Visualization of Formal Concepts

Lisa Jin

27 April 2016

Abstract

Formal concept analysis (FCA) is a method that applies to structuring the philosophical notion of human thought into logical concepts. These formal concepts follow a definite mathematical model that can be expressed as lattices. The visualization tool discussed as follows was built as a basic prototype for representing and performing operations on a formal context.

1 Introduction

Based on lattice theory, formal concept analysis (FCA) is a mathematical method for deriving a concept hierarchy from a dataset. Given a table of object-attribute relations, where each object row consists of a fixed set of binary attribute columns, we can extract concepts as partially ordered sets of objects.

A formal context can be defined as $\mathbb{K} \coloneqq (G, M, I)$ [1], or the sets of objects, attributes, and object-attribute relations ($I \subseteq G \times M$), respectively. For object g and attribute m, $(g, m) \in I$ when m is an attribute of g. Every formal concept (A, B) must further follow the relation,

$$(A_1, B_1) \leq (A_2, B_2) :\Leftrightarrow A_1 \subseteq A_2 (\Leftrightarrow B_1 \supseteq B_2).$$

These concept sets can be visually represented as concept lattices due to the strict ordering of sub- and superconcepts described above. Lattices are displayed in decreasing order by a concept's extent for our purposes; the top node, or universal set, extends to the bottom node, or empty set. At a high level, this software project has four main components:

1. **Concept Extraction**: FCA may consume exponential memory, as trying all possible subsets of the data becomes alarmingly expensive for larger datasets. Several current algorithms exist, but this tool utilized Ganter's NextClosure algorithm [2]. It iteratively reduces the search space by finding intersections of current intents [3].
2. **Edge Drawing**: Determining edges between nodes approaches $O(n!)$ complexity with the most naïve solution. The Floyd-Warshall algorithm, a polynomial-time shortest path method, was useful for identifying only the most critical edges between nodes. This correctly expressed the way subconcepts exist in FCA.
3. **Set Operations**: Five set operations were implemented: closure, boundary, interior, derived set, and isolated point of a set. Users can specify input as a numbered list of objects in some given extent.
4. **Lattice Visualization**: For complex datasets with numerous edges, concept lattices often must be hand-drawn for clarity. The visualization was successful for small datasets using an elementary approach of sorting by a concept's "weight," or number of objects in its extent. The NetworkX Python package displayed a static image with labeled nodes.

2 Choice of Data Structures

Due to the efficiency of the C++ STL, all components besides image rendering were written in C++. Of highest importance was representing concept themselves, which relied on C++'s object-oriented features. Concept nodes must hold several layers of data, including extent and intent at the lowest conceptual level, and coordinates and "weight" at the highest visualization level. We also must consider how each concept is integrated into the overall lattice structure.

Extent and bitset. A formal context begins in the raw form of a cross table that shows, for each row of objects, the attributes that are present from among all those that exist in the table. The focus on ordering by extent makes it sensible to use a Boolean vector or C++ `bitset` for each object. Apart from providing member functions for manipulating bits and performing set intersection, the `bitset` is relatively compact in memory. Converting from `string` to `bool` for input reading or `bool` to `long` for concept comparison are effortless.

Linked list node inheritance. Each concept `struct` held separate vectors of parent and child pointers in accordance with the lattice's partially ordered nature. The subset relation on set P is anti-symmetric ($x \leq y \wedge x \neq y \Rightarrow y \nleq x$ for all $x \in P$) and transitive ($x \leq y \wedge y \leq z \Rightarrow x \leq z$ for all $x, y, z \in P$). The former dictates that no pair of nodes may have both parent and child relations with each other; there can be no horizontal links. The latter states the recursive inheritance of node relations; the subset of a given node includes all derived subsets as well. Thus, a linked list structure that allows for multiple parent and child nodes was used.

3 Concept Extraction

Using the `bitset` for each intent in the Boolean matrix input, we can generate unique closures by following the lectic ordering of attributes [1]. For set A, we denote the ordering from lowest to highest as $a_1 < a_2 < \ldots < a_i < \ldots < a_m$. The $\oplus$ operator is used to generate a new extent from an existing extent and a single attribute in lectic order:

$$A \oplus a_i = (A \cap (a_1, a_2, \ldots, a_{i-1}) \cup a_i)''$$

The GenClosures algorithm generates the base extent, or closure of the empty set, upon which all further concepts are derived. It continues to call NextClosure until the current extent A is full - equivalent to an empty intent set. The $\oplus$ operator generates possible extents to add to the current set of formal concepts, $\mathcal{F}$.

Algorithm 1: GenClosures

Input: $I, \emptyset$: matrix, null object set
Output: $\mathcal{F}$: set of formal concepts

$A_{prev} \leftarrow \emptyset''$;
while $A_{cur} \neq \mathbb{U}$ **do**
  for $n \leftarrow |A| - 1$ **to** 0 **do**
    NextClosure(I, A_{cur}, n);
    if $A \notin \mathcal{F}$ **then**
      $\mathcal{F} \leftarrow \mathcal{F} \cup A$;
      break;
    else
      $A_{cur} = A_{prev}$;
    end
  end
end

Algorithm 2: NextClosure

Input: I, A_{cur}, n: matrix, extent, pos
Output: A'

$A' \leftarrow (A_{cur} \cap (a_1, \ldots, a_{n-1})) \cup a_n$;
$inter = \mathbb{U}$;
foreach $j \in A_{cur}$ **do**
  $inter \cap I[j]$;
end
foreach $k \notin A_{cur}$ **do**
  if $I[k] \subseteq inter$ **then**
    $A' \leftarrow A_{cur} \cup k$;
  end
end
return A';

In NextClosure, the most expensive computation is finding the set closure in lines 3-5. This involves initializing *inter* with the set of all attributes, and intersecting it with the attribute set of every object present in A_{cur}. This may generate a concept that already exists, and must be checked for in line 5 of GenClosures. Lines 6-9 of NextClosure simply add any objects to the existing set which are a superset of the found intersection. If the resulting concept is unique, it is added to $\mathcal{F}$ and all $|A|$ elements are iterated once again. The worst-case complexity is $O(n^2)$ with respect to the total number of objects, which is reasonable for smaller datasets.

4 Edge Drawing

Every pair of formal concepts (A_1, B_1) and (A_2, B_2) may share both a largest subset, *infinitum*, or smallest superset, *supremum*. The infinitum is formally defined as $(A_1 \cap A_2, (B_1 \cup B_2)'')$ and the supremum as $((A_1 \cup A_2)'', B_1 \cap B_2)$. In the lattice visualization, these will be characterized by the closest lower and upper points at which edges join from a concept pair.

Since infinitum and supremum must exist as unique paths within the concept lattice, it is necessary to find the transitive closure of all edges between concepts. This task was completed using the Floyd-Warshall algorithm, which uses dynamic programming to compare all pairs of graph vertices to discover the shortest path. The algorithm was generalized in this case by using Warshall's algorithm, intended for finding transitive closure of an unweighted graph. While the original algorithm made calculations for addition and minimum, these were respectively substituted with logical conjunction (AND) and disjunction (OR):

Algorithm 3: Floyd-Warshall

Input: M: Boolean adjacency matrix of all subset relations
Output: M': reduced matrix of transitive closures

```
1 for k ← 0 to |M| − 1 do
2 |   for i ← 0 to |M| − 1 do
3 |   |   for j ← 0 to |M| − 1 do
4 |   |   |   if M[i][j] and M[i][k] and M[k][j] then
5 |   |   |   |   M[i][j] ← 0;
6 |   |   |   end
7 |   |   end
8 |   end
9 end
```

The $i \times j$ adjacency matrix H is in fact square, since it represents all possible edges between a single set of concepts. We assign $H_{i,j}$ to 1 if an edge exists between i and j, and 0 otherwise. Before Warshall's algorithm is applied, H is initialized with all possible subset relations between concept nodes. For each k^{th} iteration, the algorithm determines whether a "shorter" path from i through k to j exists. The vertices i through k simply serve as intermediate points in finding transitive closure. The first condition in line 4 tests whether an edge between i and j already exists. The two conditions that follow check for transitivity, and if so, the redundant path from i to j is removed in line 5.

Dynamic programming is used in the sense that adjacency matrices are sequentially constructed over every k^{th} iteration. It continues to modify the matrix until the optimal path between all pairs of vertices is found. Complexity of this algorithm is $\Theta(n^3)$ from the three nested `for` loops, which is quite efficient considering that a graph may initially have up to $|M|^2$ edges. In conjunction with the NextClosure algorithm, the totality of these computations remain within polynomial complexity and are considerably better than brute-force approaches.

5 Set Operations

The five set operations that were implemented include closure, boundary, interior, derived set, and isolated point. Closure of set A is the smallest closed set containing A, which can be found be intersecting all of its subsets. From the closure, the subsequent operations can be derived in the following order:

1. Closure: $\overline{A} = \cap\{B \mid A \subseteq B\}$
2. Boundary: $\partial A = \overline{A} \cap \overline{A^c}$
3. Interior: $A^0 = \overline{A} - \partial A$
4. Derived set: $A^d = \{x \mid x \in \overline{A - \{x\}}\}$
5. Isolated point: $A^i = \{x \in A \mid x \notin \overline{A - \{x\}}\}$

These operations were implemented in C++ after computing all concepts, and required very basic bit manipulations of the extent `bitset`. The user is able to specify the input vector for each desired operation in a text file and view the results upon running the program. Closure and boundary result in concept outputs, while the remaining operations may yield vectors of objects that are not necessarily existing concepts.

6 Lattice Visualization

After computing concepts, edges, and set operations, the concept lattice visualization receives output text files containing results of each step. These are inputs to a NetworkX module, which plot the graph based on a set of given coordinates. The coordinates are calculated based on each concept's weight, or number of objects present in its extent. The y-axis of a given node increases incrementally with respect to this weight such that the uppermost root node contains all objects, and the levels progress downwards. All concepts within a given row are centered with respect to the number of nodes within that row. We can observe the result in Figure 1, which draws from the dataset in Table 1.

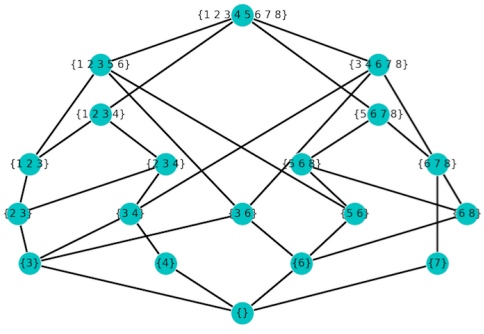
Since the visualization is a static image, options for the user to customize arrangement of nodes is not possible. However, the ordering of rows based on weight provides a pleasing structure for many simpler datasets. A possibility for the future is using a more effective algorithm for determining the row hierarchy that takes into account the edges between various levels. Due to the multiple parent and child edges, however, this is no trivial task.

Table 1: Formal context taken from [1]

	a	b	c	d	e	f	g	h	i
1	×	×					×		
2	×	×					×	×	
3	×	×	×				×	×	
4	×		×				×	×	×
5	×	×		×		×			
6	×	×	×	×		×			
7	×		×	×	×				
8	×		×	×		×			

Object rows numbered to match extents displayed in Figure 1.

Figure 1: Concept lattice output



7 Conclusion

This visualization tool utilized several algorithms that involved iteratively reducing the search space for finding vertices and edges. These approaches were limited due to size of the input matrix, yet were successfully able to compute simple matrices quickly. As distributed techniques such as MapReduce become more popular [3], it is often easy to continue using similar algorithms for formal concept analysis on more complex datasets.

Future areas of improvement lie heavily in the aesthetic side of concept lattice visualizations. Since minimizing crossings between rows of concepts is computationally expensive, it would be helpful for the image to be interactive. Through the use of libraries such as D3.js, constraints can be added to nodes such that a force-directed layout is created. The user could also be able to reposition nodes as they saw fit and access more information upon clicking on the nodes themselves.

References

[1] Rudolph Wille. *Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies*. Springer, 2005.

[2] Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the 8th International Conference on Formal Concept Analysis*, pages 312–340, Berlin, Heidelberg, 2010. Springer.

[3] Biao Xu, Ruairí Fréin, Eric Robson, and Mícheál Ó Foghlú. Distributed formal concept analysis algorithms based on an iterative mapreduce framework. In *Formal Concept Analysis: 10th International Conference*, pages 292–308, Berlin, Heidelberg, 2012. Springer.