HAL
open science

Practical Computing with Pattern Structures in FCART Environment

Aleksey Buzmakov, Alexey Neznanov

To cite this version:

Aleksey Buzmakov, Alexey Neznanov. Practical Computing with Pattern Structures in FCART Environment. Workshop FCA4AI, "What FCA can do for artificial intelligence?", Aug 2013, Beijing, China. hal-00910296

HAL Id: hal-00910296

https://inria.hal.science/hal-00910296v1

Submitted on 28 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Practical Computing with Pattern Structures in FCART Environment

Aleksey Buzmakov[1,2] and Alexey Neznanov[2]

[1] LORIA (CNRS – Inria NGE – U. de Lorraine), Vandœuvre-lès-Nancy, France
[2] National Research University "Higher School of Economics", Moscow, Russia
aleksey.buzmakov@inria.fr, aneznanov@hse.ru

Abstract. A new general and efficient architecture for working with pattern structures, an extension of FCA for dealing with "complex" descriptions, is introduced and implemented in a subsystem of Formal Concept Analysis Research Toolbox (FCART). The architecture is universal in terms of possible dataset structures and formats, techniques of pattern structure manipulation.

Keywords: Formal Concept Analysis, Pattern Structures, Software

Introduction

FCART[1] is a specialized software for data analysis by means of Formal Concept Analysis (FCA) and related methods aiming at processing an arbitrary dataset [1]. FCA processes a binary context to a concept lattice, which can be very useful for "gold mining" – obtaining a new knowledge. However, datasets are unlikely kept in the binary way where an object is described as a set of binary attributes it possesses. To deal with this problem different kinds of scalings can be applied to a dataset, converting it to a binary context. In some cases it can be slow or meaningless. Pattern structures (PSs) is an extension of FCA dealing with "complex" data [2]. However, just a couple of applications of PSs are available for the community and, moreover, neither of them are able to work with an arbitrary PS. Thus, we introduce a generalized approach to PSs within FCART.

The paper is organized as follows. Section 1 defines FCA and PSs. The next section describes the overall PS processing within FCART, divided into logical submodules of the approach. Finally, the paper is concluded before program interfaces of different modules are given.

1 FCA and Pattern Structures

Formal concept analysis (FCA) [3] is a mathematical formalism having many applications in data analysis. It process a binary context (a triple (G, M, I)

[1] `http://ami.hse.ru/issa/Proj_FCART`

where G is a set of objects, M is a set of attributes and $I \subseteq G \times M$ is a relation between them) into a concept lattice. Pattern structures (PSs) is a generalization of FCA for dealing with complex structures, such as sequences or graphs [4]. As it is a generalization it is enough to introduce only PSs.

Definition 1. *A PS is a triple $(G, (D, \sqcap), \delta)$, where G is a set of objects, $(D, \sqcap)$ is a complete meet-semilattice of descriptions and $\delta : G \to D$ maps an object to the description.*

The lattice operation in the semilattice ($\sqcap$) corresponds to the similarity between two descriptions d_1 and d_2, i.e. the description which is common between d_1 and d_2. Standard FCA can be presented in terms of PSs in the following way. The set of objects G remains, while the semilattice of descriptions is $(\wp(M), \cap)$, where $\wp(M)$ is a powerset of M, and, thus, a description is a set of attributes. The similarity operation corresponds to the set intersection, i.e. the similarity is the set of common attributes. If $x = \{a, b, c\}$ and $y = \{a, c, d\}$ then $x \sqcap y = x \cap y = \{a, c\}$. The mapping $\delta : G \to \wp(M)$ is given by, $\delta(g) = \{m \in M \mid (g, m) \in I\}$.

The Galois connection for a PS $(G, (D, \sqcap), \delta)$ between the set of objects and the semilattice of descriptions is defined as follows:

$$A^\diamond := \bigsqcap_{g \in A} \delta(g), \qquad \text{for } A \subseteq G$$

$$d^\diamond := \{g \in G \mid d \sqsubseteq \delta(g)\}, \qquad \text{for } d \in D,$$

where the partial order (or the subsumption order) on D is defined w.r.t. the similarity operation $\sqcap$: $c \sqsubseteq d \Leftrightarrow c \sqcap d = c$, and c is subsumed by d.

Definition 2. *A pattern concept of a PS $(G, (D, \sqcap), \delta)$ is a pair (A, d) where $A \subseteq G$ and $d \in D$ such that $A^\diamond = d$ and $d^\diamond = A$, A is called a concept extent and d is called a concept intent.*

As in the standard case of FCA, a pattern concept corresponds to the maximal set of objects A whose description subsumes the description d, while there is no $e \in D$, subsuming d, i.e. $d \sqsubseteq e$, describing every object in A. The set of all concepts can be partially ordered w.r.t. partial order on the extents (dually, the intents by $\sqsubseteq$), within a concept lattice.

Example 1. PSs are successfully used for interval data [5]. For example, in gene expression data every gene is described by its expression value in different situations. The meet-semilattice $(D_{ips}, \sqcap_{ips})$ includes vectors of intervals. An example of an interval PS is given by δ-function in Table 1. The description of g_1 is $g_1^\diamond = \langle [1, 3]; [3, 5]; [2, 4] \rangle$. The description materializes the fact that the gene expression in situations m_1, m_2, m_3 are within the corresponding intervals. The similarity operation ($\sqcap_{ips}$) between two interval descriptions $g_1^\diamond$ and $g_2^\diamond$ is the component-wise convex hull of intervals. Thus, $g_1^\diamond \sqcap g_2^\diamond = \langle [1, 7]; [3, 6]; [2, 5] \rangle$. The interval pattern concept lattice resulting from this PS is shown in Figure 1 (* is a special description subsuming anything).

Example 2. Given a dataset with objects described by elements of poset P, e.g. sequences (w.r.t sequence-subsequence relation) or graphs (w.r.t. subgraph isomorphism relation), a corresponding PS can be defined in the following way. The semilattice $(D, \sqcap)$ based on poset P is a subset of the powerset of P, $D \subseteq \wp(P)$, such that if $d \in D$ contains an element $p \in P$ then all its "subelements" x should be included into d, $\forall p \in d, \nexists x \leq p : x \notin d$, and the semilattice operation is the set intersection for two sets of elements. Given two patterns $d_1, d_2 \in D$, the set intersection operation ensures that if an element p belongs to $d_1 \sqcap d_2$ then any subsequence of p belongs to $d_1 \sqcap d_2$ and, thus, $(d_1 \sqcap d_2) \in D$.

However, the set of all possible "subelements" for a given pattern can be rather large. Thus, it is more efficient and representable to keep a pattern $d \in D$ as a set of all maximal elements $\tilde{d}$, $\tilde{d} = \{p \in d \mid \nexists x \in d : x \geq p\}$. Note that representing a pattern by the set of all maximal elements allows for an efficient implementation of the intersection "$\sqcap$" of two patterns.

	m_1	m_2	m_3
g_1	$[1,3]$	$[3,5]$	$[2,4]$
g_2	$[5,7]$	$[4,6]$	$[2,5]$
g_3	$[1,9]$	$[2,7]$	$[6,6]$

Table 1: An Interval PS.

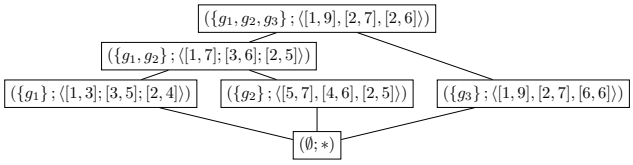

Fig. 1: The concept lattice for the PS in Table 1.

PSs can be hard to process due to the usually large number of concepts in the concept lattice and the complexity of the similarity operation (think for instance of the graph isomorphism problem). Moreover, a pattern lattice can contain a lot of irrelevant patterns for an expert. Projections of PSs "simplify" to some degree the computation and allow one to work with a reduced description. In fact, projections can be considered as constraints (or filters) on patterns respecting certain mathematical properties, ensuring that the concepts in the projected lattice have correspondence to the original ones [4].

A projection $\psi : D \to D$ is an operator, which is monotone ($x \sqsubseteq y \Rightarrow \psi(x) \sqsubseteq \psi(y)$), contractive ($\psi(x) \sqsubseteq x$) and idempotent ($\psi(\psi(x)) = \psi(x)$). A projection preserves the semilattice operation $\sqcap$ as follows. Under a projection ψ, a PS $(G, (D, \sqcap), \delta)$ becomes the projected PS $\psi((G, (D, \sqcap), \delta)) = (G, (D, \sqcap), \psi \circ \delta)$. The concepts of a projected pattern structure have a "similar" concept in the initial pattern structure [4].

2 Pattern Structures Techniques

As a PS is an abstract mathematical object, any software aiming at the PS realization should either prepare several different PSs, such as PSs based on intervals or graphs, or give to a user an opportunity to add arbitrary PSs to the software. Our goal is to process any PSs and in this case one should decide how an arbitrary semilattice can be introduced by a user. It is not possible in some cases to enumerate all elements of a semilattice. For example, the semilattice of

```
Function CloseByOne(Ext, Int)
    Data: (G, (D, ⊓), δ), extent Ext and intent Int of a concept.
    Result: All canonical ancestor concepts of the concept (Ext, Int).
    foreach S ⊆ G, S ≻ Ext do
        NewInt ⟵ ⨅_{g∈S} δ(g) ;                          /* ⊓ - intersection */
        NewExt ⟵ {g ∈ G | NewInt ⊑ δ(g)};                /* ⊑ - subsumption */
        if IsCanonicExtension(Ext, NewExt) then
            SaveConcept((NewExt, NewInt));
            CloseByOne(NewExt,NewInt);
CloseByOne(∅, ⊤);                                       /* Find all concepts... */
```

Algorithm 1: The modified version of CbO for PS processing.

graphs is infinite and even if one would like to select a finite subset of it, the subset should be significantly large in order to be useful in real-life applications. Another option is the constructive way for defining a semilattice, i.e. one should be able to keep any element of the given semilattice, to compute the semilattice operation between two elements of the semilattice and to check equality of two elements. Although the subsumption relation on a semilattice can be checked as $c \sqsubseteq d \Leftrightarrow c \sqcap d = c$, in many cases it can be more efficient to check the subsumption relation directly. Later we discuss how semilattices are processed more carefully.

But how can we build a concept lattice from a given PS? Many state-of-the-art algorithms can be slightly modified in such a way that avoid enumeration of attributes, i.e. performing only the set intersection operation and checking the subset relation without naming the attributes. This modification allows to further substitute the set intersection by the corresponding semilattice operations and to compute the concept lattice from a PS. Algorithm 1 shows the listing of the modified `CbO` [6] algorithm. Moreover, modified algorithms can easily process standard FCA by introducing the described above powerset semilattice. Since PSs can be processed with a number of different algorithms, we should allow to a user to introduce any algorithms he wants.

The following parts, called plugins, are introduced in FCART:

- A constructive semilattice description;
- Extent and Intent storages, managing extents and intents of a lattice;
- A concept lattice builder working with any available semilattices.

Now we can build a concept lattice from any PSs, but we still do not know how to process the different element nature of a semilattice, i.g. how to load or save it. These problems are discussed in the following subsection as well as the processing of projections of PSs.

2.1 Input and Output Data Formats

For the purposes of keeping and exchanging of patterns format JSON is chosen because it allows to serialize nearly any kind of data, is standardized [1], has low

[1] `http://www.json.org/`

```
{
    "Count": 3,
    "Inds":[2, 5, 8]
}
```

(a) Indices array.

```
{
    "Count": 3,
    "Inds"::[2.3, 5.5, 8.1]
}
```

(b) Real numbers array.

```
{[
    { "NodesCount" : 2 },
    { "Nodes" : [
        { "Int" : 0, "Ext" : 0 },
        { "Int" : 1, "Ext" : 1 }
    ]},
    { "ArcsCount" : 1 },
    { "Arcs" : [
        { "S" : 0, "D" : 1 },
    ]},
]}
```

(c) Concept Lattice.

Fig. 2: JSON formats for object and semilattice element descriptions.

parsing overhead, and is more compact than XML. We introduce the following general datatypes: primitives (numbers, strings), sets, ordered sets, rooted trees and general structures, i.e. graphs. *But what kind of data we need to process?* First a dataset from an external source should be converted to JSON and put into an internal collection. This imported dataset corresponds to a δ-function for a PS. Since the target semilattice can be a projection, the descriptions in this semilattice can be different from the descriptions in the imported dataset. For example, an object description can be a graph, while the projection can be a chain which can be kept in more efficient and tractable structure than a general graph. Thus we have two datatypes, one is used for a δ-function and the second is for a semilattice object. To allow for a plugin work with only the descriptions this plugin can work, the plugin specifies the external and internal datatypes by unique ID of that datatype. Figure 2 exemplifies indexes array, which can be used to keep sets, and numbers array, which can be used as the initial description of interval PS.

The next entity for exchanging between FCART and a plugin is a concept lattice. In our case it is a set of concepts with several edges. The concepts extents and intents are referred by special indexes, which come from extent and intent storages. The simple lattice is exemplified in Figure 2c.

Finally, a plugin can have its own running settings, which are given in an arbitrary JSON. For example, this properties allows us to realize a class of projections rather than a given projection, i.g. the projections of a graph to all its subgraphs of no more then k vertices, where k is a parameter of the plugin.

2.2 Pattern Manager Plugin

A semilattice $(D, \sqcap)$ is given in the constructive way by a plugin called "Pattern Manager". The main operations which should be performed by this plugin are listed in Table 2. The first two properties are description types the plugin can load from a dataset or process as patterns. Patterns here refers to an internal data format of patterns known only by this pattern manager. Loading patterns from a given JSONs, patterns can be intersected or compared. This allows to give a semilattice in the constructive way without enumerating all possible elements of a

lattice. Any patterns can be saved in a JSON of a 'GetPatternType()' type. And, finally, there are three functions which can creates patterns. To remove a pattern and clear the memory of this pattern, function 'FreePattern' is introduced.

2.3 Extent and Intent Storage Plugins

Although Pattern Manager can create a lot of patterns by the intersection or the loading operations, it is not responsible for memory it creates. Plugin 'Intent Storage' is a special layer which separates the raw representation of a pattern (an output of a Pattern Manager) and the IDs of intents, which are used in a lattice builder. Moreover, all patterns should pass through an Intent Storage and thus it controls memory for patterns. Intent Storage is responsible for the indexes it creates and, thus, it can be (de)serialized in a unified way in order to preserve the intents between sessions. Finally, as Intent Storage translates some of its call to Pattern Manager, we should initialize Intent Storage by the required Pattern Manager. The functions of Intent Storage are the same as for Pattern Manager but it should be initialized with a Pattern Manager and can be (de)serialized.

Plugin 'Extent Storage' is an analog of Intent Storage but for the extents. We know exactly what an extent is, and, thus, the additional layer 'Extent Manager' is not necessary. To work with extents in the bottom to top strategy we usually do not require any intersection operations and just add objects to this set. The interface functions of Extent Storage plugin are shown in Table 3.

2.4 Lattice Builder Plugin

Finally, to build a pattern concept lattice, a special plugin "Lattice Builder" is introduced. Lattice Builder takes as an arguments Extent and Intent storages and the path where the result lattice in the describing above format should be saved. It has three functions: 'Initialize()' taking Extent and Intent Storage plugins; 'AddObject(ObjID, IntentID)' taking a unique ID of an object which should be added to the context with the corresponding description given by its ID; and finally 'Build()' building or postprocessing a lattice and writing it to the required file. We should remember that there are two types of algorithms for building a concept lattice: incremental such as AddIntent [7], where after each addition of an object the new lattice is constructed, and non-incremental such as CbO [6], where the lattice is constructed for all objects at once. The function AddObject can be used to construct a lattice in an incremental way by the first algorithms or to collect a context by the algorithms from the second group.

2.5 Organization of Plugins

To allow the efficient implementation of any plugins, they are kept in a dynamic link library with a special API, which is called "Plugin System API". This library should contain at least three functions listed in Table 4. Function 'GetDescription' return a JSON array [2], with description of every plugin that can be found

[2] JSON is selected for plugin data representation by the previously mentioned reasons.

in the plugin system. The description contains unique ID of a plugin, type of the plugin, i.e. Patten Manager, Extent or Intent storage, or LatticeBuilder. According to the type of the plugin it contains the map from a plugin functions to the functions realized in the library.

Every plugin in a system should implement the functions listed in Table 5. Which allows to use them in a generalized way. The plugin can describe its parameters, for example the size of graph in a projection, and then load them and save them in a described JSON format. Finally, as some plugins can be initialized by other plugins, a plugin can request another plugin in its description. The initializing plugin is given to the requester as an ID and FCART has API for requesting an address of a plugin be the plugin ID and the function name.

Conclusion

Pattern structures is a very general and powerful technique for knowledge extraction form complex object descriptions with perspectives for working with Big Data. We implemented PSs within an original framework in an efficient and universal way. Current prototype is presented in FCART software and justifies the approach. The project is improving taking into account benchmark and profiling results and requirements of researches.

Acknowledgements: this research received funding from the Basic Research Program at the National Research University Higher School of Economics (Russia) and from the BioIntelligence project (France).

References

1. Neznanov, A.A., Ilvovsky, D.A., Kuznetsov, S.O.: FCART: A New FCA-based System for Data Analysis and Knowledge Discovery. In: Proc. of workshop for FCA Tools and Applications (at ICFCA'2013). (2013)
2. Kuznetsov, S.: Fitting Pattern Structures to Knowledge Discovery in Big Data. In Cellier, P., Distel, F., Ganter, B., eds.: Formal Concept Analysis SE - 17. Volume 7880 of Lecture Notes in Computer Science. Springer (2013) 254–266
3. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. 1st edn. Springer, Secaucus, NJ, USA (1997)
4. Ganter, B., Kuznetsov, S.O.: Pattern Structures and Their Projections. In Delugach, H., Stumme, G., eds.: Conceptual Structures: Broadening the Base SE - 10. Volume 2120 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 129–142
5. Kaytoue, M., Kuznetsov, S.O., Napoli, A., Duplessis, S.: Mining gene expression data with pattern structures in formal concept analysis. Information Sciences **181**(10) (2011) 1989 – 2001
6. Kuznetsov, S.O.: A fast algorithm for computing all intersections of objects in a finite semi-lattice. Automatic documentation and Mathematical linguistics **27**(5) (1993) 11–21
7. Merwe, D.V.D., Obiedkov, S., Kourie, D.: AddIntent: A new incremental algorithm for constructing concept lattices. In Goos, G., Hartmanis, J., Leeuwen, J., Eklund, P., eds.: Concept Lattices. Volume 2961. Springer (2004) 372–385

Appendix

Function	Description
ID=GetObjectType()	Returns the JSON type of an object
ID=GetPatternType()	Returns the JSON type of the patterns it works with
Pttrn=Preprocess(JSON)	Loads JSON description of an object and converts it to the internal pattern type
$Pttrn = a \sqcap b$	Computes semilattice operation between patterns a and b.
$\{True, False\}$=$a \sqsubseteq b$	Checks if pattern a is subsumed by pattern b.
$\{True, False\}$=$(a == b)$	Checks if one pattern is equal to another pattern.
Pttrn=LoadPattern() JSON=SavePattern(a)	Convert the internal pattern to/from the JSON with type GetPatternType().
FreePattern(a)	Free memory allocated for pattern a

Table 2: Main functions of Pattern Manager plugin API.

Function	Description
ID=Clone(ID)	Clones the extent with ID. ID==-1 is a special empty extent
AddObject(ID, objID)	Add the object objID to the extent ID
Size=Size(ID)	Returns the number of objects in the extent ID
ObjID= LastAddedObject(ID)	Returns the last added object to the extent ID.
ID=LoadExtent(JSON) JSON=SaveExtent(ID)	Loads and saves the extent ID form/to JSON.

Table 3: Main functions of Extent Storage plugin API.

Function	Description
Init()	Initialization of a library
Done()	Deinitialization of a library
JSON=GetDescription()	Returns the description of all the plugins that the given plugin system support

Table 4: Functions of Plugin System API.

Function	Description
Create()	Creation of a plugin object
Destroy()	Destruction of a plugin object
DescribeParams()	Describes what kind of params the plugin can have in both human-readable and machine readable forms
Load/Save Params()	Load or save params in the form described by 'DescribeParams'

Table 5: Main functions of common plugin API.