall Y \subseteq X$, Y is a generator. Equivalently, if X is not a generator, $\forall Z \supseteq X$, Z is not a generator.

The FCI and FG families are well-known reduced representations [17] for FIs, which jointly compose non-redundant bases of valid association rules, e.g. the *generic basis* [3]. Further bases require the inclusion order $\leq$ between FCIs or its transitive reduction $\prec$, i.e. the precedence relation. If $X \prec Y$, then Y is said to be the *immediate predecessor* of X.

The FCI family of a dataset together with $\prec$ compose the Hasse diagram of the *iceberg lattice*, which in turn corresponds to the frequent part of the closed itemset (CI) lattice, a.k.a. the *intent lattice* of a context [4]. In Fig. 1, views (a) and (b) depict the concept lattice of dataset $\mathcal{D}$ and its iceberg part, respectively.

3 Problem statement

A schema where first comes the lattice, then the generators (which are seen as an extra) could appear more intuitive from an FCA point of view. However, it is less natural and eventually less profitable for data mining. Indeed, while a good number of association rule bases would require the precedence links in order to be constructed, FGs are used in a much larger set of such bases and may even constitute a target structure of their own. Hence, a more versatile mining method would only output the precedence relation (and compute it) as an option, which is not possible with the current design schema. More precisely, the less rigid order between the steps of the combined task would be: (1) FCIs, (2) FGs, and (3) precedence. This basically means that precedence needs to be computed at the end, independently from FG and FCI computations (but may rely on these structures as input). Moreover, the separation of the three steps ensures a higher degree of modularity in the design of the concrete methods following our schema: any combination of methods that solve an individual task could be used, leaving the user with a vast choice. On the reverse side of the coin, total modularity comes with a price: if FGs and FCIs are computed separately, an extra step will be necessary to match an FCI to its FGs.

4 Searching for generators, closed itemsets, and links between equivalence classes: the Snow-Touch algorithm

We describe hereafter a method which relies exclusively on existing algorithmic techniques. These are combined into a single global procedure, called *Snow-Touch* in the following manner: the FCI computation is delegated to the *Charm* algorithm which is also the basis for *Charm-L* (see Section 5 for more details on *Charm-L*). FGs are extracted by our own vertical miner *Talky-G*. The two methods together with an FG-to-FCI matching technique form the *Touch* algorithm [15]. Finally, precedence is retrieved from FCIs with FGs by the *Snow* algorithm [14] using a ground duality result from hypergraph theory.

In summary, we contribute here a novel computation schema for iceberg lattices with generators (hence a **new lattice construction approach**). Moreover, we derive an efficient FCI/FG/precedence miner (especially on dense sets). We also demonstrate the practical usefulness of *Snow-Touch* as well as of the global approach for association mining based on generic rules.

4.1 Searching for generators: the Talky-G algorithm

Talky-G is a vertical FG miner. In this section we provide background on vertical algorithms such as *Eclat* and *Charm*. These two algorithms use the same pre-order traversal strategy. In contrast, *Talky-G* applies the so-called *reverse pre-order traversal*, which is detailed in Section 4.1.2. Then we introduce *Talky-G*.

4.1.1 Vertical itemset mining

Miners from the literature, whether for plain FIs or FCIs, can be roughly split into breadth-first and depth-first ones. Breadth-first algorithms, more specifically the *Apriori*-like [1] ones, apply levelwise traversal of the pattern space exploiting the anti-monotony of the frequent status. Depth-first algorithms, e.g., *Closet* [18], in contrast, organize the search space into a prefix-tree (see Fig. 2) thus factoring out the effort to process common prefixes of itemsets. Among them, the *vertical* miners use an encoding of the dataset as a set of pairs (item, tidset), i.e., $\{(i, t(i))|i \in \mathcal{A}\}$, which reportedly allows the costly database re-scans to be avoided.

Eclat [19] was the first FI-miner to combine the vertical encoding with a depth-first traversal of a tree structure, called IT-tree, whose nodes are $X \times t(X)$ pairs. *Eclat* traverses the IT-tree in a depth-first manner in a pre-order way, from left-to-right [19, 20] (see Fig. 2). *Charm* adapts the computing schema of *Eclat* to the exclusive construction of the FCIs [7]. The key challenges it faces are parsimony in generating the closedness candidates and efficiency of closedness tests on those candidates. To avoid examining the entire IT-tree of the FIs, *Charm* relies on a technique that, given a node $X \times t(X)$, looks for a Z subsuming X by combining X to Y, where $Y \times t(Y)$ is any right sibling node in the tree. Due to the specific traversal discipline, all Z are such that X is a prefix thereof (hence not all X expand to the closure of $[X]$).

To certify a candidate Z as closed, it should be checked that no set can subsume Z. Again, the traversal ensures that potential subsumers can only precede Z in the

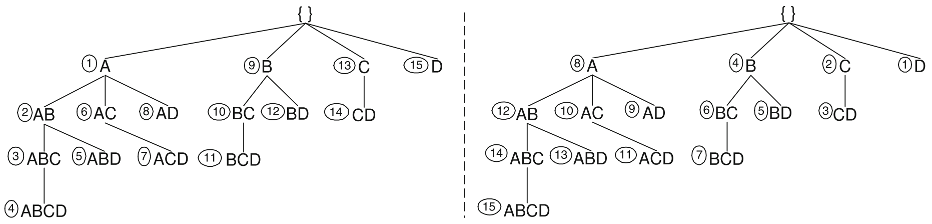


Fig. 2 *Left*: pre-order traversal with *Eclat*; *Right*: reverse pre-order traversal with *Talky-G*

traversal-induced order on $\wp(\mathcal{A})$, hence at the moment Z is tested all of them are already processed and the actual closure is known. Thus, the closedness test amounts to a lookup in the working memory for a set Y such that $t(Z) = t(Y)$, absence meaning that Z is the closure of $[Z]$. To avoid extensive search through the known part of the FCI family, *Charm* employs a hashing on $t(Z)$ (hashing is discussed in Section 4.1.4).

Zaki and Gouda also proposed an efficient approach to reduce memory usage [21]. Instead of storing the tidset of a k-itemset P in a node, the difference between the tidset of P and the tidset of the $(k-1)$-prefix of P is stored, denoted by the *diffset* of P. Note that we used this optimization technique in our algorithm *Snow-Touch*, but for an easier understanding, we skip its detailed explanation in the rest of the paper.

Charm is knowingly one of the fastest FCI-miners (together with LCM [22, 23]), hence we adopt it in our method. Other notable FCI-miners are *CLOSET+* [24] and *DBV-Miner* [25]. Looking for a similar strategy for FG computation, we observe that image-based comparisons that straightforwardly discard non-closed itemsets in *Charm* will not work in the same way with FGs. Indeed, the mere fact that for a given candidate X an FG with the same image already exists does not disqualify X (as there may be several generators in $[X]$). The test must be completed by set inclusion, i.e. X will be certified a generator if no known generator with the same image is also a subset of X.

Moreover, hidden in the above testing principles is a different traversal order: in fact, for the test to be effective, all subsets of a candidate X must be processed *before* X itself. Only then all generator subsets of X will be available for a thorough test of X being generator itself. Although such a concern is typically addressed through a breadth-first traversal strategy in mining, the same order could also be achieved with a depth-first one, yet with a different order on the items.

4.1.2 Reverse pre-order traversal

Traversing the search space so that a given set X is processed after all its subsets is a frequent requirement in combinatorial algorithms. Levelwise methods straightforwardly satisfy this condition. This strategy is used in *Next-Closure* [4] under the name of *lectic* order (or lexicographical order): it results from a mapping of $\wp(\mathcal{A})$ on $[0 \dots 2^{|\mathcal{A}|} - 1]$, where a set image is the decimal value of its characteristic vector. The mapping of the singleton sets corresponds to a fixed ordering on $\mathcal{A}$.

Then, the sets are listed in the increasing order of their mapping values which amounts to a depth-first traversal of $\wp(\mathcal{A})$. The traversal in *Charm* is similar, yet in *Next-Closure* it only depends on the precedence between CI in the lectic order, whereas *Charm* would explore the edges in the IT-tree to calculate the support and the closure of an itemset.

Following an idea in [26], called *reverse pre-order traversal*, we rank items in the initial ordering in reverse lexicographic order (E, D, C, etc.). Thus, following the increasing order of numerical equivalents, we get a depth-first right-to-left traversal of a prefix-tree representing the search space $\wp(\mathcal{A})$. As at all nodes corresponding sets are listed before the sets corresponding to descendant nodes, the processing is “pre” (rather than “post”).

In summary, our method traverses the IT-tree in a pre-order way from right-to-left. Thus, given an itemset X in a node in the IT-tree, it is guaranteed that the nodes corresponding to the subsets of X will be explored *before* X.

Algorithm 1 (main block of Talky-G)

```
 1)  root.itemset ← ∅;  // root's itemset is empty
 2)  // the empty set is included in every transaction:
 3)  root.tidset ← {all transaction IDs};
 4)  loop over the vertical representation of the dataset (attr) {
 5)     if (($attr$.supp ≥ $min\_supp$) and ($attr$.supp < $|\mathcal{O}|$)) {  // frequent and generator
 6)        root.addChild(attr);  // attr is an FG
 7)     }
 8)  }
 9)  loop over children of root from right-to-left (child) {
10)     save(child);  // process the itemset
11)     extend(child);  // discover the subtree below child
12)  }
```

Example See Fig. 2 for a comparison between the two traversals namely pre-order with *Eclat* (left) and reverse pre-order with *Talky-G* (right). The direction of traversal is indicated in circles.

4.1.3 Talky-G

Talky-G is a vertical FG miner that constructs an IT-tree in a depth-first manner in a reverse pre-order way.

Algorithm 1 provides the main block of *Talky-G*. First, the IT-tree is initialized, which includes the following steps: the root node, representing the empty set, is created. By definition, the support of the empty set is equal to the number of transactions in the dataset (100 %). *Talky-G* transforms the layout of the dataset in vertical format, and inserts under the root node all 1-long frequent generators.[1] After this, the `extend` procedure is called recursively for each child of the root in a reverse pre-order way from right-to-left. *Talky-G* concentrates on frequent generators only so that at the end all FGs are comprised in the IT-tree.

The `addChild` procedure inserts an IT-node under a node. The `save` procedure stores a frequent generator in a dedicated "list" data structure.[2] The `extend` procedure (see Algorithm 2) discovers all frequent generators in the subtree of a node. First, the procedure forms new frequent generators with the right siblings of the current node. Then, these FGs are added below the current node and are extended recursively in a reverse pre-order way from right-to-left.

The `getNextGenerator` function (see Algorithm 3) has two nodes as input parameters, and it returns a new frequent generator, or null if no frequent generator can be produced from the two input nodes. A candidate node is created by taking the union of the itemsets of the two input nodes, and the intersection of their tidsets. Thus, the input nodes are the *parents* of the candidate. Then, the candidate undergoes a series of tests. *First*, a frequency test is used to eliminate non-frequent itemsets.

[1] Recall that a 1-long itemset p is generator iff $supp(p) < 100\ \%$.

[2] We call this data structure a "list" at this point to facilitate comprehension. Full details of this data structure, which is actually a hash, are given in the next subsection.

Algorithm 2 (extend procedure of Talky-G)

```
Method:  extend an IT-node recursively (discover FGs in its subtree)
Input:   curr – an IT-node

1)  loop over siblings of curr from left-to-right (other) {
2)      gen ← getNextGenerator(curr, other);
3)      if (gen ≠ null) then curr.addChild(gen);
4)  }
5)  loop over children of curr from right-to-left (child) {
6)      save(child);  // process the itemset
7)      extend(child);  // discover the subtree below child
8)  }
```

Second, the candidate is compared to its parents. If its tidset (size) is equivalent to the tidset (size) of one of its parents, then the candidate is not a generator (by Definition 1). An itemset that survived these two tests may still not be a generator. As seen before, the reverse pre-order traversal *guarantees* that when an itemset is reached in the IT-tree, all its subsets are handled before. *Talky-G* collects frequent generators in a "list" too (see also the `save` procedure). The *third* test checks if the candidate has a proper subset with the same support in this "list" whereby a positive outcome disqualifies the candidate (Definition 1). If a candidate survives all the tests, then it is declared an FG and added to the IT-tree. Otherwise it is discarded.

Algorithm 3 (getNextGenerator function)

```
Method:  create a new frequent generator
Input:   two IT-nodes (curr and other)
Output:  a frequent generator or null

 1)  cand.tidset ← curr.tidset ∩ other.tidset;
 2)  if (cardinality(cand.tidset) < min_supp) { // test 1
 3)      return null;  // not frequent
 4)  }
 5)  // else, if it is frequent
 6)  if ((cand.tidset = curr.tidset) or (cand.tidset = other.tidset)) {  // test 2
 7)      return null;  // not a generator
 8)  }
 9)  // else, if it is a potential generator
10)  cand.itemset ← curr.itemset ∪ other.itemset;
11)  if (cand has a proper subset with the same support in the hash) {  // test 3
12)      return null;  // not a generator
13)  }
14)  // if cand passed all the tests then cand is an FG
15)  return cand;
```

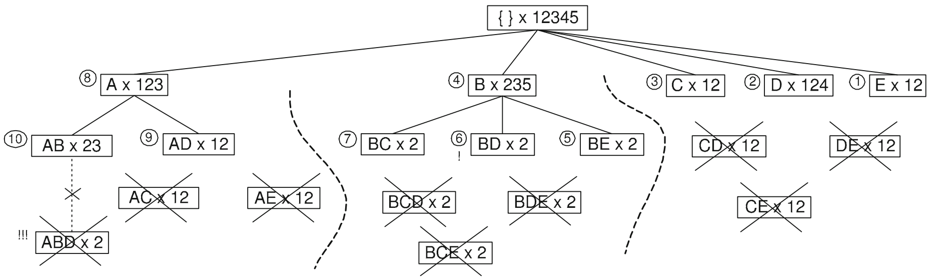


Fig. 3 Execution of *Talky-G* on dataset $\mathcal{D}$ with *min_supp* = 1 (20 %)

When the algorithm stops, all frequent generators (and *only* frequent generators) are inserted in the IT-tree *and* in the "list" of generators.

Example The execution of *Talky-G* on dataset $\mathcal{D}$ with *min_supp* = 1 (20 %) is illustrated in Fig. 3. The execution order is indicated on the left side of the nodes in circles. The algorithm first initializes the IT-tree with the root node. Since there is no full column in the input dataset, all attributes are frequent generators, thus they are added under the root. The children of the root node are examined, from *right-to-left*, each time tentatively extending them (in a recursive manner). First, node E has no right sibling, thus it cannot be extended. Node D is extended with E, but the resulting itemset DE cannot be a generator since it has the same support as its parent E. Node C is extended with D and E, but both CD and CE are discarded (for the same reason). We skip the detailed presentation of the subtree of B. The supersets of A of size two are formed (by using its right siblings) and then added to the IT-tree. Among them AC and AE are discarded because of C and E, respectively. The combination of AB and AD produces the candidate ABD. While the support of ABD is different from the supports of its parents, a proper subset thereof of the same support (BD) has been found *in a previous branch*. Hence ABD cannot be a generator (by Definition 1).

4.1.4 Fast subsumption checking

Recall that by Definition 1, a subsumer itemset cannot be a generator. Moreover, in the `getNextGenerator` function, when a new candidate itemset C is created, *Talky-G* checks whether C subsumes a previously found generator. If the test is positive, then clearly C is not a generator. This subsumption test might seem expensive, yet an efficient way to carry it out was found.

In *Talky-G* we adapted the hash structure of *Charm* for storing frequent generators together with their support values. In particular, the hash function is computed from the sum of the tids in the tidset. Thus, two equivalent itemsets get the same hash value and hence they end up in the same list (corresponding to the hash value) in the global hash structure.

Let $h(X_i)$ denote the hash function on the tidset of X_i. For the subsumption check of a candidate itemset C, we retrieve from the hash table all entries with the hash key $h(C)$. For each element G in this list, we test if $supp(C) = supp(G)$. If equality, then we test if $C \supset G$. If again positive, then C subsumes G, i.e. C is not a generator.

If C subsumes no entries in the list, then C is a generator and is therefore added to the list.

Example Figure 4 (top right) depicts the hash structure of the IT-tree in Fig. 3. The size of the hash table is set to four here. Each entry of the table is a list of itemsets with the same hash key. The table contains the FGs of dataset $\mathcal{D}$. To test the generator status of ABD, first we establish its hash value. To that end, we compute the sum of the tids in its tidset, then modulo this sum by the size of the hash table: 2 mod 4 = 2. The list at position 2 of the hash table is examined: the support of B differs from the support of ABD. Next, BE has the same support as ABD, but is not a proper subset thereof. For BD, the support is the same as ABD, and ABD is a proper superset of BD. Thus ABD cannot be a generator, so it is discarded and the search is interrupted.

4.2 Associating closed itemsets and generators to form equivalence classes: the Touch algorithm

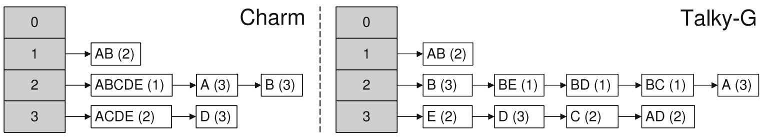
The *Touch* algorithm has three main features, namely (**1**) extracting frequent closed itemsets, (**2**) extracting frequent generators, and (**3**) associating frequent generators to their closures, i.e. identifying frequent equivalence classes.

Previously, we showed that *Charm* and *Talky-G* extract FCIs and FGs, respectively. Each algorithm uses a dedicated hash data structure for storing the found itemsets (see Fig. 4 (top)). There is one more step to do, the efficient association of frequent generators to their closures.

Our approach exploits the hash structures for the association. It is based on the following property:

Property 2 Let $h(X)$ denote the hash function on the tidset of X. Given a frequent closed itemset Y and its frequent generator Z, $h(Y) = h(Z)$ since $t(Y) = t(Z)$.

Since the two hash tables have the *same size*, and the two algorithms use the *same hash function*, it follows from Property 2 that a frequent closed itemset and its generators are in different hash tables but at the *same index position*.



FCI (supp)	FGs
AB (2)	AB
$ABCDE$ (1)	BE; BD; BC
A (3)	A

FCI (supp)	FGs
B (3)	B
$ACDE$ (2)	E; C; AD
D (3)	D

Fig. 4 *Top*: hash tables for dataset $\mathcal{D}$ with *min_supp* = 1. *Top left*: hash table of *Charm* containing all FCIs. *Top right*: hash table of *Talky-G* containing all FGs. *Bottom*: output of *Touch* on dataset $\mathcal{D}$ with *min_supp* = 1

Pseudo code First, the algorithm calls *Charm* and *Talky-G* and takes over their hash structures. Then, *Touch* matches both hash tables: For each closed X, it looks up in the FG table at the same position all subsets of X that have the same support.

Example Figure 4 (top) depicts the hash structures of *Charm* and *Talky-G*. Assume we want to determine the generators of $ACDE$ which is stored at position 3 in the hash structure of *Charm*. By Property 2, its generators are also stored at position 3 in the hash structure of *Talky-G*. The list comprises three members that are subsets of $ACDE$ with the same support: E, C, and AD. Hence, these are the generators of $ACDE$. The closed itemset A has one subset with the same support in the other hash structure at position 2 (A). This means that A is both closed *and* generator, i.e. $[A]$ is a singleton. The output of *Touch* is shown in Fig. 4 (bottom).

4.3 Computing precedence links between equivalence classes: the Snow algorithm

Snow computes precedence links on FCIs from associated FGs by using hypergraph theory. Thus, first we recall basic concepts of hypergraphs and then we introduce the *Snow* algorithm.

4.3.1 Hypergraphs

We recall that a hypergraph [27] is a generalization of a graph, where edges can connect arbitrary number of vertices.

Definition 2 (Hypergraph) A *hypergraph* is a pair $(V,\mathcal{E})$ of a finite set $V = \{v_1, v_2, \ldots, v_n\}$ and a family $\mathcal{E}$ of subsets of V. The elements of V are called vertices, the elements of $\mathcal{E}$ edges. A hypergraph is *simple* if no two edges compare for $\subseteq$, i.e. $\forall \mathcal{E}_i, \mathcal{E}_j : \mathcal{E}_i \subseteq \mathcal{E}_j \Rightarrow i = j$.

A set $T \subseteq V$ is called a *transversal* of hypergraph $\mathcal{H}$ if it has a non-empty intersection with all the edges of $\mathcal{H}$. The family of all minimal transversals of $\mathcal{H}$ constitutes the *transversal hypergraph* of $\mathcal{H}$, denoted by $Tr(\mathcal{H})$. A duality exists between a simple hypergraph and its transversal hypergraph [27]:

Proposition 1 (Duality) *For a simple hypergraph* $\mathcal{H}$, $Tr(Tr(\mathcal{H})) = \mathcal{H}$.

Examples Consider the hypergraph $\mathcal{H}$ in Fig. 5. (**1**) $\mathcal{H}$ is not simple since $A \subseteq ACD$. (**2**) $\mathcal{H}$ has two minimal transversals: AB and AC. (**3**) $Tr(\mathcal{H}) = \{AB, AC\}$. (**4**) Consider the following *simple* hypergraph: $\mathcal{G} = \{A, BC\}$. Then, $\mathcal{G}^* = Tr(\mathcal{G}) = Tr(\{A, BC\}) = \{AB, AC\}$, and $Tr(\mathcal{G}^*) = Tr(\{AB, AC\}) = \{A, BC\}$ (duality).

From a computational point of view, the extraction of $Tr(\mathcal{G})$ from $\mathcal{G}$ is a tough problem since $Tr(\mathcal{G})$ can be exponentially larger than $\mathcal{G}$. In fact, the exact complexity

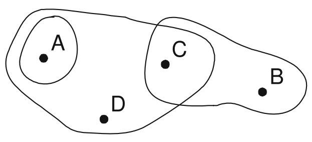


Fig. 5 A hypergraph $\mathcal{H}$, where $V = \{A, B, C, D\}$ and $\mathcal{E} = \{A, BC, ACD\}$

class is still unknown [28]. However, many algorithms for the task exist and perform well in practice.

4.3.2 Closures, precedence, and hypergraph transversals

We now define the *face* [12] and the *family of faces* of a closed itemset (CI) in a CI lattice:

Definition 3 The *face* of a CI X w.r.t. an immediate predecessor Y is the difference between those two sets. The *family of faces* is:

$$faces(X) = \{X - Y \mid X \prec Y\}$$

Example Consider the CI lattice in Fig. 1c. The CI $ABCDE$ has two faces: $F_1 = ABCDE \setminus AB = CDE$ and $F_2 = ABCDE \setminus ACDE = B$. Thus, $faces(ABCDE) = \{CDE, B\}$.

Hypergraphs naturally arise within the Hasse diagram, i.e. the CI family with precedence. Let the family of faces associated to X be $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$. Given a CI X, the associated generators compose the transversal hypergraph of its family of faces $\mathcal{F}$ seen as the hypergraph $(X, \mathcal{F})$. Clearly, $(X, \mathcal{F})$ forms a simple hypergraph. Moreover, despite language mismatch, [12] basically shows that the generators of a CI are the minimal transversals of that hypergraph.

Theorem 1 *Assume a CI X and let $\mathcal{F}$ be its family of faces. Then a set $Z \subseteq X$ is a (minimal) generator of X iff Z is a minimal transversal of $(X, \mathcal{F})$.*

Example The minimal transversals of $\{CDE, B\}$ are $\{BC, BD, BE\}$, hence these are the (minimal) generators of $ABCDE$ (see Fig. 1c).

The generator-computing method in [29] is a direct application of Theorem 1: CIs with precedence links are given, thus they yield the faces straightforwardly, which in turn generate all minimal transversals through an incremental procedure whose principles date back to [27]. Yet there are many other algorithms for transversal computation which could be used to produce the generators. Conversely, whenever scalable, concept analysis methods which output both CIs and precedence could fit the task. In contrast, beside *Charm-L*, we are not aware of other efficient miners that yield CIs with precedence without addressing transactions one by one.

As to our own approach, we assume as an input FCIs and their FGs which are relatively easier to get in an efficient manner. The computation exploits the following property which is readily deduced from Theorem 1 and Proposition 1 (yet has not been explicitly stated so far, neither explored for computation).

Property 3 Let X be a closed itemset and let $\mathcal{G}$ and $\mathcal{F}$ be the family of its generators and the family of its faces, respectively. Then, for the underlying hypergraphs it holds that $Tr(X, \mathcal{G}) = (X, \mathcal{F})$ and $Tr(X, \mathcal{F}) = (X, \mathcal{G})$.

Example Consider again the bottom concept in Fig. 1c whose CI is $ABCDE$. It has three generators: BC, BD, and BE. The transversal hypergraph of the generator family is $Tr(\{BC, BD, BE\}) = \{CDE, B\}$.

Algorithm 4 (Snow)

Description: build iceberg lattice from FCI/FG-pairs
Input: a set of FCIs with their associated FGs, i.e. frequent equivalence classes

```
1) identifyOrCreateTopCI($setOfFCIsWithFGs$);
2) // find the predecessor(s) for each equivalence class:
3) for all $c$ in $setOfFCIsWithFGs$ {
4)    $setOfFaces$ ← getMinTrs($c$.generators);
5)    $predecessorCIs$ ← getPredecessorCIs($c$.closure, $setOfFaces$);
6)    loop over the CIs in $predecessorCIs$ ($p$) {
7)       connect($c$, $p$);
8)    }
9) }
```

4.3.3 Snow

The *Snow* algorithm computes precedence links on FCIs from associated FGs by exploiting the duality with faces [14]. *Snow* exploits Property 3 by computing faces from generators. Thus, its input is made of FCIs and their associated FGs. Several algorithms can be used to produce this input, e.g. *Titanic* [6], *A-Close* [5], *Zart* [30], *Touch* (see Section 4.2), etc. Figure 4 (bottom) depicts a sample input of *Snow*.

On such data, *Snow* first computes the faces of a CI as the minimal transversals of its generator hypergraph. Next, each difference of the CI X with a face yields a predecessor of X in the closed itemset lattice.

Example Consider again $ABCDE$ with its generator family $\{BC, BD, BE\}$. First, we compute its transversal hypergraph: $Tr(\{BC, BD, BE\}) = \{CDE, B\}$. The two faces $F_1 = CDE$ and $F_2 = B$ indicate that there are two predecessors for $ABCDE$, say Z_1 and Z_2, where $Z_1 = ABCDE \setminus CDE = AB$, and $Z_2 = ABCDE \setminus B = ACDE$. Application of this procedure for all CIs yields the entire precedence relation for the CI lattice.

The pseudo code of *Snow* is given in Algorithm 4. As input, *Snow* receives a set of CIs with their associated generators. The `identifyOrCreateTopCI` procedure looks for a CI whose support is 100 %. If it does not find one, then it creates it by taking an empty set as the CI with 100 % support and a void family of generators (see Fig. 1c for an example). The `getMinTrs` function computes the transversal hypergraph of a given hypergraph. More precisely, given the family of generators of a CI X, the function returns the family of faces of X. It is noteworthy that any algorithm for transversal computation in a hypergraph would be appropriate here. In our current implementation, we use an optimized version of Berge's algorithm henceforth referred to as *BergeOpt*.[3] The `getPredecessorCIs` function calculates the differences between a CI X and the family of faces of X. The function returns the

[3]For a complete presentation of *BergeOpt*, please refer to the report [31].

set of all CIs that are predecessors of X. The `connect` procedure links the current CI to its predecessors.

Please notice that the only computationally intensive step in *Snow* is the transversal hypergraph construction. Thus, the total cost heavily depends on the efficiency of that step. We investigated the size of its input data, and we obtained similar hypergraph-size distributions in all the datasets that were used in our tests (see Table 3). Most hypergraphs only have 1 edge, which is a trivial case, whereas large hypergraphs are relatively rare. As a consequence, *BergeOpt* and thus *Snow* perform very efficiently.

5 Related work

Historically the first algorithm computing all closures with their generators and precedence links can be found in [11] (although under a different name in a somewhat incomplete manner[4]). Yet the individual tasks have been addressed separately or in various combinations by a large variety of methods.

First, the construction of all concepts is a classical FCA task and a large number of algorithms exists for it using a wide range of computing strategies. Yet they scale poorly as FCI miners due to their reliance on object-wise computations (e.g. the incremental acquisition of objects as in [11]). These involve to a large number of what is called *data scans* in data mining that are known to seriously deteriorate the performance. In fact, the overwhelming majority of FCA algorithms would suffer on the same drawback as they have been designed under the assumption that the number of objects and the number of attributes remain in the same order of magnitude. Yet in data mining, there is usually a much larger number of transactions than there are items.

As for generators, they have attracted significantly less attraction in FCA as a standalone structure. Precedence links, in turn, are sometimes computed by concept mining FCA algorithms beside the concept set. Here again, objects are heavily involved in the computation hence the poor scaling capacity of these methods. The only notable exception to this rule is the method described in [32] which was designed to deliberately avoid referring to objects by relying exclusively on concept intents.

When all three structures are considered together, after [11], efficient methods for the combined task have been proposed, among others, in [12, 13].

In data mining, mining FCIs is also a popular task [33]. Many FCI miners exist and a good proportion thereof would output FGs as a byproduct. For instance, levelwise miners such as *Titanic* [6] and *A-Close* [33] use generators as entry points into the equivalence classes of their respective FCIs. In this field, the FGs, under the name of free-sets [17], have been targeted by dedicated miners [34]. Precedence links does not seem to play a major role in pattern mining since few miners would consider them. In fact, to the best of our knowledge, the only mainstream FCI miner that would also output the Hasse diagram of the iceberg lattice is *Charm-L* [9]. In order to avoid multiple data scans, *Charm-L* relies on a specific encoding of the transaction

[4]While Algorithm 4.3 in the text formally outputs a set of implication rules on top of the Hasse diagram of the lattice, the non-closed generators can be retrieved as the left-hand sides of those rules while the closed ones … appear in the lattice.

database, called *vertical*, that takes advantage of the aforementioned asymmetry between the number of transactions and the number of items. Moreover, two ulterior extensions thereof [8, 10] would also cover the FGs for each FCI, making them the primary competitors for our own approach.

Despite the clear discrepancies in their *modus operandi*, both FCA-centered algorithms and FCI/FG miners share their overall algorithmic schema. Indeed, they first compute the set of concepts/FCIs and the precedence links between them and then use these as input in generator/FG calculation. The latter task can be either performed along a levelwise traversal of the equivalence class of a given closure, as in [9, 11], or shaped as the computation of the minimal transversals of a dedicated hypergraph,[5] as in [10, 12, 13].

6 Experimental evaluation

In this section we present the results of a series of tests. First, we provide results that we obtained on a real-life biological dataset. Then, we compare the performances of *Snow-Touch* and *Charm-L*. We demonstrate that our approach is computationally efficient. Thus, a series of computational times resulting from the application of our algorithm to well-known datasets is presented.

6.1 Analysis of antibiotic resistant genes

We looked at the practical performance of *Snow-Touch* on a real-world genomic dataset whereby the goal was to discover meaningful associations between genes in entire genomes seen as items and transactions, respectively.

The genomic dataset was collected from the website of the National Center for Biotechnology Information (NCBI) with a focus on genes from microbial genomes. At the time of writing, 1,518 complete microbial genomes were available on the NCBI website.[6] For each genome, its list of genes was collected. Only 1,250 genomes out of the 1,518 proved to be non-empty; we put them in a binary matrix of 1,250 rows $\times$ 125,139 columns. With an average of 684 genes per genome, we got 0.55 % density (i.e., large yet sparse dataset with an imbalance between the numbers of rows and of columns).

The initial result of the data mining task was the family of *minimal non-redundant association rules* ($\mathcal{MNR}$), which are directly available from the output of *Snow-Touch*. We sorted them according to the confidence. Among all strong associations, the bioinformaticians involved in this study found most appealing the rules describing the behavior of antibiotic resistant genes (e.g., the gene *mecA*). *MecA* is frequently found in bacterial cells. It induces a resistance to antibiotics such as *Methicillin*, *Penicillin*, *Erythromycin*, etc. [35]. One of the most commonly known carrier of the gene *mecA* is the bacterium known as MRSA (methicillin-resistant *Staphylococcus aureus*).

At a second step, we narrowed our focus and considered a group of three genes: *mecA*, *ampC* and *vanA* [36]. *AmpC* is a beta-lactam-resistance gene. *AmpC*

[5]Termed alternatively as *blockers* or *hitting sets*.

[6]http://www.ncbi.nlm.nih.gov/genomes/lproks.cgi

beta-lactamases are typically encoded on the chromosome of many gram-negative bacteria; it may also be present in *Escherichia coli*. *AmpC* type beta-lactamases may also be carried on plasmids [35]. Finally, the gene *vanA* is a vancomycin-resistance gene typically encoded on the chromosome of gram-positive bacteria such as *Enterococcus*. The idea was to relate the presence of these three genes to the presence or absence of any other gene or a combination thereof.

Table 1 shows an extract of the most interesting rules found by our algorithm. These rules were selected from a set of 18,786 rules.

For instance, the rule (1) in Table 1 says that the gene *mecA* is present in 85.71 % of cases when the set of genes {*clpX*, *dnaA*, *dnaI*, *dnaK*, *gyrB*, *hrcA*, *pyrF*} is present in a genome.

Antibiotic resistant genes are often acquired by bacteria via the process of horizontal gene transfer (HGT) [37]. All the obtained rules presented in Table 1 suggest the most probable cases, which physicians should study in more detail, when an antibiotic resistant gene can be accepted by a bacterium during (or after) the antibiotic treatment. Thus, these rules can be used to decide which antibiotic should be taken by a patient depending on the presence of certain genes (located in the left part of the rules).

At a third step of our experimental study, we were interested in negative associations between antibiotic resistant genes and all other genes. To do that, we needed to invert our dataset, i.e. 1 s became 0 s and vice versa in the matrix, except for the columns of the three antibiotic genes. This resulted in rules where the antecedent is made of absent genes, while the consequent contains the above-mentioned antibiotic resistant genes.

The inverted dataset was so dense (99.45 %) that $\mathcal{MNR}$ rules could not be extracted from it directly. Thus, we created three smaller views of the dataset, a view for each of the three genes. A view contained such genomes that included the

Table 1 An extract of the generated minimal non-redundant association rules

(1)	{*clpX*, *dnaA*, *dnaI*, *dnaK*, *gyrB*, *hrcA*, *pyrF*} → {*mecA*} (supp=96 [7.68 %]; **conf=0.857 [85.71 %]**; suppL=112 [8.96 %]; suppR=101 [8.08 %])
(2)	{*clpX*, *dnaA*, *dnaI*, *dnaK*, *nusG*} → {*mecA*} (supp=96 [7.68 %]; **conf=0.835 [83.48 %]**; suppL=115 [9.20 %]; suppR=101 [8.08 %])
(3)	{*clpX*, *dnaA*, *dnaI*, *dnaJ*, *dnaK*} → {*mecA*} (supp=96 [7.68 %]; **conf=0.828 [82.76 %]**; suppL=116 [9.28 %]; suppR=101 [8.08 %])
(4)	{*clpX*, *dnaA*, *dnaI*, *dnaK*, *ftsZ*} → {*mecA*} (supp=96 [7.68 %]; **conf=0.828 [82.76 %]**; suppL=116 [9.28 %]; suppR=101 [8.08 %])
(5)	{*clpX*, *dnaA*, *dnaI*, *dnaK*} → {*mecA*} (supp=97 [7.76 %]; **conf=0.815 [81.51 %]**; suppL=119 [9.52 %]; suppR=101 [8.08 %])
(6)	{*greA*, *murC*, *pheS*, *rnhB*, *ruvA*} → {*ampC*} (supp=99 [7.92 %]; **conf=0.227 [22.71 %]**; suppL=436 [34.88 %]; suppR=105 [8.40 %])
(7)	{*murC*, *pheS*, *pyrB*, *rnhB*, *ruvA*} → {*ampC*} (supp=99 [7.92 %]; **conf=0.221 [22.15 %]**; suppL=447 [35.76 %]; suppR=105 [8.40 %])
(8)	{*dxs*, *hemA*} → {*vanA*} (supp=29 [2.32 %]; **conf=0.081 [8.15 %]**; suppL=356 [28.48 %]; suppR=30 [2.40 %])
(9)	{*dxs*} → {*vanA*} (supp=30 [2.40 %]; **conf=0.067 [6.73 %]**; suppL=446 [35.68 %]; suppR=30 [2.40 %])

After each rule, the following measures are indicated: support, confidence, support of the left-hand side (antecedent) and support of the right-hand side (consequent)

Table 2 An extract of the generated *negative* minimal non-redundant association rules

(1)	{¬*ptsI*} → {*ampC*} (supp=105 [8.40 %]; **conf=0.084 [8.41 %]**; suppL=1249 [99.92 %]; suppR=105 [8.40 %])
(2)	{¬*metG*} → {*mecA*} (supp=70 [5.60 %]; **conf=0.092 [9.20 %]**; suppL=761 [60.88 %]; suppR=101 [8.08 %];)
(3)	{¬*ptsI*} → {*vanA*} (supp=30 [2.40 %]; **conf=0.024 [2.40 %]**; suppL=1249 [99.92 %]; suppR=30 [2.40 %];)

After each rule, the following measures are indicated: support, confidence, support of the left-hand side (antecedent) and support of the right-hand side (consequent)

given antibiotic gene. The view with *mecA* contained 101 genomes, the view with *ampC* had 105 genomes, while the view with *vanA* contained 30 genomes. However, these views were still very large with 125,139 attributes (genes). Thus, only the most frequent genes were kept in the views, whereas the remainder were removed. This process of reducing the number of genes was repeated until a view had about 1,000 genes only. At that point, $\mathcal{MNR}$ rules were extracted from the views. Table 2 shows the most confident negative rules, one rule per view. After analyzing these rules (see Table 2), we concluded that there are no genes whose absence triggers the acquisition of antibiotic resistant genes *mecA*, *ampC* and *vanA*.

Two other interesting bioinformatics applications of the described algorithm can be also considered in the future. The first issue that could be addressed is that of how the presence or absence of certain individual genes, proteins, polymorphisms (i.e., parts of these genes or proteins) or their combinations in a species (e.g., *Homo sapiens*) genome can lead to the development of a specific disease or to the resistance to this disease. The second issue that could be treated using the presented algorithm concerns the investigation of how the presence or absence of certain individual proteins or their combinations, which are related to certain biological functions, can influence the presence or absence of other proteins expressing different biological functions. Such a case study would shed light on the main existing functional dependencies, not necessarily related to diseases, and help infer a functional dependency map characterizing the species under study [38].

6.2 Snow-Touch vs. Charm-L

We evaluated *Snow-Touch* against *Charm-L* [9, 10]. The experiments were carried out on a bi-processor Intel Quad Core Xeon 2.33 GHz machine running Ubuntu GNU/Linux with 4 GB RAM. All times reported are real, wall clock times, as obtained from the Unix *time* command between input and output. *Snow-Touch* was implemented entirely in Java. For performance comparisons, the authors' original C++ source of *Charm-L* was used. *Charm-L* and *Snow-Touch* were executed with these options: `./charm-l -i input -s min_supp -x -L -o COUT -M 0 -n`; `./leco.sh input min_supp -order -alg:dtouch -method:snow -nof2`. In each case, the standard output was redirected to a file. The diffset optimization technique [21] was activated in both algorithms.[7]

[7] *Charm-L* uses diffsets by default, thus no explicit parameter was required.

Table 3 Database characteristics

Database name	# records	# non-empty attributes	# attributes (in average)	Largest attribute
T20I6D100K	100,000	893	20	1,000
T25I10D10K	10,000	929	25	1,000
C20D10K	10,000	192	20	385
C73D10K	10,000	1,592	73	2,177
Mushrooms	8,416	119	23	128
Chess	3,196	75	37	75
Connect	67,557	129	43	129

Benchmark datasets For the experiments, we used several real and synthetic dataset benchmarks (see Table 3). The synthetic datasets T20 and T25, using the IBM Almaden generator, are constructed according to the properties of market basket data. The C20 and C73 datasets contain census data from the PUMS sample file. The Mushrooms database describes mushrooms characteristics. The Chess and Connect datasets are derived from their respective game steps. The latter three datasets can be found in the UC Irvine Machine Learning Database Repository. Typically, real datasets are usually more dense than synthetic data. Response times of the two algorithms on these datasets are presented in Fig. 6.

Charm-L *Charm-L* represents a state-of-the-art algorithm for closed itemset lattice construction [9]. *Charm-L* extends *Charm* to directly compute the lattice while it generates the CIs. In the experiments, we executed *Charm-L* with a switch to compute (minimal) generators too using the *minhitset* method. In [10], Zaki and Ramakrishnan present an efficient method for calculating the generators, which is actually the generator-computing method of Pfaltz and Taylor [29] (see also Theorem 1). This way, the two algorithms (*Snow-Touch* and *Charm-L*) are *comparable* since they produce *exactly the same output*.

Performance on sparse datasets On T20 and T25, *Charm-L* performs better than *Snow-Touch* (see Fig. 6). The reason is that T20 and T25 produces long sparse bitvectors, which gives some overhead to *Snow-Touch*. In our implementation, we use bitvectors to store tidsets. However, as can be seen in the next paragraph, our algorithm outperforms *Charm-L* on all the dense datasets that were used during our tests.

Performance on dense datasets On C20, C73, Mushrooms, Chess and Connect, we can observe that *Charm-L* performs well only for high values of support (see Fig. 6). Below a certain threshold, *Snow-Touch* gives lower response times, and the gap widens as the support is lowered. When the minimum support is set low enough, *Snow-Touch* can be several times faster than *Charm-L*. Considering that *Snow-Touch* is implemented in Java, we believe that a good C++ implementation could perform even better.

Performance of Snow While Fig. 6 indicates the complete execution times of *Snow-Touch* between input and output, in Table 4 we collected data about the performance of the *Snow* module only. The first column specifies the various minimum support

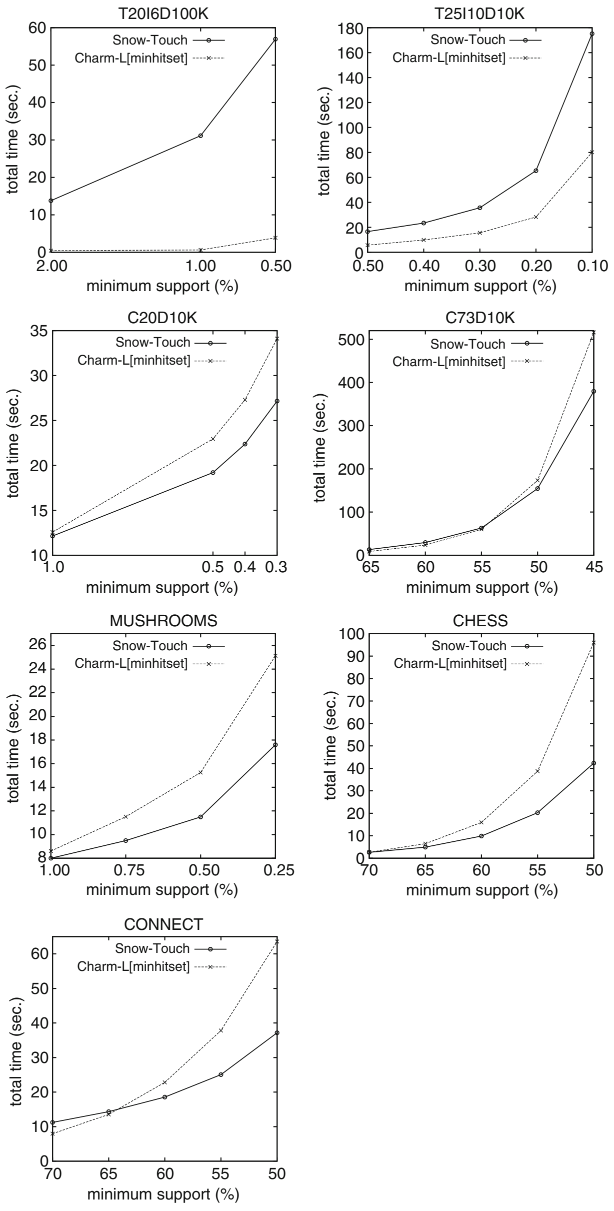


Fig. 6 Response times of *Snow-Touch* and *Charm-L*

values for each of the datasets (low for sparse datasets, higher for dense ones). The second and third columns comprise the number of FCIs and the execution time of *Snow* for finding the precedence order between the FCIs (given in seconds). As can be seen, *Snow* is able to discover the order very efficiently in both sparse and dense datasets. To explain the reason for that, recall that the only computationally intensive step in *Snow* is the transversal hypergraph construction. Thus, the total cost heavily depends on the efficiency of that step. Furthermore, to find out why the underlying

Table 4 Response times of *Snow*

min_supp (%)	# concepts (including top)	*Snow* (finding order)
T20I6D100K		
0.75	4,711	0.11
0.50	26,209	0.36
0.25	149,218	3.24
T25I10D10K		
0.40	83,063	1.07
0.30	122,582	2.73
0.20	184,301	4.48
C20D10K		
0.50	132,952	3.04
0.40	151,394	4.37
0.30	177,195	4.29
C73D10K		
65	47,491	1.51
60	108,428	3.97
55	222,253	10.13
Mushrooms		
20	1,169	0.05
10	4,850	0.17
5	12,789	0.47
Chess		
65	49,241	0.85
60	98,393	1.77
55	192,864	3.95
Connect		
65	49,707	0.54
60	68,350	0.78
55	94,917	1.82

algorithm *BergeOpt* (see Section 4.3.3) performs so well, we investigated the size of its input data. Figure 7 shows the distribution of hypergraph sizes in the datasets T20I6D100K, Mushrooms, chess, and C20D10K.[8] Note that we obtained similar hypergraph-size distributions in the other three datasets too. Figure 7 indicates that most hypergraphs only have 1 edge, which is a trivial case, whereas large hypergraphs are relatively rare. As a consequence, *BergeOpt* and thus *Snow* perform very efficiently.

According to our experiments, *Snow-Touch* can construct the concept lattices faster than *Charm-L* in the case of dense datasets. From this, we draw the hypothesis that our direction towards the construction of FG-decorated concept lattices is more beneficial than the direction of *Charm-L*. That is, it is better to extract first the FCI/FG-pairs and then determine the order relation between them than first extracting the set of FCIs, constructing the order between them, and then determining the corresponding FGs for each FCI.

[8]For instance, the dataset T20I6D100K by $min\_supp = 0.25$ % contains 149,019 1-edged hypergraphs, 171 2-edged hypergraphs, 25 3-edged hypergraphs, 0 4-edged hypergraphs, 1 5-edged hypergraph, and 1 6-edged hypergraph.

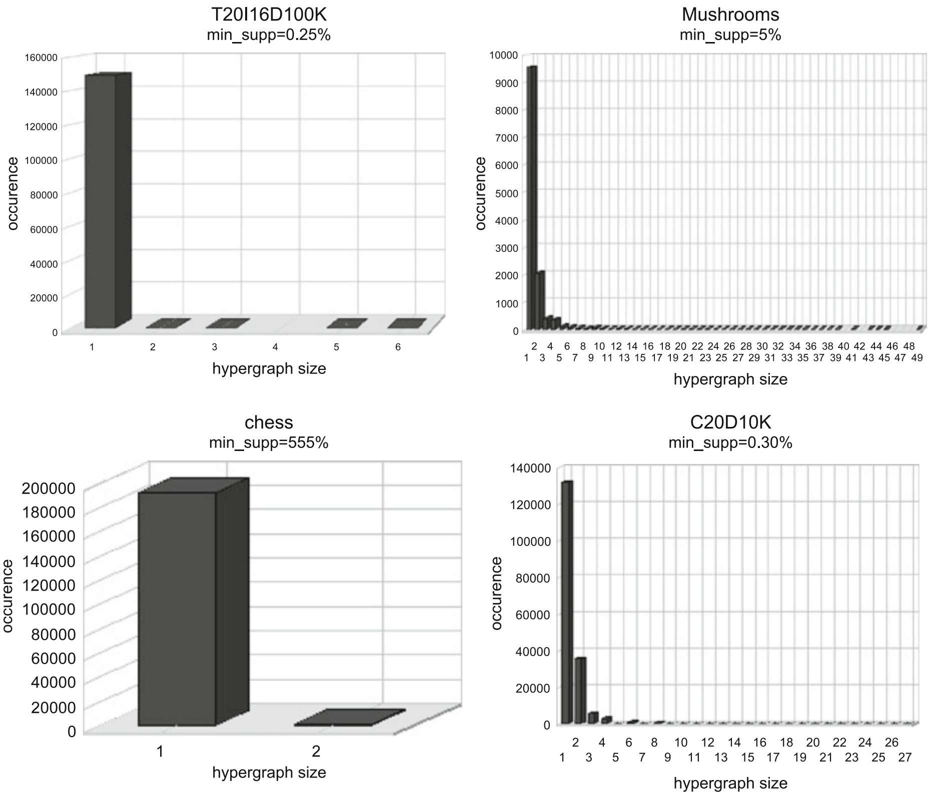


Fig. 7 Distribution of hypergraph sizes

7 Conclusion

We presented a new design schema for the task of mining the iceberg lattice and the corresponding generators out of a large context. The target structure directly involved in the construction of a number of association rule bases and hence is of a certain importance in the data mining field. While previously published algorithms follow the same schema, i.e., construction of the iceberg lattice (FCIs plus precedence links) followed by the extraction of the FGs, our approach consists in inferring precedence links from the previously mined FCIs with their FGs.

We presented an initial and straightforward instantiation of the new algorithmic schema that reuses one existing method for the first steps, i.e. the popular *Charm* FCI miner. The following steps are carried out by two novel methods, the first one for FG extraction, *Talky-G*, and the second one, for precedence, *Snow*. *Charm* and *Talky-G* are further assembled into a unique vertical FCI/FG miner, *Touch*, by means of an FGs-to-FCIs matching procedure. The resulting iceberg plus FGs miner, *Snow-Touch*, is far from an optimal algorithm, in particular due to redundancies in the first two steps. Nevertheless, an implementation thereof within the Coron platform (in Java) has managed to outperform its natural competitor, *Charm-L* (in C++) on a wide range of datasets, especially on dense ones.

In a different vein, we have tested the capacity of our approach to support practical mining task by applying it to the analysis of genomic data. While a large number of associations usually come out of such datasets, many of them are redundant with respect to each other, by limiting the output to only the generic ones, our method helped focus the analysts' attention to a smaller number of significant rules.

As an improvement, we plan to re-implement *Snow-Touch* in C++ and expect the new version to be even more efficient. As a next step, we are studying a more integrated approach for FCI/FG construction that requires no extra matching step. This should result in substantial efficiency gains. On the methodological side, our study underlines the duality between generators and order w.r.t. FCIs: either can be used in combination with FCIs to yield the other one. It rises the natural question of whether FCIs alone, which are output by a range of frequent pattern miners, could be used to efficiently retrieve first precedence, and then FGs.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. of the 20th Intl. Conf. on Very Large Data Bases (VLDB '94), pp. 487–499. Morgan Kaufmann, San Francisco, CA (1994)
2. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining minimal non-redundant association rules using frequent closed itemsets. In: Proc. of the Computational Logic (CL '00). LNAI, vol. 1861, pp. 972–986. Springer (2000)
3. Kryszkiewicz, M.: Concise representations of association rules. In: Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery, pp. 92–109 (2002)
4. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Berlin/Heidelberg (1999)
5. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Proc. of the 7th Intl. Conf. on Database Theory (ICDT '99), pp. 398–416. Jerusalem, Israel (1999)
6. Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., Lakhal, L.: Computing iceberg concept lattices with Titanic. Data Knowl. Eng. **42**(2), 189–222 (2002)
7. Zaki, M.J., Hsiao, C.J.: CHARM: an efficient algorithm for closed itemset mining. In: SIAM Intl. Conf. on Data Mining (SDM' 02), pp. 33–43 (2002)
8. Zaki, M.J.: Mining non-redundant association rules. Data Min. Knowl. Disc. **9**(3), 223–248 (2004)
9. Zaki, M.J., Hsiao, C.J.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Trans. Knowl. Data Eng. **17**(4), 462–478 (2005)
10. Zaki, M.J., Ramakrishnan, N.: Reasoning about sets using redescription mining. In: Proc. of the 11th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD '05), pp. 364–373. Chicago, IL, USA (2005)
11. Godin, R., Missaoui, R.: An incremental concept formation approach for learning from databases. Theor. Comput. Sci. **133**, 387–419 (1994)
12. Pfaltz, J.L.: Incremental transformation of lattices: a key to effective knowledge discovery. In: Proc. of the 1st Intl. Conf. on Graph Transformation (ICGT '02), pp. 351–362. Barcelona, Spain (2002)
13. Le Floc'h, A., Fisette, C., Missaoui, R., Valtchev, P., Godin, R.: JEN: un algorithme efficace de construction de générateurs pour l'identification des règles d'association. Spec. num. Revue des Nouvelles Technologies de l'Information **1**(1), 135–146 (2003)
14. Szathmary, L., Valtchev, P., Napoli, A., Godin, R.: Constructing iceberg lattices from frequent closures using generators. In: Discovery Science. LNAI, vol. 5255, pp. 136–147. Springer, Budapest, Hungary (2008)
15. Szathmary, L., Valtchev, P., Napoli, A., Godin, R.: Efficient vertical mining of frequent closures and generators. In: Proc. of the 8th Intl. Symposium on Intelligent Data Analysis (IDA '09). LNCS, vol. 5772, pp. 393–404. Springer, Lyon, France (2009)
16. Boulicaut, J.F., Bykowski, A., Rigotti, C.: Free-Sets: a condensed representation of boolean data for the approximation of frequency queries. Data Min. Knowl. Disc. **7**(1), 5–22 (2003)

17. Calders, T., Rigotti, C., Boulicaut, J.F.: A survey on condensed representations for frequent sets. In: Boulicaut, J.F., Raedt, L.D., Mannila, H. (eds.) Constraint-Based Mining and Inductive Databases. Lecture Notes in Computer Science, vol. 3848, pp. 64–80. Springer (2004)
18. Pei, J., Han, J., Mao, R.: CLOSET: an efficient algorithm for mining frequent closed itemsets. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 21–30 (2000)
19. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: Proc. of the 3rd Intl. Conf. on Knowledge Discovery in Databases, pp. 283–286 (1997)
20. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
21. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD '03), pp. 326–335. ACM Press, New York, NY (2003)
22. Uno, T., Asai, T., Uchida, Y., Arimura, H.: LCM: an efficient algorithm for enumerating frequent closed item sets. In: Goethals, B., Zaki, M.J. (eds.) FIMI. CEUR Workshop Proceedings, vol. 90. CEUR-WS.org (2003)
23. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets. In: Bayardo, R.J. Jr., Goethals, B., Zaki, M.J. (eds.) FIMI. CEUR Workshop Proceedings, vol. 126. CEUR-WS.org (2004)
24. Wang, J., Han, J., Pei, J.: CLOSET+: searching for the best strategies for mining frequent closed itemsets. In: Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD '03), pp. 236–245. ACM Press (2003)
25. Vo, B., Hong, T.P., Le, B.: DBV-Miner: a dynamic bit-vector approach for fast mining frequent closed itemset. Expert Syst. Appl. **39**(8), 7196–7206 (2012)
26. Calders, T., Goethals, B.: Depth-first non-derivable itemset mining. In: Proc. of the SIAM Intl. Conf. on Data Mining (SDM '05). Newport Beach, USA (2005)
27. Berge, C.: Hypergraphs: Combinatorics of Finite Sets. North Holland, Amsterdam (1989)
28. Eiter, T., Gottlob, G.: Identifying the minimal transversals of a hypergraph and related problems. SIAM J. Comput. **24**(6), 1278–1304 (1995)
29. Pfaltz, J.L., Taylor, C.M.: Scientific knowledge discovery through iterative transformation of concept lattices. In: Proc. of the Workshop on Discrete Applied Mathematics in Conjunction with the 2nd SIAM Intl. Conf. on Data Mining, pp. 65–74. Arlington, VA, USA (2002)
30. Szathmary, L., Napoli, A., Kuznetsov, S.O.: ZART: a multifunctional itemset mining algorithm. In: Proc. of the 5th Intl. Conf. on Concept Lattices and Their Applications (CLA '07), pp. 26–37. Montpellier, France (2007)
31. Szathmary, L., Valtchev, P., Napoli, A.: Efficient mining of frequent closures with precedence links and associated generators. Research Report RR-6657, INRIA (2008)
32. Baixeries, J., Szathmary, L., Valtchev, P., Godin, R.: Yet a faster algorithm for building the hasse diagram of a galois lattice. In: Proc. of the 7th Intl. Conf. on Formal Concept Analysis (ICFCA '09). LNAI, vol. 5548, pp. 162–177. Springer, Darmstadt, Germany (2009)
33. Pasquier, N.: Mining association rules using formal concept analysis. In: Proc. of the 8th Intl. Conf. on Conceptual Structures (ICCS '00), pp. 259–264. Shaker-Verlag (2000)
34. Li, J., Li, H., Wong, L., Pei, J., Dong, G.: Minimum description length principle: generators are preferable to closed patterns, pp. 409–414. In: AAAI, AAAI Press (2006)
35. Philippon, A., Arlet, G., Jacoby, G.A.: Plasmid-determined AmpC-type β-lactamases. Antimicrob. Agents Chemother. **46**(1), 1–11 (2002)
36. Schwartz, T., Kohnen, W., Jansen, B., Obst, U.: Detection of antibiotic-resistant bacteria and their resistance genes in wastewater, surface water, and drinking water biofilms. Microbiol. Ecol. **43**(3), 325–335 (2003)
37. Boc, A., Philippe, H., Makarenkov, V.: Inferring and validating horizontal gene transfer events using bipartition dissimilarity. Syst. Biol. **59**(2), 195–211 (2010)
38. Gjuvsland, A.B., Hayes, B.J., Omholt, S.W., Carlborg, O.: Statistical epistasis is a generic feature of gene regulatory networks. Genetics **175**, 411–420 (2007)