

A Python Library for FCA with Conjunctive Queries

Jens Kötters

Abstract. The Python library presented in this paper implements a graph-based formalization of conjunctive queries which supports visualization and algebraic operations. The library also supports the generation of concept lattices, where concepts described by conjunctive queries rather than attribute sets as in standard FCA. Concept lattices are defined over power context families instead of formal contexts. Power context families are sequences of formal contexts introduced by Wille[6] which describe relational data. An interactive Jupyter notebook session is presented and used to illustrate core API features. Intension graphs, power context families and morphism tables have visualizations as SVG or HTML provided by the library and supported by the notebook.

Keywords: Conjunctive Queries, Power Context Families, Python Library, Concept Lattices

1 Introduction

In Formal Concept Analysis[3], a concept is mathematically represented by a pair (A, B) of sets. The *extent* A is a set of formal objects (the instances of the concept), and the *intent* B is a set of formal attributes (the meaning of the concept). Extent and intent formalize the philosophical notions of *extension* and *intension*. Whereas extension is modeled quite naturally as a collection of things, the notion of intension is less tangible and although prevalent definitions refer to collections, there are other ways to formalize meaning.

When communicating the meaning of concepts in everyday life, we make use of sentences (in our respective languages). Oftentimes, the meaning is too complex to be conveniently or precisely expressed by an accumulation of attributes representing simple properties. Formal languages like formal logic or Conceptual Graphs are better suited to capture such complexity and correspond more directly to the natural languages we use to make meaning accessible. In this regard, we may consider formal languages for the mathematical representation of intension, and thus as the basis for an alternative formalization of concepts. Concepts with non-standard intents have previously been introduced on a general basis by Ganter and Kuznetsov[2], which allows intents to be taken from an arbitrary semilattice (as far as finite object sets are concerned). The semilattice elements are called *patterns*, and the semilattice itself is defined as part of a *pattern structure*, which roughly corresponds to a formal context and determines

the concept lattice. The theory of pattern structures has been applied to various application domains. This paper is concerned with a special case of pattern structures where intents are attribute-labeled graphs with a designated node.

Figure 1 shows the *granddaughter* graph as an example. The nodes of the graph represent unspecified persons (the letters A, B and C are node IDs and can be considered variable names). The lowest node carries a *female* label. The two edges carry *parent* labels and denote that the source is a parent of the target. The graph describes a situation, and the colored node is designated (i.e. subject of the description). The graph thus describes the concept of all granddaughters. In terms of expressivity, these graphs correspond to the class of *conjunctive*

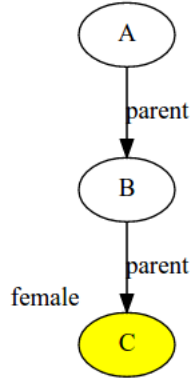


Fig. 1. Granddaughter pattern

queries well-known in the theory of databases. Equivalently, they correspond to logical formulas built from atoms using only conjunction (\wedge) and existence quantification (\exists). In[4], concept lattices have been defined based on a representation of conjunctive queries by *windowed* relational structures (the window marks the designated elements), and [5] introduces *windowed intension graphs* as a graph-based formalization which directly corresponds to the visual representation (as in Fig. 1). Despite the connection to database theory, the approach described here is meant to be application-specific; it is meant to be a general purpose mathematical model of real-world concepts.

The Python library presented in this paper consists roughly of two parts. The first part consists mainly of the `IntensionGraph` class, which implements conjunctive queries and features algebraic operations on graphs in its API. The `PowerContextFamily` implements Willes power context families [6] as a relational data model wherein intension graphs can be realized. The `IntensionGraph` class supports graphs with an arbitrary number of designated elements. The second part of the library implements pattern structures and is parameterized by the `IntensionGraph` class through a well-defined in-

terface. This part of the implementation currently only supports graphs with a single designated element.

Section 2 documents the correspondence between intension graphs and logical formulas. Section 3 focuses on the morphism (pre-)order on intension graphs. Addition and multiplication of graphs are presented in these sections. Section 4 covers power context families, and Sect. 5 focuses on the concept lattice. The family tree example of [4] (Figs. 1 and 2) will be reused as an example in Sects. 4 and 5.


2 Primitive Positive Formulas

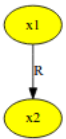
In what follows, we will present the API functions in the context of an interactive console session (Figs. 2, 3, 4, 5 and 6) in the Jupyter notebook (formerly IPython). The labels `In[n]` and `Out[n]` in these figures denote the input and output of the n -th interaction, and these shall be referred to as *cell n* . Cell 1 imports the library, thus making API functions accessible in the notebook. The intension graph shown in cell 2 represents the atom $m(x)$. The intension graph in cell 3 represents the atom $R(x1, x2)$. Intension graphs implement a `_repr_svg_` method which returns an SVG representation of a graph. The Jupyter notebook supports this method by rendering the SVG when a graph expression is input. Rendering does not take place when the expression is part of a statement, so to produce a graphic in cell 3, we repeat the expression after the assignment. The sum of intension graphs represents the conjunction of formulas (cell 4). Using these operations, we can produce any conjunction of atoms. To apply existential quantification, we call the `windowed` function with an arbitrary number of key-value-arguments, where every node whose ID is *not* listed as a value is considered existentially quantified, and the keys are alias names for the free variables (aliases are never confused with IDs, so there is no name collision when an alias equals some nodeID). In the SVG output, the yellow nodes are free and the white nodes are existentially quantified. When adding two graphs, the library may change node IDs to ensure unique node IDs in the sum; this is achieved by prefixing node IDs with operand indices (cf. cell 6). Node IDs do not have to resemble variable names; arbitrary nonempty strings can be chosen. The current implementation does not show alias names in the SVG, but these can be obtained from the `window` attribute (cell 7). When adding two graphs, nodes are merged if they have the same alias, the node IDs do not influence the result. The graphs generated by `node` and `star` have aliases equal to node IDs (cell 8), which is why in the previous examples it seems that nodes were merged by ID.

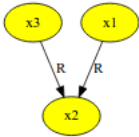
3 Order Properties

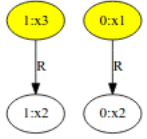
Given intension graphs G and H , we consider G to be entailed by H if we can map G to H (preserving all edges, labels and aliases). Entailment gives rise to a preorder on intension graphs (a preorder is like an order but allows elements to be

```

In [1]: from cgnav import *
In [2]: node(["m"], "x")
Out[2]: m


In [3]: s = star(["R"], "x1", "x2"); s
Out[3]:


In [4]: t = star(["R"], "x3", "x2"); s+t
Out[4]:


In [5]: d = s.windowed(x="x1") + t.windowed(y="x3"); d
Out[5]:


In [6]: d.window
Out[6]: {'x': 0:x1, 'y': 1:x3}

In [7]: (s+t).window
Out[7]: {'x1': x1, 'x2': x2, 'x3': x3}

```

Fig. 2. Console session (part 1)

equivalent). To start with, let us consider the entailment preorder for graphs with empty window (i.e. logical sentences). The `IntensionGraph` class provides a binary product operation which realizes an infimum in the entailment preorder (cf. [1, pg. 208], also [4, Cor.1]). For an example, consider the graph G from cell 8. The product pairs up nodes in all combinations (one from each operand) and describes their commonalities in a single (not necessarily connected) graph. It can be verified that the graphs G and $G * G$ (cell 9) entail each other, which is expected because the product realizes an infimum. The `(components)` method (cf. cell 10) decomposes a graph into a list of its connected components, and the component $c2$, which happens to be the middle component in the SVG of $G * G$, can be mapped to G in two possible ways. The `(morphisms)` method computes all morphisms (i.e. all preserving maps, see above) from one graph to another, and collects these in a `Table` object, which renders as HTML (via a `_repr_html_` method, see cell 10). For a fixed system of labels (which corresponds to a logical signature), the `terminal` method returns a maximal intension graph (cell 11).

Let us also consider intension graphs with a single designated node (designated by the alias "0"). The designated node in the product is obtained by pairing up nodes with the same alias. In cell 12, we have multiplied two copies of G with aliases set to different nodes (the call `windowed(nid)` is a shorthand for `windowed("0"=nid)`), and in this setting we identify the product with the single component which holds the designated node. So the product of connected graphs is connected in this case. The table of morphisms (cell 13) contains only one morphism on the first factor (unlike in cell 10) because morphisms must preserve aliases. The maximal graph with a single designated node is shown in cell 14.

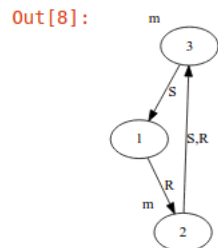
4 Power Context Families

The library can read in power context families in a custom file format which is derived from Burmeister's *cxt* representation format for formal contexts. The *pcf* format differs in that the file must start with a line "B-PCF" instead of "B", and may contain additional contexts after the first one. The contexts are stated in the order of the power context family (up to a largest index), and empty contexts must be stated by two "0"-lines. The commands in cell 15 read in a power context family, which displays as HTML. Power context families can be represented as intension graphs, using the `ig` function (cell 16).

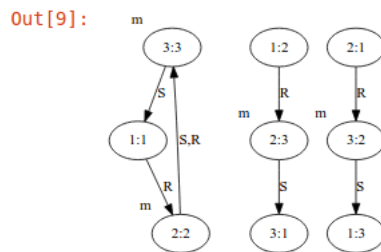
5 Concept Lattices

Concept lattices of intension graphs can be described by pattern structures[2]. The constructor of the `PatternStructure` class takes as parameters the pattern class and a power context family (cell 19). Lattice building is currently only supported for intension graphs with a single designated node, as described in Sect. 3. A simple `build` command builds the concept lattice (cell 19), but the library does not deal with inherent complexity problems (morphism checks), so

```
In [8]: G = (star(["R"], "1", "2") + star(["R", "S"], "2", "3") \
+ star(["S"], "3", "1") + node(["m"], "2") \
+ node(["m"], "3")).windowed(); G
```



```
In [9]: prod = G * G; prod
```



```
In [10]: c1,c2,c3 = prod.components()
c2.morphisms(G)
```

Out[10]:

3:1	2:3	1:2
3	2	1
1	3	2

```
In [11]: M = AttributeSystem([["m"],[],["R","S"]])
IntensionGraph.terminal(M)
```

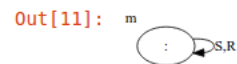


Fig. 3. Console session (part 2)

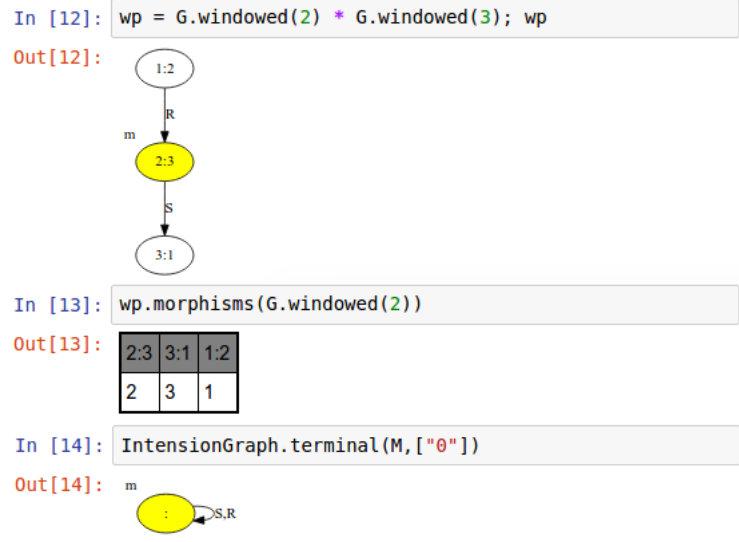


Fig. 4. Console session (part 3)

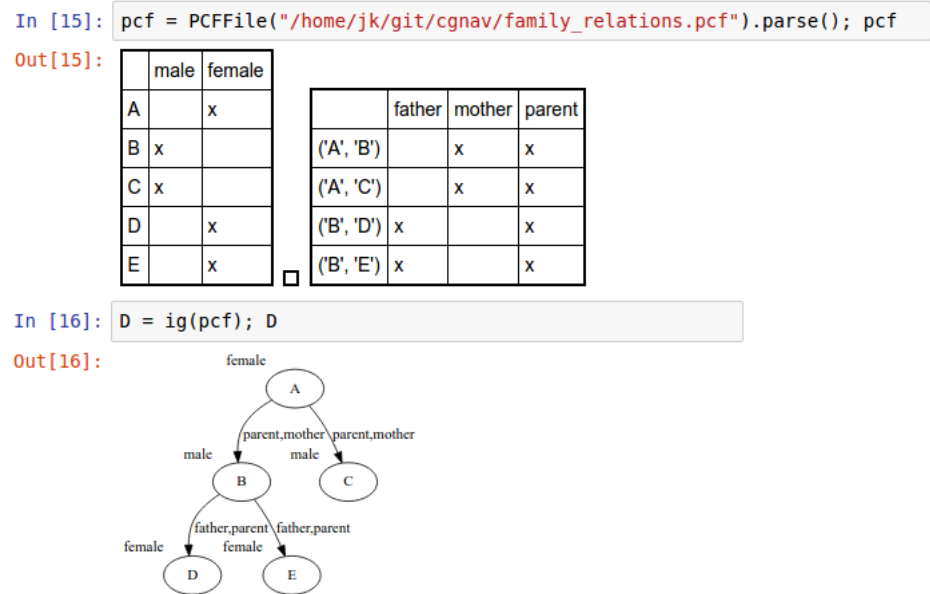


Fig. 5. Console session (part 4)

the build algorithm will likely fail for anything but small examples. The pattern class must expose a certain interface required by the build algorithm, and IntensionGraph is currently the only implementation of the interface. The

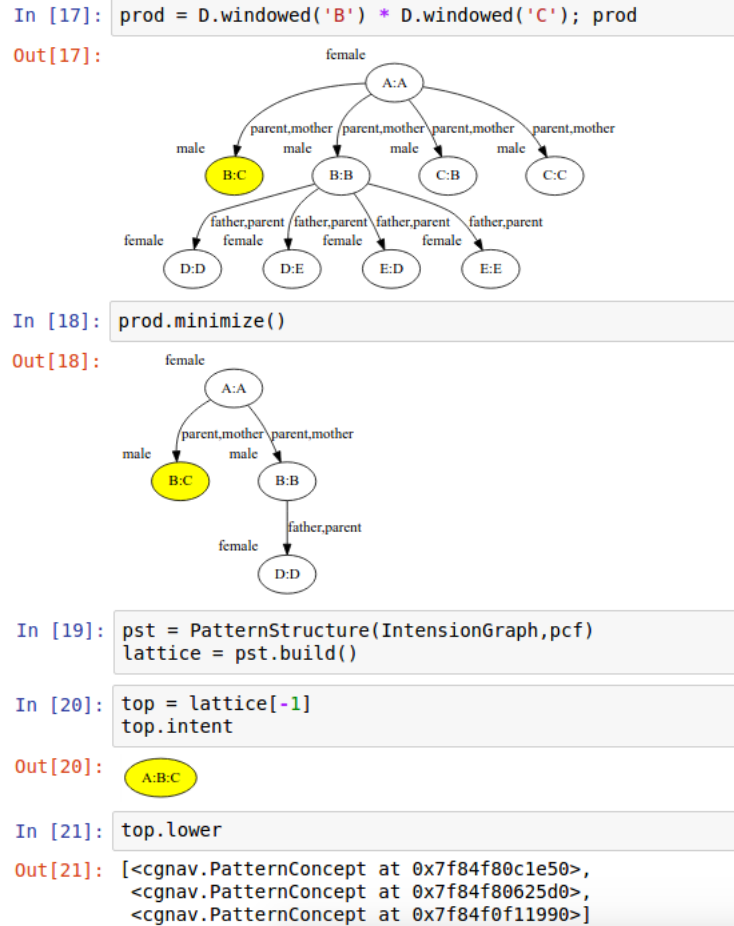


Fig. 6. Console session (part 5)

build algorithm is a variant of Ganter's NextConcept algorithm. The algorithm multiplies two patterns in each construction step, starting from the object intents. The object intents are obtained from D (see cell 16) by choosing the respective object as a designated element. Cell 17 computes the intent for the concept generated by objects B and C . The minimize function maps each intension graph to another, node-minimal intension graph (cell 18), which can be used for display and further computations. The concept lattice is simply represented as a concept list, where list order respects concept order. In particular, the

top concept is the last element in the list (cell 20). Each concepts has attributes `upper` and `lower`, which hold lists containing the upper and lower neighbors, respectively.

6 Conclusion

The core API functions of a library for intension graphs and concept lattices thereof have been presented. Intension graphs formalize conjunctive queries and have solutions w.r.t. a given power context family. The library and two sample pcf files are available at <https://github.com/koetters/cgnav>. Although the library is independent of the Jupyter environment, the visualizations provided by the environment may be instructive and facilitate further development. Efficiency of computations must be improved to allow for somewhat larger power context families. Transformation of data from other sources (like RDF or relational databases) into power context families could be a useful feature and may involve conceptual scaling (e.g. for numeric values).

References

1. Chein, M., Mugnier, M.L.: Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs. Advanced Information and Knowledge Processing, Springer, London (2009)
2. Ganter, B., Kuznetsov, S.O.: Pattern structures and their projections. In: Delugach, H.S., Stumme, G. (eds.) Proceedings of ICCS 2001. LNCS, vol. 2120, pp. 129–142. Springer (2001)
3. Ganter, B., Wille, R.: Formal concept analysis: mathematical foundations. Springer, Berlin (1999)
4. Köttters, J.: Concept lattices of a relational structure. In: Pfeiffer, H.D., Ignatov, D.I., Poelmans, J., Gadiraju, N. (eds.) Proceedings of ICCS 2013. LNCS, vol. 7735, pp. 301–310. Springer (2013)
5. Köttters, J.: Intension graphs as patterns over power context families. In: Proceedings of CLA 2016 (2016), to appear
6. Wille, R.: Conceptual graphs and formal concept analysis. In: Lukose, D., Delugach, H.S., Keeler, M., Searle, L., Sowa, J.F. (eds.) Proceedings of ICCS 1997, 5th International Conference on Conceptual Structures. LNCS, vol. 1257, pp. 290–303. Springer, Heidelberg (1997)