Constraint-based Pattern Mining

Siegfried Nijssen
KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
Universiteit Leiden
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
siegfried.nijssen@cs.kuleuven.be

Albrecht Zimmermann
INSA Lyon, LIRIS CNRS UMR 5205
Bâtiment Blaise Pascal, 69621 Villeurbanne cedex, France
albrecht.zimmermann@insa-lyon.fr

October 28, 2014

Abstract

Many pattern mining systems are designed to solve one specific problem, such as frequent, closed or maximal frequent itemset mining, efficiently. Even though efficient, their specialized nature can make these systems difficult to apply in other situations than the one they were designed for. This chapter provides an overview of generic constraint-based mining systems. Constraint-based pattern mining systems are systems that with minimal effort can be programmed to find different types of patterns satisfying constraints. They achieve this genericity by providing (1) high-level languages in which programmers can easily specify constraints; (2) generic search algorithms that find patterns for any task expressed in the specification language. The development of generic systems requires an understanding of different classes of constraints. This chapter will first provide an overview of such classes constraints, followed by a discussion of search algorithms and specification languages.

Keywords: Constraints, languages, inductive databases, search algorithms

1 Introduction

A key component of a pattern mining system is the constraint that is used by the system. A frequent itemset mining system, for instance, is characterized by the use of a minimum support constraint; an association rule mining system,

similary, is identified by a minimum confidence constraint. Constraints define to a large degree which task a pattern mining system is performing.

However, the focus of many pattern mining systems on one particular type of constraint can make their use cumbersome. As an example, consider a frequent itemset mining system that one wishes to apply in a context where the utility of the items is important as well. As a basic frequent itemset mining system does not support utilities, we cannot use it directly; we either have to:

- understand the code of the frequent itemset mining algorithm to add an additional constraint to it;
- or, write a second algorithm for processing the results of the frequent itemset mining system to evaluate the additional constraint for each of the itemsets found.

Both options are cumbersome. The second option is likely to be computationally inefficient if the number of frequent itemsets is large. The first option can be efficient, provided that the programmer has a deep understanding of the code that is being modified.

These disadvantages have led researchers to develop more general systems that provide easy-to-use interfaces for specifying the constraints that the pattern mining systems need to use during the search. The development of these systems has involved several challenges:

- the identification of general classes of constraints, all of which can be processed in a generic and similar way;
- the development of languages in which constraints can be expressed, such that all expressions in the language correspond to constraints in a class of constraints supported by a system;
- the development of search algorithms that can deal with constraints in a certain class.

This chapter will provide an overview of the state-of-the-art for each of these challenges. We will first formalize the problem of constraint-based pattern mining, including a discussion of different classes of constraints. Subsequently, we will discuss the most common search algorithms for these classes of constraints. Finally, we will discuss the languages that allow for the expression of constraints in pattern mining.

2 Problem Definition

Constraint-based mining starts from the observation that many pattern mining problems can be seen as instances of the following generic problem statement:

Given

- a data language $\mathcal{L}_{\mathcal{D}}$

- a database $\mathcal{D} \subseteq 2^{\mathcal{L}_\mathcal{D}}$ with transactions
- a pattern language $\mathcal{L}_\pi$
- a constraint $\varphi : \mathcal{L}_\pi \times 2^{\mathcal{L}_\mathcal{D}} \mapsto \{0, 1\}$

Find all patterns $\pi \in \mathcal{L}_\pi$ for which $\varphi(\pi, \mathcal{D}) = 1$.

The *pattern language* typically describes the syntax of the patterns we wish to find in the data. *Constraints* typically describe the statistical, syntactical, or other requirements that we wish these patterns to satisfy on the data.

Frequent itemset mining (see chapter ...), for example, is an instance of this generic setting, with the following choices:

- the database has transactions that are subsets of a given set of items $\mathcal{I}$;
- the pattern language is the set of all subsets of $\mathcal{I}$: $\mathcal{L}_\pi = 2^\mathcal{I}$;
- the *minimum support* constraint $\varphi_{minsup}(\pi, \mathcal{D})$ is true if and only if the number of transactions of $\mathcal{D}$ that contain π is large enough, in other words, it is true if and only if:

$$|cover(\pi)| = |\{d \in \mathcal{D} | \pi \subseteq d\}| \geq \theta,$$

  where θ is a user-defined threshold.

By modifying the pattern language, the data language and the constraints, different data mining problems can be formalized. The main aim of *constraint-based pattern mining* is to build generic languages in which programmers can express pattern mining problems in terms of constraints, and to develop systems that can process statements in these languages.

2.1 Constraints

Constraints can be categorized along several dimensions, for instance:

1. which information is used when evaluating the constraint? Possibilities include that the constraint only evaluates the syntax of the pattern, that the constraint requires a database of transactions, or that the constraint requires a database with labeled transactions.
2. which properties do the constraints have? The most well-known property is that of (anti-)monotonicity, but other properties have been identified as well.

In terms of constraint-based pattern mining, combining constraints from different categories of the former dimension is typically easy, whereas this proves challenging for the latter dimension. Hence, existing work has focused on the latter dimension and we will elaborate on these constraint categories below.

Anti-monotonicity Most pattern mining algorithms assume the existence of a *coverage relation* between patterns and transactions in the data. In the case of frequent itemset mining, for example, an itemset π covers a transaction $d \in \mathcal{D}$ iff $\pi \subseteq d$; hence, the subset relation is used as coverage relation. In graph mining, the subgraph isomorphism relation may be used; in sequence mining, the subsequence relation.

A second important relation is the *generality relation*. A generality relation is essentially a partial order on the set of patterns in $\mathcal{L}_\pi$. We will denote this relationship with the symbol $\succeq$: if pattern π_1 is more general than pattern π_2, we will write $\pi_1 \succeq \pi_2$.

A generality relation $\succeq$ is *compatible* with a coverage relation if it satisfies the following property for all possible transactions d: if $\pi_1 \succeq \pi_2$ and π_2 covers example d, then π_1 covers example d.

A good generality relation is usually not difficult to choose. If the coverage relation is transitive, one can always use the coverage relation as generality relation as well. For instance, in itemset mining, the subset relation is usually used as generality relation as well: an itemset π_1 is more general than an itemset π_2 iff $\pi_1 \subseteq \pi_2$.

Based on the generality relationship, we can define the *anti-monotonicity* property of constraints[1]. A constraint $\varphi(\pi, D)$ is called anti-monotonic iff it holds for all patterns π_1, π_2 that

> if $\pi_1 \succeq \pi_2$ and $\varphi(\pi_1, D)$ is false, then $\varphi(\pi_2, D)$ is false.

Minimum support is the most well-known constraint that is anti-monotonic, but several other constraints are also anti-monotonic [21, 11]. Assuming that we use the subset relation to determine the generality relation, the following constraints on itemsets are anti-monotonic:

- the maximum length constraint $|\pi| \leq \theta$ for a fixed θ;
- the maximum sum of costs constraint $c(\pi) \leq \theta$ is anti-monotonic, where $c(\pi)$ sums up the costs of the items in the itemset, $c(\pi) = \sum_{i \in \pi} c(i)$, and $c(i) \leq 0$ is a cost that is associated to each item;
- a generalization constraint, which for a given set I requires that all itemsets found satisfy $\pi \subseteq I$;
- conjunctions or disjunctions of other anti-monotonic constraints.

These constraints can be generalized to other types of patterns as well.

Monotonocity Closely related to anti-monotonicity is monotonicity. A constraint is called *monotonic* iff for all patterns π_1, π_2:

> if $\pi_1 \succeq \pi_2$ and $\varphi(\pi_1, D)$ is true, then $\varphi(\pi_2, D)$ is true.

[1]Note that in some publications, anti-monotonic constraints are called monotonic, and monotonic constraints anti-monotonic[21].

In other words, monotonicity is the "reverse" of anti-monotonicity; if a constraint $\varphi(\pi)$ is anti-monotonic, its negation $\neg\varphi(\pi)$ is monotonic. This includes constraints such as:

- the maximum support constraint $|\{d \in \mathcal{D} | \pi \subseteq d\}| \leq \theta$;
- the minimum size constraint $|\pi| \geq \theta$;
- the minimum sum of costs constraint $c(\pi) \geq \theta$;
- a negated generalization constraint $\pi \not\subseteq I$;
- a specialization constraint $\pi \supseteq I$.

This relationship between monotonic and anti-monotonic constraints, one of reversal and negation, already hints at the difficulty in using both types of constraints at the same time for efficient pattern enumeration.

Convertible (anti)-monotonicity Whether a constraint is (anti)-monotonic depends on the generality relation chosen. One of the most well-known examples is that of the maximum *average cost* of an itemset, $c(\pi) = \sum_{i \in \pi} c(i)/|\pi| \geq \theta$. If we use the subset relation to define the generality, this constraint is not anti-monotonic. Consider the following two items with their corresponding costs: $c(1) = 1$ and $c(2) = 3$. If our cost threshold is 2, the average cost of $\{1, 2\}$ is 2 and satisfies the requirement; however, itemset $\{2\}$, while a subset, does not satisfy the constraint.

However, assume that we would use the following generality order:

> $\pi_1 \succeq \pi_2$ if we can obtain π_1 from π_2 by repeatedly removing from π_2 the item with the highest cost.

Then under this order the constraint is anti-monotonic: after all, by removing the most costly items from an itemset, the average cost of the items in the itemset can only go down and it hence also must satisfy the constraint.

Note that this order is *compatible* with the use of the subset relation as coverage relation; hence, this constraint can be combined with a minimum support constraint. Such an order can be *incompatible* with another order needed to make another constraint anti-monotonic, however.

Constraints which have this property, i.e., that a different generality relation needs to be used than the coverage relation to obtain (anti-)monotonicity, are called *convertible (anti)-monotonic* in the literature [28].

Succinctness Succinctness was originally defined for itemsets [25], but we will use a slightly different definition here which is applicable to other pattern languages as well: we will call any constraint *succinct* that can be enforced by manipulating the data. Consider the following two examples:

- we want to find frequent itemsets without item i: if we remove item i from the database, we will no longer find such itemsets;

- we want to find frequent itemsets that include item i: if we remove all transactions without item i from the database, and then remove item i from the remaining transactions, we can add item i to every itemset we find in the resulting database to obtain the desired set of itemsets.

These examples can easily be generalized to require the inclusion or exclusion of an itemset $\pi \subseteq \mathcal{I}$.

Condensed representations The set of patterns satisfying the above constraints may still be large. *Condendensed representations* consitute an additional approach for reducing a set of patterns. The main idea is to determine a small set of patterns that still is sufficiently large to determine a full set of patterns. The property that a pattern is part of a condensed representation can also be seen as a constraint.

We will discuss two of the most well-known cases here.

> Given a generality relation $\succeq$, a pattern π is called *closed* if there is no more specific pattern π' with $\pi \succ \pi'$ such that $cover(\pi) = cover(\pi')$.

Intuitively, closed frequent patterns [27] allow one to recover a set of frequent itemsets together with their supports.

A subtle issue is the combination of the closedness constraint with other constraints. As an example, consider the maximum size constraint. One can distinguish two settings:

- a setting in which one searches for patterns satisfying the size constraint among those patterns that are closed;
- a setting in which one searches for patterns that are closed, restricting the set of patterns that are considered in the closedness definition only to those that satisfy the constraint.

As an example, assume that $\{1\}$ is not closed and that $\{1, 2\}$ is closed, while we have a maximum size constraint of 1. Then itemset $\{1\}$ would not be in the output in the first setting, but would be in the output of the second. Constraint-based pattern mining systems can differ in their approach for dealing with this issue.

Another condensed representation is that of maximal patterns.

> Given a generality relation $\succeq$ and constraint φ, a pattern π that satisfies constraint φ is called *maximal* with respect to constraint φ if there is no more specific pattern π' with $\pi \succ \pi'$ such that π' satisfies the constraint.

Compared to closed itemsets, maximal itemsets [1, 21] no longer allow one to recover the supports of a set of patterns.

If the constraint φ is a minimum support constraint, one typically refers to maximal frequent patterns. Whereas maximal frequent patterns are the

most popular, it can also be useful to study maximality with respect to other constraints. Essentially, any anti-monotonic constraint defines a *border* in the space of patterns where all patterns that satisfy the constraints are on one side of the border, while all other patterns that do not satisfy it are on the other side [21, 11].

Similarly, also a *monotonic* constraint defines a border: in this case, the border one is looking for is that of *minimal* patterns that satisfy the constraints.

Different borders can be combined. Probably the most well-known example of this is found in the analysis of supervised data. If the database consists of two classes of examples, one can ask for all patterns that are frequent in the one, but infrequent in the other; the resulting set of patterns has two borders: one of the most specific patterns in this set, the other of the most general ones.

Boundable Constraints The minimum support constraint is one example of a constraint of the kind $f(\pi) \geq \theta$. Over the years, more complex functions have been studied. One example is that of accuracy (see the chapter on supervised patterns), which calculates the accuracy of a pattern when used as a classification rule on supervised data. Many such functions no longer have the (anti-)monotonicity property. In some cases, however, one can identify an alternative function f' such that:

- it is feasible to mine all patterns with $f'(\pi) \geq \theta$;
- $f'(\pi) \geq f(\pi)$.

In this case, all patterns satisfying $f(\pi) \geq \theta$ could be determined by first mining all patterns with $f'(\pi) \geq \theta$ and then calculating $f(\pi)$ for all patterns found. Function f' can be considered a *relaxation* of function f [31]. In Chapter **??** it was discussed that for supervised data such bounds often exist.

3 Level-Wise Algorithm

Most constraint-based mining algorithms can be seen as generalized versions of frequent pattern mining algorithms. Similar to frequent itemset mining algorithms, consequently, both breadth-first (*BFS*) or level-wise, and depth-first search algorithms have been proposed. The earliest techniques typically were BFS approaches, on which later works improved. Hence, we discuss them first.

3.1 Generic Algorithm

The setting which is closest to frequent pattern mining is that of constraint-based mining under anti-monotonic constraints. In this case, we can perform a level-wise search that is mostly equal to that of the Apriori algorithm [21]. The search starts from the empty pattern, and proceeds by specializing this pattern in a breadth-first fashion.

In Algorithm 1, a description of this algorithm is given. In this pseudo-code, we use two operators: a *downward refinement operator* ρ to specialize patterns and an *upward refinement operator* δ to generalize patterns. A downward refinement operator is an operator which for any pattern π returns a *set* of more specific patterns (i.e. for all patterns $\pi' \in \rho(\pi)$ it holds that $\pi \succ \pi'$). Typically, we assume that this operator is *globally complete*, i.e. its repeated application starting from the empty pattern will produce the complete pattern language[2]. Furthermore, this operator works in "small steps", it tries to create new patterns which are minimally more specific (*least specific specializations*).

An example of a downward refinement operator for itemset mining is $\rho(\pi) = \{\pi \cup \{i\} \mid i > \max(\pi)\}$, assuming a total order $>$ on the items; eg., if our language is $2^{\{1,2,3,4\}}$, with the usual order of integers over the items, $\rho(\{2\}) = \{\{2,3\},\{2,4\}\}$.

Similarly, the upward refinement operator δ returns generalizations. For a given pattern π, it is assumed to only generate patterns that should have been seen before pattern π by the level-wise algorithm.

The key property on which the algorithm relies is the anti-monotonicity of constraint φ under the chosen generality relation: by refining only patterns that satisfy the constraint in line 6 and by checking generalizations in line 7, patterns are removed from consideration that are known to specialize patterns that do not satisfy φ.

Algorithm 1 Level-Wise Search(Constraint: φ)

1: $\mathcal{C} := \{\emptyset\}$
2: $\mathcal{S} := \emptyset$
3: **while** $\mathcal{C} \neq \emptyset$ **do**
4: $\quad S := \{\pi \in \mathcal{C} \mid \varphi(\pi) \text{ is true}\}$
5: $\quad \mathcal{S} := \mathcal{S} \cup S$
6: $\quad \mathcal{C}' := \bigcup_{\pi \in S} \rho(\pi)$
7: $\quad \mathcal{C} := \{\pi \in \mathcal{C}' \mid \delta(\pi) \backslash \mathcal{S} = \emptyset\}$
8: **return** $\mathcal{F}$

Note that this algorithm can also be applied to convertible and boundable constraints [30]: in this case, a modified generality relation or constraint is used. It can also be applied in a straightforward manner if there are both anti-monotonic constraint and non anti-monotonic constraints: in principle, we ignore the non anti-monotonic constraints during the search, and evaluate the remaining constraints for *all* found patterns *afterwards*, in a *post-processing phase*.

In the presence of *monotonic* constraints, we can improve somewhat on the need to check *all* patterns [21, 18, 19, 11]. The main idea is here to traverse the patterns in a level-wise fashion in *reverse order* by starting with the most specific

[2]More formally, let $\rho^n(\pi)$ denote the set $\bigcup_{\pi' \in \rho(\pi)} \rho^{n-1}(\pi')$, with $\rho^0(\pi) = \pi$, and let $\rho^*(\pi) = \bigcup_{i=1}^{\infty} \rho^i(\pi)$, then a refinement operator is complete if $\rho^*(\emptyset)$ equals the pattern language $\mathcal{L}$.

patterns that satisfy the anti-monotonic constraint. Since the generalizations of a pattern that does not satisfy a monotonic constraint will not satisfy the constraint either, we can stop this reverse traversal at the point at which we no longer have patterns that satisfy the constraint. This does not change the fact that we cannot enforce the monotonic constraint in the first mining phase, however.

The level-wise algorithm is easily changed to deal with *border representations*. Assume that upward refinement operator δ generates all *least general generalizations* of a pattern π (a pattern π' is a least general generalization for a pattern π if there is no pattern π'' with $\pi' \succ \pi'' \succ \pi$). Then for each pattern π that satisfies an anti-monotonic constraint, we can essentially identify its immediate generalizations, which are clearly *not* maximal, and remove them from the solution set.

For instance, in the case of itemset mining such an operator is $\delta(\pi) = \{\pi \backslash \{i\} \,|\, i \in \pi\}$. It would remove all immediate subsets of an itemset from the output. As it is assumed that the upward refinement operator will always generate patterns that must have been seen already, we do not need to explicitly remove other generalizations from the output: they will have been removed at an earlier stage. With similar ideas, the minimal patterns on the border of a monotonic constraint can also be found.

4 Depth-First Algorithm

Note, however, that even though the output of these modified BFS algorithms for finding borders is correct, the running time will not be much better than that of an algorithm that generates *all* patterns satisfying the constraint. Most algorithms that are able to obtain dramatically better run times in practice are depth-first algorithms.

4.1 Basic Algorithm

The most basic depth-first constraint-based mining algorithm is given in Figure 2 and only supports anti-monotonic constraints.

Algorithm 2 Depth-First Search(Constraint: φ , pattern π)

```
1: F := ∅
2: if φ(π) is true then
3:    for π' ∈ ρ(π) do
4:       F := F ∪ Depth-First Search (φ,π')
5: return F
```

Essentially, compared to the earlier level-wise algorithm, this algorithm traverses the search space in a different order in which some long patterns are already considered before some shorter patterns are evaluated. As a result, optimizations based on the fact that short patterns have been seen before long

patterns are not used. In practice, however, these algorithms can be more efficient. The reason for this is that most implementations take care to maintain datastructures which allow for *incremental* constraint evaluation: for instance, to calculate the support of a pattern, they do not traverse the whole dataset, but only consider those transactions covered by the pattern's parent in the search tree. As depth-first algorithms do not need to maintain a large number of candidates, maintaining such additional data structures is feasible. A well-known datastructure in this context is the FP-Tree (see the chapter on pattern growth for more details) [29].

Note that the above algorithm works for any pattern language, including graphs, strings and trees, as long as we know that the constraint φ is anti-monotonic.

When some constraints are not anti-monotonic, a basic approach for dealing with them, as in the case of breadth-first search, is to ignore them during the search and post-process the output of the above agorithm. A similar trick can be used for boundable constraints. In this case, the anti-monotonic bound is used during the depth-first search, and each pattern found is finally evaluated using the original constraint in a post-processing step.

Many studies have explored the possibilities for deriving more efficient algorithms for more complex constraints than anti-monotonic constraints. Most of these studies have focused on the pattern language of *itemsets*, as it appears additional pruning is most easily derived for itemsets. We will discuss these approaches in a generic way in the next paragraph, inspired by work of Bucila et al. and Guns et al. [8, 14].

4.2 Constraint-based Itemset Mining

The key idea in efficient depth-first constraint-based itemset mining algorithms is to maintain four sets in each node of the search tree, which are upper- and lower-bounds on the itemsets and transaction sets that can still be found:

- I_U, the largest itemset we believe we can still find;
- I_L, the smallest itemset we believe we can still find;
- T_U, the largest transaction set we believe we can still find;
- T_L, the smallest transaction set we believe we can still find.

For some constraints, not all these 4 sets need to be maintained, but in the most generic setting all 4 sets are maintained.

During the search, any modification of any of these 4 sets may be a reason to modify another of these 4 sets as well. We will refer to this process of modifying one set based on the modification of another set as *propagation*. Different approaches differ in the algorithms and data structures used to do propagation.

An overview of the generic algorithm is given in Algorithm 3, in which $I_L = T_L = \emptyset$, $I_U = \mathcal{I}$ and $T_U = \mathcal{D}$. Line 1 performs the propagation for the constraints. Propagation may signal that no solution can be found in the current

branch of the search tree by setting *stop* to true. If the lower- and upper-bound for the itemset are identical, a pattern has been found and is added to the output. Otherwise, in line 6 an item is selected, which is recursively added to the itemset (line 7), or removed from consideration (line 8).

Note that the same itemset can never be found twice: an item which is added in line 7, will never be added to an itemset that is considered in the search tree explored in the call of line 8.

Algorithm 3 Depth-First Search(Constraint: φ , I_L, I_U, T_L, T_U)

```
1:  I'_L, I'_U, T'_L, T'_U, stop := Propagate (I_L, I_U, T_L, T_U, φ)
2:  if not stop then
3:    if I'_L = I'_H then
4:      return {I'_L}
5:    else
6:      Pick an item i ∈ I'_U \ I'_L
7:      return  Depth-First Search(φ, I'_L ∪ {i}, I'_U, T'_L, T'_U) ∪
8:                Depth-First Search(φ, I'_L, I'_U \ {i}, T'_L, T'_U)
9:  else
10:   return ∅
```

We will now consider how this algorithm can be instantiated for different types of mining settings.

Frequent Itemset Mining This is the most simple setting. Essentially, in this case, the following propagation steps are executed:

1. T'_U is restricted to $cover(I_L)$;
2. I'_U is restricted to those items in I_U that are frequent in the database containing only the transactions of T'_U (in other words, the items that are frequent in the projected database for itemset I_L, see chapter **??**);

T'_L and I'_L are not modified by propagation.

For these choices, the search is highly similar to that of Eclat or FP-Growth; at every point in the search tree, we maintain a list of candidate items that can be added to the current itemset; the set of candidate items is reduced based on the minimum support threshold.

Attentive readers may have noticed that the search tree for the generic algorithm presented here is binary, whereas for most itemset mining algorithms the tree is not binary. This is, however, only a minor conceptual difference: the recursive calls in line 8 of our generic algorithm essentially correspond to a traversal of the candidate list in traditional frequent itemset mining algorithms, where we remember which items we may no longer consider.

The clear benefit of the non-traditional perspective is that other constraints can be added with minor effort.

Minimum Sum of Cost and Minimum Support A first approach for dealing with a monotonic minimum-sum-of-cost constraint is to add the following propagation [28]:

3 if the sum of costs of itemset I'_U is lower than the desired threshold, set *stop* to true.

The rationale for this propagation step is that we can stop a branch of the search if the most expensive itemset we can still reach is not expensive enough.

The benefit of this approach is that this propagation step is relatively easy to calculate, while for high thresholds it will already prune effectively.

A more elaborate approach was proposed by Bonchi et al. [7, 6, 5]. Its essential observation is that if an itemset needs to be both frequent and expensive, this means that a certain number of transactions in the data needs to be expensive as well. This leads to the following propagation steps:

1. T'_U is set to $T_U \cap cover(I_L)$;
2. I'_U is set to those items in I_U that are frequent in the database restricted to the transactions in T'_U;
3. from T'_U all transactions d are removed for which $c(d \cap I'_U) < \theta$, where θ is the cost threshold;
4. if T'_U was changed, go back to step 2;
5. if $I'_U \subset I'_L$, set *stop* to true.

The interesting idea in this approach is the presence of a feedback loop: the removal of items can make some transactions too cheap; when a transaction is too cheap, it will not be in the cover of an itemset, and we can remove it from consideration; this however will reduce the support of items further, potentially making them infrequent in the projected database.

The advantage of this approach is that it can reduce the size of the search tree even further. The disadvantage is that the propagation is more complex too calculate, as it involves a traversal of the data. To remedy this, Bonchi et al. [6] studied settings in which the above loop is not executed in all nodes of the seach tree.

Minimum and Maximum Support A similar idea can be used when we have a minimum support threshold on some transactions ($\mathcal{D}^+$), and a maximum support threshold on the other transactions ($\mathcal{D}^-$) [9, 19].

1. T'_U is set to $cover(I_L)$;
2. I'_U is set to those items in I_U that are frequent in the database restricted to the transactions in $T'_U \cap \mathcal{D}^+$;
3. T'_L is set to $cover(I_U)$;

4. if $|T'_L \cap \mathcal{D}^-| > \theta$, where θ is the maximum support threshold for the negative examples, then set *stop* to true.

In this approach, the main idea is that if the lowest support that can be reached on the negative transactions is not low enough, we can stop the search.

Maximal Frequent Itemsets Of particular interest in the constraint-based mining literature is the discovery of border (or boundary) representations. The simplest such setting is the discovery of maximal frequent itemsets, which can be obtained by means of the following propagations:

1. T'_U is set to $cover(I_L)$;
2. I'_U is set to those items in I_U that are frequent in the database restricted to the transactions in T'_U;
3. T'_L is set to $cover(I'_U)$;
4. if some item not in I'_U is frequent in the database restricted to the transactions in T'_L, set *stop* to true.
5. if $|T'_L| \geq \theta$, I'_L is set to I'_U.

The arguments for these steps are the following: the set T'_L represents those transactions that will be covered by any itemset we will find in the future; if there is an item that covers a sufficiently large number of these transactions, but we cannot add this item in the current branch of the search tree, we stop traversing this branch in line 4, as elsewhere we will find itemsets that include this item.

On the other hand, if the itemset consisting of all remaining items is frequent, clearly this itemset must be maximal; we can directly include all items in the itemset.

This search strategy is embodied in the MaxMiner algorithm [1]. It was generalized to the case of finding border representations under arbitrary monotonic and anti-monotonic constraints by Bucila et al. [8].

Closed Frequent Itemsets Closed itemset mining can be achieved by another modification of the propagation for frequent itemset mining:

1. T'_U is set to $cover(I_L)$;
2. I'_U is set to those items in I_U that are frequent in the database restricted to the transactions in T'_U;
3. let I'' contain those items in $\mathcal{I}$ which are present in all transactions in T'_U;
4. if I'' contains items not in I'_U, set *stop* to true; otherwise, let I'_L be I''.

Remember that in closed itemset mining the task is to find itemsets such that no superset has the same coverage. This propagation ensures this: in line 4, if an item can be added to the current itemset without changing the coverage, it will be added immediately if this is allowed; however, if this item may not be added, as we branched over it earlier, we stop the search, as we can no longer find closed itemsets in the current part of the search space.

This search strategy is embodied in the LCM closed itemset mining algorithm [32]. The combination of closed itemset mining with constraints was studied in more detail in the D-Miner system by Besson et al. [2, 3].

4.3 Generic Frameworks

The similarity between these depth-first search algorithms indicates that it may be possible to combine different constraints and condensed representations. Indeed, this is the key idea underlying most generic frameworks for constraint-based mining.

The DualMiner algorithm [8] essentially represents a generic depth-first algorithm for finding border representations that extends the ideas found in the MaxMiner algorithm. The authors of that work brought this development to its logical conclusion by introducing the concept of "witnesses" [17], itemsets on which constraint satisfaction is tested to derive pruning information for parts of the search space. Witness-based mining subsumes mining under (anti-)monotonic and convertible constraints, and is capable of handling additional constraint classes, and mining under conjunctions of constraints. The D-Miner system combines closed itemset mining (formal concept analysis) with constraints [2, 3].

The Constraint Programming for Itemset Mining framework [14] is built on the observation that constraint-based search, and constraint programming in particular, has been studied extensively in the general artificial intelligence literature. It shows that the mining tasks discussed earlier can be reformalized in terms of constraints present in generic constraint programming systems; furthermore, such systems provide a generic framework for constraint propagation which makes it easy to combine different constraints.

4.4 Implementation Considerations

In the above description, we intentionally left unaddressed how the indicated propagation is performed in detail. In principle, all different data structures that have been studied in the frequent itemset mining literature can be used in this context as well. For instance, the MaxMiner and DualMiner algorithms use vertical representations of the data most similar to that of Eclat; the FP-Bonsai algorithm, on the other hand, uses FP-Trees [7]. The impact of data structures in a Constraint Programming framework was studied by Nijssen et al [26]. These studies confirm that for good run times the choice of data structure is important; however, many of the above propagation procedures can be adapted to different data representations, and hence the two aspects can be considered orthogonal.

5 Languages

Most of the systems studied earlier require a language for the specification of constraints. Roughly speaking, three categories can be distinguished within these languages: special purpose languages, SQL inspired languages, and constraint programming based languages.

Special purpose languages Many constraint-based mining systems implement a small special purpose language. As an example, this is an expression in the language underlying the SeqLog system [20]:

```
database ca = smiles_file("molecules.ca");
database ci = smiles_file("molecules.ci");

predicate ca = minimum_frequency(ca, 10);
predicate ci = maximum_frequency(ci, 500);

mine ca and ci;
```

Essentially, this language provides a small set of built-in primitives such as `smiles_file` for reading a data file, `minimum_frequency` for specifying a minimum support constraint and `maximum_frequency` for specifying a maximum support constraint. For each of these primitives, the system is aware of the properties such as (anti-)monotonicity, which ensures that any conjunction or disjunction of constraints that is written is down can be processed by the system.

Similar special purpose languages were proposed by several other authors [31, 23]; they differ in the constraints that are supported and the type of patterns that can be found (itemsets [31, 23], strings [12, 20], ...).

Languages built on SQL A clear disadvantage of special purpose languages is that they are yet additional languages that the programmer has to learn. Given that many datasets are stored in databases, several projects have studied the integration of constraint-based pattern mining in database systems.

The first class of such methods aims to extend SQL with additional syntax for the formalization of data mining tasks. One early example is the `MINE RULE` operator [22]:

```
MINE RULE Associations AS
SELECT DISTINCT 1..n item as BODY, 1..n item AS HEAD,
                SUPPORT, CONFIDENCE
FROM Purchase
WHERE price <= 150
GROUP BY transaction
EXTRACTING RULES WITH SUPPORT: 0.1, CONFIDENCE: 0.2
```

This example mines association rules with minimum support 0.1, confidence 0.2, limiting the search to items with a price lower than $150, a succinct constraint. Another example is the DMQL language [15]:

```
FIND association rules
RELATED TO beer, wine, diapers
FROM products
WHERE value >= 100
WITH support threshold = 0.1
WITH confidence threshold = 0.2
```

In this example we search for association rules related to three specific products, in those transactions that have a value higher than 100; the parameters of the association rule discovery process are similar to the previous example. A third example is SPQL [5].

The advantage of these languages is that well-known syntax can be used for the expression for constraints. Furthermore, common SQL syntax can be used to specify the input of the mining task or to process its output further.

At the same time, the programmer still has to learn the additional primitives, such as the `FIND` or `MINE RULE` keywords. An alternative perspective is to avoid extending the language, but to add *mining views* to a database [4]. They are virtual tables, which, once queried, will trigger the execution of mining algorithms. This is an example:

```
SELECT R.rid, C1.*, C2.*, R.conf
FROM Sets S, Rules R, Concepts C1, Concepts C2,
WHERE R.cid = S.cid AND C1.cid = R.cida AND C2.cid = R.cidc AND
      S.supp >= 30 AND R.conf >=80
```

Here, `Sets`, `Rules` and `Concepts` are virtual mining views.

A limitation of most SQL-based approaches is however that they are limited to itemset patterns or association rules. How to specify graph mining or sequence mining tasks in this context is still an open question. Most constraint-based graph mining or sequence mining systems currently use special purpose languages.

The general idea of linking constraint-based mining to database querying has been studied in the area of *inductive databases* and *inductive querying* [16, 10].

Constraint programming Constraint-based mining has many similarities to generic constraint satisfaction problem (CSP) solving as studied in the Artificial Intelligence (AI) community. Both areas essentially require the discovery of solutions in a space of possible solutions satisfying constraints. To deal with generic CSPs, the AI community has developed generic systems known as *constraint programming systems*. These systems provide languages in which programmers can specify constraint satisfaction problems; statements in these languages can be solved by various types of solvers, including generic propagation-based solvers. As we have seen earlier, many depth-first constraint-based itemset mining systems are also based on propagation, and hence it is not surprising that generic constraint-based itemset mining fits naturally into a constraint programming context as well.

This observation was used by Guns et al. to formalize constraint-based itemset mining tasks in generic constraint programming languages [14, 13]. This is an example in the most recent version of the MiniZinc constraint programming language:

```
int: Nr I; int: NrT; int: Freq;
array[1..NrT] of set of 1..NrI: TDB;
var set of 1..Nr I: Items;
constraint card ( cover ( Items, TDB ) ) >= Freq;
solve satisfy;
```

It specifies the task of frequent itemset mining; `cover` is a function available in a MiniZinc library, implemented in the MiniZinc language ifself as well.

Statements in the MiniZinc language can be executed by a generic constraint programming system, or by a specialized data mining system, if one exists [13]. However, it was shown that generic constraint programming systems implement many types of propagation automatically, and hence that specialized systems are often not needed if a task can be modelled in the MiniZinc language.

Similar to the SQL-based languages, it is at this moment not understood how to integrate graph mining or sequence mining tasks in an elegant matter in the CP setting.

6 Conclusions

In this chapter we provided an overview of classes of constraints, algorithms for solving constraint-based mining problems and languages for specifying contraint-based mining tasks.

The trend in constraint-based mining has been to build increasingly generic systems. While initially constraint-based mining systems provided special purpose languages that only supported slightly more constraints than specialized frequent itemset mining algorithms did, in recent years the range of constraints has expanded, as well as the genericity of the languages supporting constraint-based mining, culminating in the integration with generic constraint satisfaction systems and languages.

Several open challenges remain. These include a closer integration of constraint-based mining with pattern set mining, getting a better understanding of how to integrate statistical requirements in constraint-based mining systems, and mining structured databases such as graph or sequence databases using sufficiently generic languages.

References

[1] R. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1998.

[2] Jérémy Besson, Céline Robardet, and Jean-François Boulicaut. Constraint-based mining of formal concepts in transactional data. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference*, pages 615–624, Sydney,Australia, May 2004. Springer.

[3] Jérémy Besson, Céline Robardet, Jean-François Boulicaut, and Sophie Rome. Constraint-based concept mining and its application to microarray data analysis. *Intell. Data Anal.*, 9(1):59–82, 2005.

[4] Hendrik Blockeel, Toon Calders, Élisa Fromont, Bart Goethals, Adriana Prado, and Céline Robardet. An inductive database system based on virtual mining views. *Data Min. Knowl. Discov.*, 24(1), 2012.

[5] Francesco Bonchi, Fosca Giannotti, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Roberto Trasarti. A constraint-based querying system for exploratory pattern discovery. *Inf. Syst.*, 34(1):3–27, 2009.

[6] Francesco Bonchi, Fosca Giannotti, Alessio Mazzanti, and Dino Pedreschi. Examiner: Optimized level-wise frequent pattern mining with monotone constraint. In *ICDM*, pages 11–18. IEEE Computer Society, 2003.

[7] Francesco Bonchi and Bart Goethals. Fp-bonsai: The art of growing and pruning small fp-trees. In *PAKDD*, pages 155–160, 2004.

[8] Cristian Bucila, Johannes Gehrke, Daniel Kifer, and Walker White. DualMiner: A dual-pruning algorithm for itemsets with constraints. In *Proceedings of The Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton,Alberta,Canada, July 23–26 2002.

[9] L. De Raedt and S. Kramer. The level wise version space algorithm and its application to molecular fragment finding. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 853–862. Morgan Kaufmann, 2001.

[10] Luc De Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, 2002.

[11] Luc De Raedt, Manfred Jaeger, Sau Dan Lee, and Heikki Mannila. A theory of inductive query answering. In *ICDM*, pages 123–130. IEEE Computer Society, 2002.

[12] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Spirit: Sequential pattern mining with regular expression constraints. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 223–234. Morgan Kaufmann, 1999.

[13] Tias Guns, Anton Dries, Guido Tack, Siegfried Nijssen, and Luc De Raedt. Miningzinc: A modeling language for constraint-based mining. In *IJCAI*, 2013.

[14] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

[15] Jiawei Han, Yongjian Fu, Krzystzof Koperski, Wei Wang, and Osmar Zaiane. Dmql: A data mining query language for relational databases. In *SIGMOD'96 Workshop. on Research Issues on Data Mining and Knowledge Discovery (DMKD'96)*, 1996.

[16] T. Imielinski and H. Mannila. A database perspectivce on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[17] Daniel Kifer, Johannes Gehrke, Cristian Bucila, and Walker M. White. How to quickly find a witness. In *PODS*, pages 272–283. ACM, 2003.

[18] Stefan Kramer, Luc De Raedt. Feature Construction with Version Spaces for Biochemical Applications. In *Proceedings of the Eighteenth International Conference on Machine Learning(ICML)*, 2001.

[19] Stefan Kramer, Luc De Raedt, and Christoph Helma. Molecular feature mining in hiv data. In *KDD-2001: The Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001.

[20] Sau Dan Lee and Luc De Raedt. An algebra for inductive query evaluation. In *ICDM*, pages 147–154, 2003.

[21] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[22] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A new sql-like operator for mining association rules. In *VLDB*, pages 122–133, 1996.

[23] Jean-Philippe Métivier, Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, and Samir Loudni. A constraint-based language for declarative pattern discovery. In *ICDM Workshops*, pages 1112–1119, 2011.

[24] S. Morishita and J. Sese. Traversing itemset lattice with statistical metric pruning. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2000.

[25] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 13–24, 1998.

[26] Siegfried Nijssen and Tias Guns. Integrating constraint programming and itemset mining. In *ECML/PKDD (2)*, pages 467–482, 2010.

[27] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In Catriel Beeri and Peter Buneman, editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1999.

[28] Jian Pei and Jiawei Han. Can we push more constraints into frequent pattern mining? In *KDD*, pages 350–354, 2000.

[29] Jian Pei and Jiawei Han. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explorations*, 4(1):31–39, 2002.

[30] Jian Pei, Jiawei Han, and Laks V. S. Lakshmanan. Pushing convertible constraints in frequent itemset mining. *Data Min. Knowl. Discov.*, 8(3):227–252, 2004.

[31] Arnaud Soulet and Bruno Crémilleux. Mining constraint-based patterns using automatic relaxation. *Intell. Data Anal.*, 13(1):109–133, 2009.

[32] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *Discovery Science*, pages 16–31, 2004.