A Python Library for FCA with Conjunctive Queries

Jens Kötters

Abstract. The paper presents a Python library for building concept lattices over power context families, using intension graphs (which formalize conjunctive queries) as concept intents. The `IntensionGraph` class implements intension graphs and algebraic operations on them. An interactive Jupyter notebook session is illustrated and used to present 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

A Python library is presented which uses conjunctive queries as intents to build concept lattices over relational data. The data is assumed to be representable by a power context family [5]. Conjunctive queries are formalized by intension graphs, which are introduced in [4] as attribute-labeled graphs with an optional *window* (which designates distinguished nodes). Concept lattices are obtained as described in [3], where conjunctive queries have been formalized using relational structures. In fact, the family tree example of [3] (Figs. 1 and 2) will be reused as an example in Sects. 4 and 5. Although `IntensionGraph` is the main class of the presented library, the particular formalization of conjunctive queries is not central to this paper as it focuses more on the visual representation, which is independent of the underlying formalism.

Conjunctive queries correspond to *primitive positive (pp)* formulas. These formulas are built from atoms using only conjunction ($\wedge$) and existence quantification ($\exists$). Section 2 documents this correspondence. Section 3 focuses on the morphism (pre-)order on intension graphs. Algebraic operations are presented in these sections. Section 4 covers power context families, and Sect. 5 focuses on the concept lattice.

2 Primitive Positive Formulas

In what follows, we will present the API functions in the context of an interactive console session (Figs. 1, 2, 3, 4 and5) 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 indeces (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.

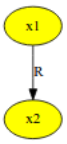
3 Order Properties

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 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 [3, 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).

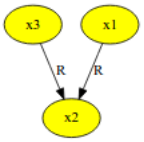
```
In [1]: from cgnav import *

In [2]: node(["m"],"x")
Out[2]:
```

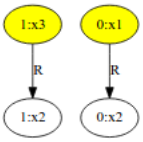

```
In [3]: s = star(["R"],"x1","x2"); s
Out[3]:
```




```
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]:
```



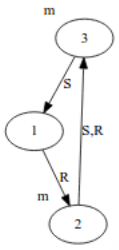

```
In [6]: d.window
Out[6]: {'x': 0:x1, 'y': 1:x3}
In [7]: (s+t).window
Out[7]: {'x1': x1, 'x2': x2, 'x3': x3}
```

Fig. 1. Console session (part 1)

Let us also consider intension graphs with a single designated node (designated

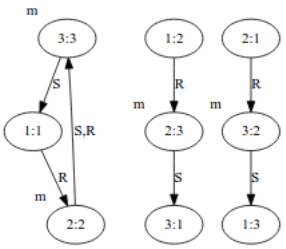
```
In [8]: G = (star(["R"],"1","2") + star(["R","S"],"2","3")\
            + star(["S"],"3","1") + node(["m"],"2")\
             + node(["m"],"3")).windowed(); G
```

Out[8]:




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

Out[9]:




```
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)
```

Out[11]:

Fig. 2. Console session (part 2)

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

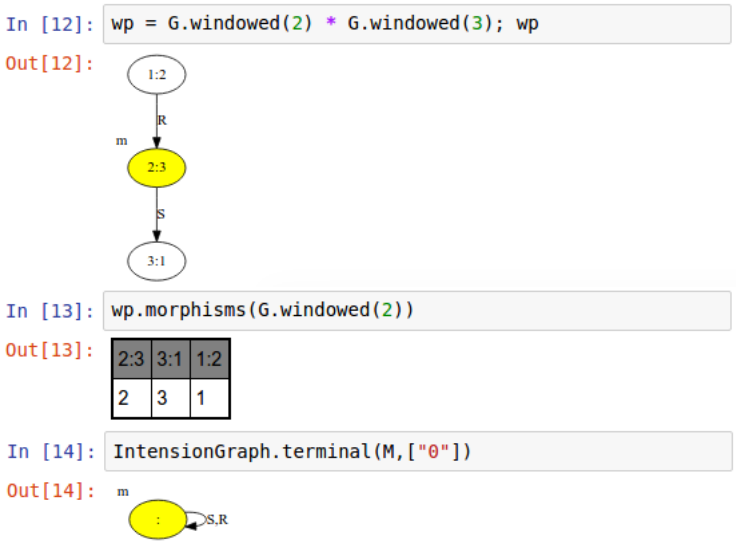


Fig. 3. Console session (part 3)

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 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

In [15]:
```
pcf = PCFFile("/home/jk/git/cgnav/family_relations.pcf").parse(); pcf
```

Out[15]:

	male	female
A		x
B	x	
C	x	
D		x
E		x

	father	mother	parent
('A', 'B')		x	x
('A', 'C')		x	x
('B', 'D')	x		x
('B', 'E')	x		x

In [16]:
```
D = ig(pcf); D
```

Out[16]:

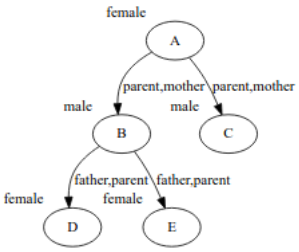


Fig. 4. Console session (part 4)

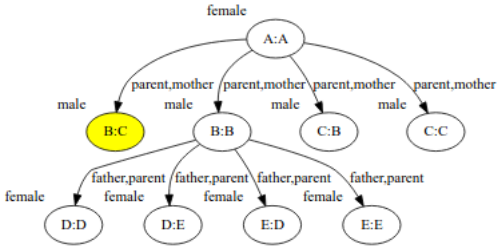
`IntensionGraph` is currently the only implementation of the interface. The build algorithm is a variant of Ganter's NextConcept algorithm. The algorithm multiplies two patterns in each construction step, starting from the object intents. The object intents are obtained from D (see cell 16) by choosing 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

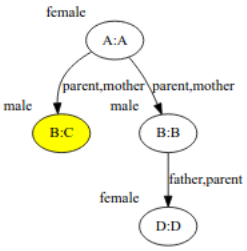
```
In [17]: prod = D.windowed('B') * D.windowed('C'); prod
```

Out[17]:




```
In [18]: prod.minimize()
```

Out[18]:




```
In [19]: pst = PatternStructure(IntensionGraph,pcf)
         lattice = pst.build()

In [20]: top = lattice[-1]
         top.intent
```

Out[20]:


```
In [21]: top.lower
```

```
Out[21]: [<cgnav.PatternConcept at 0x7f84f80c1e50>,
          <cgnav.PatternConcept at 0x7f84f80625d0>,
          <cgnav.PatternConcept at 0x7f84f0f11990>]
```

Fig. 5. Console session (part 5)

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. Kötters, 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)
4. Kötters, J.: Intension graphs as patterns over power context families. In: Proceedings of CLA 2016 (2016), to appear
5. 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)