On declarative modeling of structured pattern mining

Tias Guns and Sergey Paramonov and Benjamin Negrevergne
KU Leuven, Belgium Inria Rennes, France
{first.lastname@cs.kuleuven.be} {first.lastname@inria.fr}

Abstract

Since the seminal work on frequent itemset mining, there has been considerable effort on mining more structured patterns such as sequences or graphs. Simultaneously, the field of constraint programming has been linked to the field of pattern mining resulting in a more general and declarative constraint-based itemset mining framework. A number of recent papers have logically proposed to extend the declarative approach to structured pattern mining problems.

Because the formalism and the solving mechanisms are vastly different in specialised algorithm and declarative approaches, assessing the benefits and the drawbacks of each approach can be difficult. In this paper, we introduce a framework that formally defines the core components of itemset, sequence and graph mining tasks, and we use it to compare existing specialised algorithms to their declarative counterpart. This analysis allows us to draw clear connections between the two approaches and provide insights on how to overcome current limitations in declarative structured mining.

Introduction

Pattern mining is a well-studied subfield of data mining, where the goal is to find patterns in data. Different types of patterns have been investigated, such as itemsets (e.g. a set of books that customers bought), sequences (e.g. phone call logs) and graphs (e.g. molecules or designs).

Depending on the application, the user might have a certain view on which patterns are interesting. In constraint-based mining the user expresses this interestingness through constraints on the patterns. The best-known constraint is the minimum-frequency constraint, which indicates that all patterns that appear frequently in the data (limited by a threshold) are potentially interesting.

A wide range of constraints have been proposed in the literature, often together with specialised algorithms. However, combining constraints has typically required modifications to the original algorithms. Declarative constraint-based methods offer a promising alternative as they can handle complex constraints natively. Indeed, many works [6, 9, 18] have shown the benefits of constraint-based itemset mining.

Itemset mining however is a special case, as checking whether a pattern matches a transaction (a data entry) simply consists of checking whether the pattern is a substructure in the transaction. For example, it is easy to verify that \{a,c\} \subseteq \{a,b,c,d\} and \{a,c\} \not\subseteq \{a,b,d\}. For sequence mining checking this is not so easy: let \langle a,c \rangle be a sequence pattern, then it matches \langle x,a,x,y,c,a,x,c \rangle in three different ways, at positions (2,5),(2,8),(6,8). Hence, one should search for whether there exists at least one embedding, instead of simply checking a logical relation. For graphs this is further complicated as this amounts to a subgraph isomorphism check, which is an NP-complete problem.

To better understand and analyse the situation, we define the key components of a structured mining problem. We extend the theory of Mannila and Toivonen [15] on constraint-based mining with more fine-grained operators. More explicitly, we define the concept of matching operator, conicity operator and extension operator.

Using these principles, we analyze how existing state-of-the-art algorithms implement these operators, and how they can be formulated for use in declarative systems. This deeper understanding of both the theory and the relation to practice can pave the way to more general and declarative structured pattern mining systems.

Operators of pattern mining

In the seminal paper of Mannila and Toivonen [15] pattern mining is presented as the task of finding \( TH(\mathcal{L}, r, q) \), given a language of patterns...
\(\mathcal{L}\), a database \(r\) and a selection predicate \(q\), such that \(\mathcal{T}h(\mathcal{L}, r, q) = \{\varphi \in \mathcal{L} | q(r, \varphi)\text{ is true}\}\). The paper draws the link to the level-wise algorithm and how it can be instantiated to tasks such as association rule mining and episode mining.

The formalism fits a declarative solving approach in that \(q\) is in fact a constraint specification. However, \(\mathcal{L}\) is left unspecified and may be non-trivial and \(q\) may use more expressive constructs than solvers support, e.g. beyond first order logic.

To better investigate the characteristics of different structure mining problems, we present a more fine-grained framework with an explicit pattern type and different operators. The two prime approaches in that regard are association rule mining and episode mining.

**Data, patterns and operators**

**Data** Pattern mining is always in the context of one or more datasets from which we can derive which patterns are interesting and which ones are not. We assume a dataset is a set of transactions \((\text{tid}, \text{point})\) where \(\text{tid}\) is the transaction identifier and \(\text{point}\) is the actual data entry.

**Pattern** We define a pattern as an object of a certain syntactic type, for example, an itemset represented as a set of items. However, an itemset can also be represented as a bitvector or as an array of items.

Given a pattern \(p\), the length operator \(\text{length}_t(p)\) returns the length of a pattern of type \(t\), for example the number of items in an itemset or number of edges of a graph.

**Generating operator** \(\mathcal{G}_t\) is the operator that generates all possible patterns of type \(t\). We simply write \(\mathcal{G}\) if the pattern type is irrelevant or clear from the context. The generating operator will ensure only valid patterns are generated. For example, in a vector representation of an itemset every item must appear only once, and a graph pattern has to be connected.

We also need to ensure that every pattern is generated only once. For example, Figure 1 shows two graphs that are syntactically different (different labels for different identifiers), however they are isomorphic and semantically represent the same graph. For this, we introduce the canonicality operator \(\mathcal{K}\). It maps a pattern \(p\) to a canonical representation of the pattern \(\mathcal{K}(p)\). This canonical representation will be the same (syntactically) for any two patterns that are semantically equal.

The generating operator should only generate one pattern from the class of patterns with the same canonical representation, for example, the canonical pattern for which \(\mathcal{K}(p) = p\).

**Constraining operator** \(\mathcal{C}_c\) is the operator checking validity of a pattern \(p\) on a set of constraints \(c\). Each constraint can involve constants such as thresholds or datasets. \(\mathcal{C}_c(p) = \top\) iff \(p\) satisfies all constraints in \(c\).

The most fundamental constraint is the minimum frequency constraint. This constraint requires a matching operator \(\mathcal{M}_{\leq}\) that will be used the count the number of transactions that match the pattern. The relation \(\sqsubseteq\) is the relation defining structure specificity, that is, when an object of a certain type is a sub-structure (e.g. subset, sub-graph, ...) of another object of the same type.

**Matching operator** \(\mathcal{M}_{\leq}\). This operator maps a pattern \(p\) and a data point \(g\) of the same type, to a set of functions \(h\) over \(p\) for which \(h(p)\) is a sub-pattern of \(g\): \(\mathcal{M}_{\leq}(p, g) = \{h : h(p) \sqsubseteq g\}\). One \(h(p)\) is sometimes called an embedding or an occurrence of \(p\) in \(g\). For example for graphs, \(\sqsubseteq\) is the subgraph relation and \(h\) is an isomorphism between the nodes, see Figure 1.

The goal of the matching operator is to enumerate all possible embeddings/occurrences \(h\). Often, the operator is used just to detect whether an embedding exists, that is: \(\mathcal{M}_{\leq}(p, g) \neq \emptyset\). An important concept is the cover of a pattern in a dataset, that is, the set of all transaction identifier that it matches:

\[
\text{cover}(p, D) = \{(\text{tid}, g) \in D | \mathcal{M}(p, g) \neq \emptyset\}
\]  

(1)

**Pattern mining formalization**

Using these operators, we can define, in line with Mannila and Toivonen [15], the generic constraint-based pattern mining problem as that of generating all canonical solutions that satisfy the constraints, that is:

\[
A = \{p \in \mathcal{G} | \mathcal{K}(p) = p \land C(p)\}
\]

**Standard frequent pattern mining** consists of enumerating all patterns that match at least a minimum number \(\theta\) of transactions:

\[
A = \{p \in \mathcal{G} | \mathcal{K}(p) = p \land |\text{cover}(p, D)| \geq \theta\}
\]

**Figure 1:** Example of two isomorphic graphs, when two syntactically different graphs are semantically equivalent
Pattern-extension formalization

While the above formalization is correct, it hides the fact that in all efficient mining algorithms, the pattern generation process grows new patterns by extending a smaller pattern. The main benefit is that one can then identify the frequent extensions, extensions leading to a frequent pattern, and avoid generating and testing patterns that will be infrequent.

The extension of a pattern of size \( l \) into a pattern of size \( l + 1 \) is made explicit through an extension operator \( \triangleright \). The operator \( \triangleright \) is such that \( p \triangleright e = p' \) with \( \text{length}(p') = \text{length}(p) + 1 \) and \( p' \supseteq p \).

We can now refine the generating operator using layers of patterns of a particular length. We introduce a parameterized generating operator \( G^l \), where \( l \) is the pattern size such that for each \( p \) in \( G^l \) it holds that \( \text{length}(p) = l \) and \( p \in \mathcal{G} \).

We can inductively define \( G^l \) with \( G^0 = \{ \emptyset \} \) as follows:

\[
G^{l+1} = \{ p' | p \in G^l \land p' = p \triangleright e \land \mathcal{K}(p') = p' \}
\]

Frequent extensions Before defining frequent extensions we first define the possible extension of a single transaction, transaction extensions \( TE(p, g) \) of a pattern \( p \) and a transaction \( (\text{tid}, g) \) \( \in D \), as follows:

\[
TE(p, g) = \{ e | h \in \mathcal{M}_{\supseteq}(p, g) \land h(p) \triangleright e \subseteq g \}
\]

We can then derive the set of frequent extensions \( FE \) as the set of all transaction extensions that are frequent wrt to a constant \( \theta \):

\[
FE(p, D, \theta) = \{ e | |\{(\text{tid}, g) \in D | e \in TE(p, g)\}| \geq \theta \}
\]

We can now inductively define \( G^l \) with frequent extensions as follows:

\[
G^{l+1} = \{ p' | p \in G^l \land e \in FE(p, D, \theta) \land p' = p \triangleright e \land \mathcal{K}(p') = p' \}
\]

This is the generation operator that is used in all state-of-the-art pattern mining algorithms. Also, when using declarative systems, we want to use the concept of frequent extensions as it is a key factor to scalability.

Pattern types

We now discuss for itemset, sequence and graph mining how these operators are typically implemented; both in state-of-the-art methods and in existing declarative methods for that pattern type.

Itemsets

An itemset pattern is simply a set of items, where we assume that the set of possible items \( I \) is known beforehand. The generating operator enumerates all \( 2^{|I|} \) subsets of \( I \), while the relation \( \subseteq \) of the matching operator is simply the subset relation (and the mapping \( h \) is the identity mapping and is unique for each \( p \) and \( g \)).

Machine representation

In highly efficient algorithms such as LCM [27] and Eclat [29], the itemset pattern has a machine representation of either a Boolean vector of size \( |I| \), or a sparse vector.

The generating operator has to avoid generating the same machine representation twice. Algorithms typically employ a pattern extension approach and assume an order \( r : I \rightarrow \mathbb{N} \) over the items. By only extending patterns with items that have a higher order than the highest ordered item in the pattern, one can avoid generating the same pattern twice. The generation operator is hence:

\[
G^{l+1} = \{ p \cup e | p \in G^l \land e \in I \land r(e) > \max_{i \in g} r(i) \}
\]

Furthermore, highly efficient algorithms use database projection to obtain the frequent extensions. Before extending a pattern, first all transactions that do not include the pattern are projected away. Then, the frequency of all remaining items (higher in the order than the current highest item) are counted and only the frequent ones are considered. More formally, the transaction extensions \( TE(p, g) \) of pattern \( p \) in transaction \( g \) are:

\[
TE(p, g) = \{ e | e \in g \land r(e) > \max_{i \in g} r(i) \}
\]

The set of frequent extensions and generator operator are given in Equations 3 and 4 and use the \( TE \) defined here. This can be done with a single scan over the database.

Declarative solvers

Constraint programming has been used to model and solve constraint-based itemset mining. Also ASP [11], SAT [10] and compilation to BDDs [5] has been used. Most work has happened using CP [7, 9], so we detail that formulation.

The CP representation of a pattern is a Boolean vector of size \( |I| \). The generating operator is the search strategy employed by the solver, which is depth-first search over the Boolean variables. This implicitly imposes an order on the variables and so no two identical patterns will be generated.

The \( \text{cover}(p, D) \) set is modeled explicitly as a Boolean vector of size \( |D| \) and whether a transaction is covered is expressed for each transaction separately: \( tid \in \text{cover}(p, D) \iff I \subseteq \text{trans}(tid) \) where \( \text{trans}(tid) \) is assumed to return the transaction with identifier \( tid \). Taking into account that \( I \) is represented as a Boolean vector, this constraint can be written as a reified linear sum over Booleans [7].

To achieve the same effect as database projection, the generating operator is not changed. In-
stead, the frequent extension property is expressed as an additional constraint and added to the con-
straining operator. The constraint states for each item individually that the item has to be frequent when projecting away transactions that are not covered: \( \forall i \in I : |\{tid \in cover(p, D) \land i \in trans(tid)\}| \geq \theta \). Indeed, every item in a fre-
cquent pattern will be frequent. However, during search this constraint will remove items that are in-
freqent given a partial solution, and hence those items will not be considered in the search, which corresponds to the generating operator with frequent extensions.

Because all operators including the frequent extension property can be expressed as regular con-
straints, declarative constraint based methods are very suited for this type of problem: stan-
dard search can be used and everything else is ex-
pressed as constraints.

### Sequence mining

In sequence mining, a pattern is a sequence of symbols taken from an alphabet \( \Sigma \). The matching operator is such that \( M\epsilon(p, g) = \{ h : h(p) \subseteq g \} \).

For sequences, \( h \) is called an embedding and it is a tuple of integer that maps the symbols at certain positions in a pattern to different positions. For example, embedding \( h = (2, 4) = (1 \mapsto 2, 2 \mapsto 4) \) maps sequence \( (a, b) \) to sequence \( (\epsilon, a, y, b, b) \) with the symbols at positions 1 and 3 undefined. The inclusion relation \( \subseteq \) verifies that the defined symbols of the source match the symbols at that position in the target pattern:

**Definition 1 (Sequence inclusion)** Let \( p = (p_1, \ldots, p_m) \) and \( g = (g_1, \ldots, g_n) \) be two sequences with sizes \( m \leq n \).

\[
p \subseteq g \iff \forall i \in 1, \ldots, m : p_i \neq \epsilon \rightarrow p_i = g_i.
\]

The inclusion relation is a key part of the matching operator. For example, one can verify given \( h = (2, 4) \) that \( h((a, b)) \subseteq (\epsilon, a, y, b, b) \) and so \( h \) is one embedding of the pattern in the transaction. Also, \( M\epsilon((a, b), (\epsilon, a, y, b, b)) = \{(2, 4), (2, 5)\} \).

#### Machine representation

Specialised algorithms represent the sequence as an ordered list of symbols, similar to the above formulation.

The use of frequent pattern extensions is very natural given all possible embeddings. However, in order to be more efficient the PrefixSpan \([23]\) algorithm includes the following observation: one must not consider all possible embeddings \( h \), in-
stead one should only verify the ‘earliest’ embedding, that is, the embedding \( h \) such that the last position \( \max_{h \in \mathcal{E}} l \) is minimal. In sequence mining, all symbols after this earliest last position are possible extensions. More formally, the transaction extension operator \( TE(p, g) \) is:

\[
\{ g_i \mid i > \min\{ \max(h) | h \in p\mathcal{E}(g) \} \}.
\]

It can be verified and collected with a single scan over the transaction, in contrast to enumerating all embed-
ings.

#### Declarative solvers

In many declarative con-
straint solvers, a finite-domain fixed-length repre-
sentation of the pattern is needed. In two related con-
straint programming formulations of sequence min-
ing \([18, 12]\), sequence patterns are defined as follows: let \( k \) be the length of the largest sequence (e.g. largest in the data), a sequence pattern is then modeled as an array of \( k \) symbols including a distinct ‘no symbol’ symbol \( \epsilon \). With this re-
presentation, the sequence \( (a, b, c) \) is represented as \( [a, b, c, \epsilon, \epsilon] \) for \( k = 5 \).

The size of a pattern sequence is simply the number of symbols that are not \( \epsilon \). So the size of \( [a, b, c, \epsilon, \epsilon] \) is \( 3 \).

The canonical form of a sequence pattern is such that \( \epsilon \) symbols may only appear at the very end of the pattern, i.e. \( R([a, \epsilon, b, \epsilon]) = [a, b, c, \epsilon] \).

The main challenge is in implementing the matching operator used in the cover relation. In logical form, \( M(p, g) \neq \emptyset \) conforms to \( \exists h : h(p) \subseteq g \). However, the number of possible em-
beddings \( h \) is exponential, so this constraint can not efficiently be expressed in propositional logic.

Three alternatives have been proposed in the con-
straint programming literature: in \([16]\) the ap-
proach is to encode each transaction as a (non-) de-
terministic finite automaton and use the automa-
ton constraint to verify that an embedding exists. In \([18]\) one proposal is to expose for each trans-
action the embedding \( h \) as a separate set of vari-
ables, and to do a separate existential search af-
after all pattern variables are assigned. Also in \([18]\) it is proposed to hide the matching operator in a global constraint. Furthermore, this global con-
straint implements the same linear-scan technique as pioneered by PrefixSpan, and can also export the transaction extensions as explained earlier.

These choices have trade-offs: the DFA ap-
proach is a pure constraint formulation and most generic but does not support frequent extensions and is least scalable; the existential search ap-
proach requires non-trivial changes to the solver search to avoid side-effects; and the global con-
straint is most scalable but constraining the match-
ing operator (max-gap, max-span) would require re-implementing the global constraint.

#### Graphs

In graph mining, a pattern is a set of labeled edges, i.e. a graph. Each labeled edge is a tuple \( (v_1, v_2, a_1, a_2, a) \) where \( v_1, v_2 \) are vertices of the edge, \( a_1, a_2 \) their labels and \( a \) is the label of the edge itself. The length is the number of edges
in a pattern, i.e. the size of the graphs in Figure 1 is four. The generating operator generates all possible subgraphs of the graphs in $D$. The upper bound on the number of edges is the length of the largest graph in $D$.

For graphs, the embedding $h$ is a graph isomorphism, that is, a label preserving bijection of the vertices, and the inclusion relation $\subseteq$ is the subgraph relation. Then, for a pattern $p$ and a graph $g$ the matching operator $M_{\subseteq}(p,g)$ is equal to the set of all subgraph isomorphisms from $p$ to $g$.

**Machine representation** Specialised algorithms such as gSpan [28] and Gaston [21] represent a graph as a sequence of labeled edges $(a_1, a_2, a)$ and use a pattern extension approach.

The generating operator must ensure that only connected graphs and no two isomorphic graphs are generated. Doing an isomorphism check between the generated pattern and all previously generated patterns would be overly expensive. Instead, a canonical form of a graph can be used. gSpan [28] proposes the use of a DFS code to verify canonicity: a graph can be traversed using depth-first search in multiple ways and each traversal imposes a linear order over the edges. Together with an order over the labels, one can search for the traversal that leads to the minimal DFS code. Isomorphic graphs can be proven to have the same minimal DFS code. Hence, if for the current pattern there exists a DFS code smaller than the DFS code induced by its sequence of edges representation, we know that the current pattern is not canonical and should be discarded.

**Frequent extensions** Checking subgraph isomorphism between a pattern and each transaction is very expensive, hence it pays to consider only the frequent extensions. To collect the frequent extensions, one should not just check that an embedding exists ($M(p,g) \neq \emptyset$) but rather to enumerate all embeddings and to collect the possible extensions $\{(a_1, a_2, a)\}$ of these embeddings (Equation 2). Some methods store the possible embeddings per transaction (occurrence lists), this is a memory/computation trade-off.

**Declarative solvers** Graph mining with declarative solvers has not been investigated much, with the exception of the use of inductive logic programming where there is a whole body of work on learning rules (patterns) from relational data [13, 14, 17].

Looking at constraint-based solvers, we only know of recent work [22] where an FO(.) system is used. The generating operator is simplified by assuming a template pattern that the pattern needs to be a subgraph of; this template can be inferred from the data and typically domain-specific. The matching operator is implemented as a separate call to the FO(.) solver to check for each transaction whether it is subgraph isomorphic to the pattern. Canonicity is verified by checking whether there is a lower ordering for this pattern in the template.

Compared to sequences, the main challenge in graph mining is not only the matching operator/subgraph isomorphism, but also to verify canonicity. The main problem is that the whole graph mining process can not be expressed in first order logic. We see three possible ways in which declarative solvers can be used for graph mining nonetheless:

The first is to write the function of each operator as a separate declarative program (e.g. subgraph isomorphism check, canonicity check). Then, we can devise an algorithm that calls these operators/programs as needed. An example is given in Algorithm 1. This is close in spirit to what was done recently for graph and query mining [22].

Another approach is to use a higher order logic, e.g., matching of a pattern $p$ in a dataset $D$ is a second order construction $\forall \text{tid} : \text{tid} \in \text{cover}(p,D) \leftrightarrow \exists h : h(p) \to g$ (simplified statement). That might be executed in an SMT solver that would be able to solve subproblems such as finding constrained functions (second order existential quantifiers) and propagate such constraints.

Finally, another possibility is to hide the higher order complexity in global constraints in CP. Once can implement the DFS code computation, and the isomorphism check (+frequent extension enumeration) as separate global constraints. Together with the ‘graph as sequence of labeled edges’ representation, this would be close in spirit to the sequence mining in CP approach [18].

**Constraints & dominance relations**

We discriminate between constraints added to the constraining operator and constraints that modify the matching operator. We also consider preferences over the set of all solutions through the dominance operator.

**Constraining operator** These constraints apply to each individual pattern. Examples include

**Algorithm 1: Naive graph mining solver**

```plaintext
Specify: $\mathcal{M}$, $\mathcal{K}$, $C$, $\cup$
Input : $D$ Output : $ps$
$ps \leftarrow \emptyset$;
for $l \in 0 \ldots l_{max}$ do
    $g^l \leftarrow \text{generate}(ps, \mathcal{M}, \mathcal{K}, \cup, D)$  \text{Eq. 4}
    for $g \in g^l$ do
        if $C(g)$ then $ps \leftarrow ps \cup \{g\}$;
end
```

---

**Declarative solvers** Graph mining with declarative solvers has not been investigated much, with the exception of the use of inductive logic programming where there is a whole body of work on learning rules (patterns) from relational data [13, 14, 17].

Looking at constraint-based solvers, we only know of recent work [22] where an FO(.) system is used. The generating operator simplifies by assuming a template pattern that the pattern needs to be a subgraph of; this template can be inferred from the data and typically domain-specific. The matching operator is implemented as a separate call to the FO(.) solver to check for each transaction whether it is subgraph isomorphic to the pattern. Canonicity is verified by checking whether there is a lower ordering for this pattern in the template.

Compared to sequences, the main challenge in graph mining is not only the matching operator/subgraph isomorphism, but also to verify canonicity. The main problem is that the whole graph mining process cannot be expressed in first order logic. We see three possible ways in which declarative solvers can be used for graph mining nonetheless:

The first is to write the function of each operator as a separate declarative program (e.g. subgraph isomorphism check, canonicity check). Then, we can devise an algorithm that calls these operators/programs as needed. An example is given in Algorithm 1. This is close in spirit to what was done recently for graph and query mining [22].

Another approach is to use a higher order logic, e.g., matching of a pattern $p$ in a dataset $D$ is a second order construction $\forall \text{tid} : \text{tid} \in \text{cover}(p,D) \leftrightarrow \exists h : h(p) \to g$ (simplified statement). That might be executed in an SMT solver that would be able to solve subproblems such as finding constrained functions (second order existential quantifiers) and propagate such constraints.

Finally, another possibility is to hide the higher order complexity in global constraints in CP. Once can implement the DFS code computation, and the isomorphism check (+frequent extension enumeration) as separate global constraints. Together with the ‘graph as sequence of labeled edges’ representation, this would be close in spirit to the sequence mining in CP approach [18].

**Constraints & dominance relations**

We discriminate between constraints added to the constraining operator and constraints that modify the matching operator. We also consider preferences over the set of all solutions through the dominance operator.

**Constraining operator** These constraints apply to each individual pattern. Examples include
minimum/maximum frequency, minimum/maximum pattern length, inclusion/exclusion of pattern elements, weighted sums over pattern elements or transactions, etc. Sequences can furthermore be subject to regular expression constraints [3]. Graphs can have minimum/maximum degree constraints or constraints on the topology such as having cycles.

**Matching operator.** Some constraints change the definition of what a valid embedding $h$ is. This is not applicable to itemsets ($h$ is unique). For sequences, the best example is a constraint on the maximum gap in the matching between two subsequent symbols. For example, $(a, b)$ is included in $(a, c, b)$ with a maximum gap of 1 but not in $(a, c, c, b)$ where $h = (1, 4)$ has a gap of 2. For graphs, one can consider approximate matchings, where node-edge insertions and deletions are allowed [20]. Also applicable is the case where the frequency is defined by total number of occurrences across all transactions: $\sum_{(p_1, g) \in D} M(p, g)$. Dominance operator. Some constraints are over all pairs of possible solutions, e.g. that there exists no other valid pattern such that... [19]. Constraints that fall in this category are maximality (no super pattern is also frequent), closedness (no super pattern with same frequency) and relevant subgroup discovery (no pattern with higher frequency on one part of the data and lower on the other). In general, these constraints impose a preference (a preorder) over the solutions.

**Machine representation** If possible, it is typically better to push constraints into the generating operator so that patterns that do not satisfy the constraints aren’t generated in the first place. This is the case for frequency and frequent extensions for example. Another example is the maximum length constraint which can be pushed in the generating operator $G^{i+1}$ as follows:

$$\{ p' \mid p \in G^i \land length(p) < s \land p' = p \sqcup \_ \land \chi(p') = p' \}$$

This approach is correct for all anti-monotonic constraints $c$, that is, if for any two patterns $p$ and $p'$ s.t. $p \subseteq p'$ it follows that $c(p) \rightarrow c(p')$.

Other constraints cannot be pushed in the generating operator as extensions in $G^i$ not satisfying them may be still be in $G^{i+x}$ for $x > 0$.

Some dominance constraints can be formulated to be a part of the constraining operator, for example closedness and maximality for itemsets [27]. However, in general they can not and algorithms maintain a repository of previously found solutions and check each new solution to the ones in the repository.

**Declarative solvers** The constraining operator naturally fits declarative solvers. Pushing constraints in the generating operator is typically not possible, but the propagation mechanisms of the constraint-based solvers might have the same effect (e.g. for max size this is the case).

Generic methods for adding constraints to the matching operator are an open problem when using declarative solvers, indeed, the encoding of the matching operator for sequences and graphs is challenging in itself (see previous section).

The dominance operator on the other hand has a natural solution method by solving a chain of satisfaction problems (e.g. a dynamic CSP [24]). A generic framework for dominance relations and itemset mining using CP exists [19].

**Related works**

*Data Mining Template Library (DMTL)* [1] has been proposed as a unifying framework for mining tasks. It aims to define basic building blocks (in C++) that can be combined to build any mining algorithm. Gaston [21] is a pattern mining algorithm designed for both sequences and graphs in the presence of only a coverage constraint. Generic constraint-based imperative methods have been studied as well [25] [4].

Building on more theoretical grounds, [3, 20] proposed to formalise the problem of mining patterns as the problem of mining closed sets in a set system. This generic framework can be used to solve a variety of mining tasks ranging from itemset mining problems to simple instances of graphs under constraints.

De Raedt et al. [6] have proposed constraint programming as a unifying framework for itemset mining. This has been extended to structured mining tasks [18, 12, 16] as well as dominance relations [19].

**Conclusions**

We investigated the key components of structured pattern mining tasks through a formal framework. Using that, we reviewed how specialised methods implement these operators and how they can be formulated in declarative constraint solvers.

We showed how itemset mining fits a declarative approach especially well, as its matching and canonical operator are trivial to express. This is not the case for sequence and graph mining, and we discussed possible solutions to this.

In the future, we expect higher-order languages and hybridisations between solvers and specialised methods/operators to further boost declarative pattern mining. How to maintain generality in hybrid approaches is an open question though.
References


