Combining FCA Software and Sage

Uta Priss

Edinburgh Napier University, School of Computing,
www.upriss.org.uk

Abstract. This paper discusses in how far FCA software can be combined with the computer algebra system Sage. The motivation for this paper is teaching mathematics to software engineering students using Sage and FCA which highlights differences and connections between mathematical and computational structures. Furthermore, this paper provides implementation details on how Sage's functions for matrices, graphs, and poset/lattice theory can be applied to FCA data and concludes with hints for how to build an FCA/Sage tool.

1 Introduction

This paper discusses the use of Formal Concept Analysis (FCA) in combination with the open source computer algebra system (CAS) Sage[1]. The paper is motivated by the author's experiences with using FCA and Sage in an introductory mathematics class for software engineering students. Although using CAS software for teaching maths is not new, we believe that the use of CAS software in a software engineering context and in combination with FCA is new. It was an aim of the class to highlight connections and differences between mathematical and computational structures.

Mathematical models and computational implementations use different data structures, types of variables and general strategies. This becomes very apparent when one combines functions from a CAS tool (such as Sage) with programming constructs from an (object-oriented) procedural language (such as Python). Functional and logic programming languages (such as ML, Haskell, Prolog and Scheme) tend to be more suitable for representing mathematical structures than procedural languages but most real-world application software is written in languages, such as Java, C-derivates or scripting languages. Therefore, it seems preferable to teach software engineering students how to use mathematics from a computational setting they are familiar with (using an object-oriented procedural language) instead of teaching them to use a maths-specific language. Since Sage, the most advanced, most complete open source CAS that is currently available, is written in Python, the questions arise as to how to express mathematical structures in Python and, in particular, how to use FCA software with Python and Sage.

Rudolf Wille the founder of FCA also pioneered the use of FCA for teaching mathematics to non-mathematics students (Wille, 1995). This encouraged us to use data analysis and FCA as a guiding thread throughout the mathematics class. Because FCA data is represented as formal contexts which are binary relations or binary matrices,

[1] http://www.sagemath.org/

connections can be established with relations, functions (as binary relations), matrix operations and graph theory as a way of graphically representing binary relations. Lattices and trees provide a means for representing hierarchies (for example the directory structure in a computer), search structures and orderings. Descriptive statistics and FCA can be introduced as two different, complementing means for data analysis. Last but not least, attribute implications in FCA lattices provide a connection to formal logic.

From a practical viewpoint it would be desirable if Sage contained an FCA package, but that is not currently the case. Therefore, the main part of this paper contains a detailed investigation with respect to how FCA data can be imported into and explored with Sage, which FCA features are already available and which still need to be implemented by the FCA community. It is hoped that this paper will stimulate the development of an FCA package for Sage by the community.

This paper does not provide an introduction to FCA. Pointers to introductory FCA materials and software are available on-line[2] and in the main FCA textbook by Ganter & Wille (1999). It is beyond the scope of this paper to introduce the Sage software but it should be mentioned that Sage can be used in a variety of manners: via an on-line interface, via a command-line (REPL, Read-eval-print loop) interpreter or as libraries imported into stand-alone Python scripts. The on-line and command-line interfaces are similar to a calculator. This means that the students type simple statements (for example a mathematical equation) and see the result. Or they can build larger scripts, which are checked and executed by the interpreter after each line. The on-line interface allows to store the calculations and to publish them on the web. It is also possible to retrieve graphical output in form of plots and graphs. Sage is an umbrella tool for other open source tools from many areas of mathematics (algebra, number theory, logic, calculus, statistics, numerical mathematics, graph theory and others). The developers of Sage avoid to "reinvent the wheel" and, instead, incorporate tools that already exist by writing interfaces and wrappers. Sage can also serve as an interface to some commercial CAS tools.

The rest of this paper is organised as follows: section 2 elaborates on the motivation for this paper – the challenges of teaching mathematics with Sage and FCA. Section 3 discusses different aspects of combining Sage and FCA software, in particular with respect to matrices, graphs and posets/lattices. Section 4 provides a concluding elaboration on some of the questions that need to be answered in order to build a FCA/Sage tool.

2 Motivation: teaching mathematics with Sage and FCA

This section discusses the challenges involved in teaching the use of mathematical software to software engineering students. Ideally, the students should see mathematical structures as something that extends and integrates with their existing programming knowledge, instead of seeing maths as something that is disconnected and unrelated. Therefore it is important to highlight differences and connections between mathematical and computational structures.

[2] http://www.fcahome.org.uk

The differences between mathematical and computational structures can be described from a semiotic perspective: Priss (2004) argues that variables in mathematics and computation are different. Variables in programming languages are triadic signs which have a name, a type/value pair and a context (or state). Variables in mathematics, however, are binary signs, where the distinction between name and value is irrelevant and the context is more abstract and independent of a temporal existence. Although humans need time to read a mathematical proof and need to understand the sequence of arguments in that proof, from a logical viewpoint, mathematical proofs do not have a temporal sequence and, instead, consist of equivalent transformations. For these reasons, $n = n+1$ is a completely normal programming language statement indicating the temporal change of the value of a variable, whereas in mathematics the same statement is a contradiction because mathematical variables do not have state-dependent values. Because mathematics operates in abstract, hypothetical contexts, constructs such as infinity are acceptable whereas computer programs, which always exist in a specific computer at a specific point in time and place (in memory) using a specific language, can never accurately represent such abstract constructs.

Another difference is presented by the basic data structures of mathematics and programming. Mathematics is grounded in set theory whereas the basic data structures of procedural programming languages are lists or arrays. Python by itself (without Sage) already has some interesting features for representing mathematical constructs such as sets and recursive functions. From a computational viewpoint, sets are an advanced data structure which is complicated to implement and has higher demands on resources than arrays. Arrays have an implicit ordering which tells the computer how to process the elements. When software engineering students first encounter a Set type in a programming language they tend to be surprised by the fact that the program will not necessarily print the elements of the set in the same sequence in which they were entered and that duplicate elements disappear or produce an error. Farahani (2009) argues that Python is particularly good at representing mathematical structures such as sets. Extensional definitions (`Set([1,3,5])`) in Python are obviously similar to mathematical notations; but Python also supports intensional definitions of sets. For example, the mathematical definition of a set $S := \{x \mid x = 2n; 0 \leq n \leq 19\}$ corresponds to `S = Set([2*x for x in range(1,19)])` and $A = \{x \mid x^2 + x - 6 = 0\}$ corresponds to `A=Set([x for x in range (-50,50) if x**2+x-6==0])`. These examples show that the representations of sets in Python can be quite close to mathematical notations, with some obvious differences, such as the value of x in A requires a fixed range in Python whereas it ranges across infinity in the mathematical representation. The use of Python for implementing mathematical expressions is also recommended by Fangohr (2004) who compares using C, MATLAB and Python for teaching mathematics to engineering students and concludes that Python is the most intuitive out of the three.

Another example for demonstrating similarities between mathematical and computational structures are recursive definitions of functions. This is not Python-specific, but in fact possible with most, if not all, modern programming languages. For example, the factorial function $n! = 1$ for $n = 0$ and $n! = n(n-1)!$ for $n > 0$ can be defined

as a recursive function `factorial` which returns 1 in the first case and returns `n * factorial(n-1)` in the second case.

After exploring Python's own maths-like expressions, it is of interest to see what Sage has to offer to software engineering students. Using Sage, Boolean logic can be explicitly evaluated. This is interesting because all programming languages use Boolean logic implicitly (because they are ultimately implemented with 0s and 1s), but explicit use of Boolean logic is not always easy. In Sage, a simple logic puzzle such as

> If you invite Susan, you must invite John.
> You must invite either John or Mary, but not both.
> You must invite either Susan or Mary or both.
> If you invite Mary, you must also invite Susan.

can be evaluated with `propcalc.formula("(s->j)&(j^m)&(s|m)&(m->s)")`. Because the size of the underlying truth table grows exponentially with the number of variables in the formula, an extensional (truth-table based) evaluation of more complicated formulas is not feasible. This demonstrates the need for intensional (Boolean algebraic) solutions and the use of more expressive forms of logic.

The notions of "extension" and "intension" with respect to the evaluation of logical formulas and with respect to extensional and intensional definitions of sets provide a connection with FCA. From an FCA viewpoint, the most interesting packages in Sage are those that implement logic, matrices, graphs, lattice theory and statistics. In an ideal situation, one should be able to enter a formal context and then analyse the data from any of those areas. The next section discusses in how far that is already possible and what is still missing. In a mathematics class this can be achieved by presenting the students with a coursework exercise consisting of data sets that need to be analysed from different aspects. For example, values of central tendency can be analysed using graph-theory (looking at nodes with a high Google-style PageRank), descriptive statistics or exploring FCA lattices. Other useful software for such a class is a stand-alone graphical FCA tool and a spreadsheet tool for basic descriptive statistics because the language R which is integrated in Sage is probably a semester-long topic in itself. A simpler statistics tool for basic data analysis appears to be still missing from Sage.

3 Combining FCA and Sage

This section discusses in how far existing Sage modules are relevant for FCA purposes and how FCA software can interact with them. First, it needs to be considered how Sage is normally used and how FCA data should be prepared so that it can be imported into Sage. The main use of Sage is in an interactive mode which is well-suited for a learning environment where students enter expressions one line at a time and see instant results (or error messages). It is possible to import Sage modules into stand-alone Python scripts, but because it takes a significant amount of time and memory resources to load Sage, such scripts are slow. Therefore, for stand-alone applications that need only some mathematical functions it is more efficient to load individual packages (without using Sage's wrappers for these tools) and not to use the Sage core modules. Unfortunately,

this means that slight modifications of the code are required depending on whether functions are accessed via Sage or directly via the packages. Furthermore, there is some overlap of functionality between different packages. For example, matrices are defined in Numpy, Scipy, Sympy and Sage – each with different data structures and functions. This means that there are often many different possibilities for implementing certain mathematical structures. Any future FCA/Sage software developer needs to analyse this situation carefully in order to decide on the best way to integrate an FCA tool with the existing tools.

In order to use FCA with Sage, it is necessary to import/export data. There are as many different FCA file formats as there are FCA tools, but conversion between different FCA file formats can be achieved, for example, by using FcaStone[3]. Therefore it should be sufficient to write functions that export/import data using a small subset of the existing formats. As long as there is no dedicated FCA/Sage tool, any existing command-line FCA tool (such as FcaStone) which computes concept lattices from formal contexts can already be used with Sage via a "system call". Even though system calls may not be particularly efficient, it probably does not matter compared to the high resource demands of Sage itself. As discussed above, if efficiency is required (because of large contexts, high volume of operations) then single purpose, stand-alone tools are a better solution than Sage. But if Sage is used interactively (which implies smallish contexts), a system call to any existing command-line FCA tool will not cause any noticeable delays. Sage functions need to be written that import data in this manner but that is not difficult and some examples are given below.

Currently Sage has functions for matrices, graphs, and poset/lattice theory which are all relevant to FCA. All three of these are discussed in the following subsections. A crucial difference between abstract maths and FCA is that FCA is more applied and data-oriented. Formal contexts are not just matrices, but, instead they are matrices where each row/column corresponds to an object/attribute. Concept lattices are not just abstract graphs, but, instead, the concepts are labelled by objects and attributes. The final subsection of this section discusses the particular needs of FCA with respect to managing the names and labels of objects and attributes which is a challenge for Sage because Sage is more aimed at abstract mathematical structures.

3.1 Matrix operations

Matrix operations are of relevance to FCA because they can be used to implement context operations. Unfortunately, as discussed below, the matrix operations in Sage are not exactly the ones needed for FCA. Formal contexts can be imported as matrices into Sage with just a few lines of code (see Appendix). A future FCA/Sage tool would need to provide a data type (or class) for formal contexts which keeps track of which row/column of a matrix corresponds to which object/attribute. The matrix operations that are currently available in Sage do not consider objects and attributes. Thus it is up to a user to employ these operations sensibly.

Priss (2009) argues that relation algebra is a natural generalisation of the context operations that are described in the standard FCA textbook (Ganter & Wille, 1999).

[3] http://fcastone.sourceforge.net/

Relation algebra is an algebra of Boolean matrices based on the operations union, intersection and relational composition. It has nothing to do and should not be confused with Codd's relational algebra which provides the foundation of relational databases. The details of why relation algebra is relevant for FCA are discussed elsewhere (Priss, 2009) and are beyond the scope of this paper. As far as we know there are currently only two implementations of relation algebra: an optimised high performance version called RelView[4] and a simple script-based version called FcaFlint[5] neither of which is written in Python.

Sage supports many different types of matrices but not relation algebra/Boolean matrices. Currently matrix elements in Sage must form a ring[6]. Thus, currently binary matrices in Sage can only be defined over the Galois field 2, GF(2), or, in other words, as modulo-2 arithmetic. The multiplication tables in GF(2) and relation algebra are the same but the addition differs in that $1+1$ results in 0 in GF(2) and results in 1 in relation algebra. Addition corresponds to inclusive-or in relation algebra and to exclusive-or in GF(2). As a consequence of the differences in addition, matrix multiplication is different as well. Table 1 summarises the operations that are available in Sage and correspond to context operations in FCA. Most operations should be self-explanatory. "Test for containment" is an order relation between matrices which results in "true" if the matrix on the right has a 1 at least in each position in which there is a 1 in the matrix on the left. Density is defined in Sage as "the ratio of the number of non-zero positions and the total number of positions" (i.e. number of rows multiplied by number of columns).

Table 1. List of FCA context operations that correspond to Sage's matrix operations

context operations	matrices over GF(2) in Sage
dual matrix (mirrored along diagonal)	`transpose(m)`
apposition of n and m	`n.augment(m)`
subposition of n and m	`n.stack(m)`
null matrix of dimension i	`matrix(GF(2),`i`,`i`,0)`
diagonal matrix of dimension i	`matrix(GF(2),`i`,`i`,1)`
test for equality	`==`
test for containment	$<=$
switching rows i and j	`m.swap_rows(`i`,`j`)`
switching columns i and j	`m.swap_columns(`i`,`j`)`
forming a submatrix	`m.submatrix(`i`,`j`,`k`,`l`)`
calculating density	`m.density()`

[4] http://www.informatik.uni-kiel.de/∼progsys/relview.shtml

[5] http://fcastone.sourceforge.net/fcaFlint.html

[6] There was a discussion on the Sage mailing list where a user asked for a Boolean matrix algebra and was told by the Sage lead developer, Stein, that matrices must be constructed over rings. Another Sage developer (C. Witty) then argued that Sage should also allow matrices over lattices. But it seems that so far this has not yet been implemented. See http://www.mail-archive.com/sage-support@googlegroups.com/msg04348.html

The core operations of addition, subtraction and all forms of matrix multiplication are different between relation algebra and GF(2). While union, intersection and composition from relation algebra are useful for FCA applications (Priss, 2009), it is not clear whether the GF(2) operations have interesting FCA applications. Calculating inverse matrices is different as well. In GF(2), the multiplication of a matrix and its inverse (if it exists) results in the diagonal matrix. In FCA and relation algebra, forming an inverse or complement means to turn 0s into 1s and 1s into 0s. It is not difficult to write functions for the missing relation algebra operations (an example for "union" is given in the Appendix). Only union, composition and calculation of inverse matrices are needed because the other ones can be derived. Implementing such operations in an efficient manner that is fully compatible with Sage's matrices, however, is slightly more difficult because Sage's matrix operations are implemented in Cython. Instead of adding to the Sage core, another option would be to add relation algebra operations to Numpy's or Sympy's matrix operations.

3.2 Graphs

The NetworkX package which is part of Sage provides a wide range of graph operations, including standard graph theoretical operations (cliques, components, cycles and so on) and application-oriented operations (calculating the Google PageRank of the nodes or determining whether a graph is a small world network). With respect to FCA, both formal contexts and lattices can be represented as graphs. In this case a formal context would be a bipartite graph. NetworkX then provides, for example, graph-theoretical measures of centrality and clustering which could be compared to lattice-theoretical measures.

Using the read_edgelist() function a graph can be imported from a comma-separated-value (csv) file of a formal context or concept lattice. For formal contexts, it is possible to create csv files with FcaStone or to export them from databases or spreadsheets. For concept lattices, dot files generated by FcaStone can be converted into csv files with a few lines of code (see Appendix).

The graph of a concept lattice should be imported as a DiGraph (directed graph). A picture of a graph in Sage can be produced either through the show() function or via NetworkX and the Graphviz software (if this is installed). Although it is possible to label nodes, the kind of double labelling of concepts in a concept lattice with objects and attributes does not seem to be supported. If the concept lattices are imported via their dot files as generated by FcaStone, then the concepts are only labelled by numbers. This is probably not a significant problem because presumably one would use the NetworkX functions only in addition to standard FCA software, not instead of.

It should be mentioned that one of the graph operations is close to one of the missing relation algebra-related operations (as discussed in the previous section). The graph theoretical function

```
G.transitive_closure().adjacency_matrix()
```

provides a form of transitive relational composition. Unfortunately, because Sage requires adjacency matrices of graphs to be symmetric, non-reflexive and square this

composition operation is not applicable to all Boolean matrices. In our opinion, this connection shows that relation algebra provides a link between matrix and graph theory that is missing in Sage.

3.3 Lattice operations

Once a concept lattice has been imported as a graph as described in the previous section, it can be converted into Sage's Poset or LatticePoset types (see Appendix). As long as the lattice graph `G` has been imported as a DiGraph and is indeed a poset (or lattice), the commands `P = Poset(G)` or `L = LatticePoset(G)` create a poset `P` or lattice `L` which has access to any of Sage's functions that are available to these types. For example, one can issue the commands

```
P.top()
P.is_meet_semilattice()
L.is_distributive()
```

and so on (more examples are in the Appendix). While these functions may not be very useful for real world applications, it is helpful to have such functions available in a teaching environment.

Currently, there does not seem to be an easy way to reverse this process and convert a lattice that is created in Sage back into a formal context. For example, `P = Posets.BooleanLattice(4)` creates a Boolean lattice with 4 atoms. While it is possible to compare this lattice to a lattice that was derived from a formal context, neither can be converted back into a formal context. In the previous section it was mentioned that it is difficult to label NetworkX graphs in the usual FCA manner. The same problem also applies to lattices: it is currently not possible to use "labelled lattices"[7] in Sage.

3.4 Names and labels

The discussion in the previous three sections shows that a crucial topic for FCA software is the management of the names of objects and attributes. Formal contexts are matrices in which the names of rows and columns are significant. A data type for formal contexts needs to be created which supports matrix operations but keeps track of objects and attributes at the same time. As discussed by Priss (2009), implementing context operations in a manner that is meaningful for objects and attributes is a challenge. For example, for relational composition the set of attributes of the left context should be the same and in the same order as the set of objects of the right context. If they are not in the correct order, they first need to be reordered. Furthermore the operations should be generalised to non-square matrices, which is a further challenge and creates a structure which is no longer a relation algebra (Priss, 2009).

[7] A discussion on the Sage mailing list reveals that it is possible to use labels in lattices for presentation purposes but not for label-preserving isomorphisms. See http://www.mail-archive.com/sage-devel@googlegroups.com/msg35734.html

Concept lattices require a "double labelling" of the nodes with objects and attributes. While there is support in NetworkX for labelling nodes and edges in a graph and limited support in Sage for labelling nodes and edges in a lattice for presentation purposes, there appears to be no support for the kind of double labelling required by FCA. Again this is a challenge for FCA software.

4 Conclusion

The analysis conducted in this paper demonstrates that it is already possible to use Sage functions with formal contexts and lattices if they are imported as comma-separated-value files. Some of the operations that are useful for FCA (in particular with respect to context operations and labelling) are still missing from Sage. Furthermore, it is not yet easy to export data from Sage into FCA formats.

Before anybody sets out to write an FCA package for Sage, it would be useful, however, if there was some discussion within the FCA community with respect to what functionality would actually be required from such a tool. Furthermore, it would be useful to conduct some user testing with a simulation of a REPL environment. It seems that an ideal application for a FCA/Sage tool is interactive exploration such as attribute exploration. On the other hand, most likely users would not want to enter a formal context one element at a time or to manipulate the graph of a concept lattice in a command-line tool. Existing graphical (GUI) FCA tools provide this functionality and should not be replicated by an FCA/Sage tool. It may be possible to make use of existing implementations of FCA algorithms, in particular code from one of the existing Python FCA tools might be useful.

The following list summarises some of the issues, challenges and opportunities relating to a FCA/Sage tool:

- Should one develop a stand-alone Python FCA tool that can be packaged with Sage or directly write Sage modules or modules for one of Sage's components (Numpy or Sympy)?
- One would need a class (data type) for formal contexts and a class for concept lattices. A particular challenge for these classes is to integrate with existing Sage structures (matrices, graphs, lattices) in a manner that respects objects and attributes.
- User testing should be conducted before a class for formal contexts and lattices is finalised because it can potentially be difficult to interact with such a class in a command-line environment. In particular, it needs to be evaluated which FCA algorithms or applications are particularly relevant for a command-line environment and which are better for GUI tools.
- An implementation of relation algebra is needed. This is both relevant for matrices and graphs (for example with respect to the transitive closure of graphs). Sage would need to allow matrices to be defined over lattices as well as rings.
- There should be classes for "labelled posets" and "labelled lattices" in Sage which would provide support for labelling of nodes and edges, but also for the kind of "double labelling" with objects and attributes as used in FCA.

– From a research perspective, an integration of FCA and Sage would make it easy to conduct experiments that compare matrix and graph structures to lattice properties. This might lead to new theoretical insights.

It is hoped that this paper might stimulate discussions in the FCA community about these topics. To conclude this paper, it should be mentioned that the feedback from the students in the class "Mathematics for Software Engineering" which was a motivation for this paper was overwhelmingly positive. Students commented, for example, that the class "helped to lose fear of maths". This seems to be some evidence that an FCA- and Sage-based approach to teaching mathematics to non-mathematicians is promising.

References

1. Fangohr, H. (2004). *A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering.* In: Computational Science, Workshop on Computing in Science and Engineering Academic Programs, Springer, LNCS 3039, p. 1210-1217.
2. Farahani, Ali; Uhlig, Ronald P. (2009). *Use of Python in Teaching Discrete Mathematics.* 2009 Annual Conference of. American Society for Engineering Education (ASEE).
3. Ganter, Bernhard, & Wille, Rudolf (1999). *Formal Concept Analysis. Mathematical Foundations.* Berlin-Heidelberg-New York: Springer.
4. Priss, Uta (2004). *Signs and Formal Concepts.* In: Eklund (ed.), Concept Lattices: Second International Conference on Formal Concept Analysis, Springer Verlag, LNCS 2961, p. 28-38.
5. Priss, Uta (2009). *Relation Algebra Operations on Formal Contexts.* In: Rudolph; Dau; Kuznetsov (eds.), Proceedings of the 17th International Conference on Conceptual Structures, ICCS'09, Springer Verlag, LNCS 5662, p. 257-269.
6. Wille, Rudolf (1995). *"Allgemeine Mathematik" als Bildungskonzept für die Schule.* In: Mathematik allgemeinbildend unterrichten. Biehler et al. (eds). Aulis, p. 41-55.

Appendix

This appendix contains examples of Python/Sage code for using FCA with Sage. More explanation for these code examples and sample data files to be used with this code are provided at http://fcastone.sourceforge.net/fcasage.html

Loading a formal context into a matrix:

```
import numpy
from sage.all import *
M1 = Matrix(numpy.loadtxt("sageExampleMatrix.txt"))
M1a = M1.change_ring(GF(2))
```

A function for calculating the union (relation algebra addition) of matrices:

```
from sage.all import *
m=matrix(GF(2),2,[1,0,0,1])
n=matrix(GF(2),2,[1,1,0,1])
```

```
k =matrix(GF(2),2)

def union(a,b,c):
    for i in range(2):
        for j in range(2):
            res = int(a[i][j]) | int(b[i][j])
            k[i,j] = res

union(n,m,k)
```

Importing a formal context as a bipartite graph:

```
import networkx
from sage.all import *
G = networkx.read_edgelist("sageExample.csv",delimiter=",")
G2 = Graph(G)
```

Converting a dot file (as produced by FcaStone) into a csv file that allows to import a concept lattice as a graph:

```
import re
file = open("sageExample.dot","r")
text = file.readlines()
file.close()
outputfile = open("sageExampleLattice.csv","w")
keyword1 = re.compile(r"\[|\{|\}")
keyword2 = re.compile(r" -> ")
for line in text:
    if not keyword1.search (line):
         outputfile.write(keyword2.sub(",",line))
```

Lattice operations:

```
import networkx
from sage.all import *
G = networkx.read_edgelist("sageExampleLattice.csv",delimiter=",",\
create_using=DiGraph())
P = Poset((G.networkx_graph().nodes(),G.networkx_graph().edges()),\
cover_relations=True)
P.show()
P1 = Poset(G)
P1 == P
P.top()
P.bottom()
P.is_meet_semilattice()
L = LatticePoset(G)
L.is_distributive()
```