

Engineering

Industrial & Management Engineering fields

Okayama University

Year 2001

A visualization method for knowledge
represented by general logic programs

Mariko Sasakura
Okayama University

This paper is posted at eScholarship@OUDIR : Okayama University Digital Information
Repository.

<http://escholarship.lib.okayama-u.ac.jp/industrial-engineering/38>

A visualization method for knowledge represented by general logic programs

Mariko Sasakura

Department of Information Technology, Okayama University
Tushima 3-1-1, Okayama, 700-8530, Japan
sasakura@momo.it.okayama-u.ac.jp

Abstract

In this paper, we describe a visualization method for general logic clauses as the first step of a visualization of logic programs. Since we think inclusion is essential to represent an outline of knowledge, we propose a method based on Euler diagrams to visualize inference rules represented as a set of general logic clauses which consist of literals with no variable. We develop a prototype system and show that complex rules which are hard to understand in text are easy to understand by diagrams.

1. Introduction

We are interested in visualizing an outline of knowledge intuitively. In this paper, we discuss a visualization method, which we call *inclusion diagrams*. An inclusion diagram is an extension of an Euler diagram. We can visualize general logic programs by inclusion diagrams.

We have known diagrams help us to understand complex or huge amount of information. For example, Euler diagrams are often used to explain sets theory for education. Also, Euler already found an inference rule can be represented by sets so they can be represented by Euler diagrams. However, Euler diagrams have two drawbacks to represent a set of large amount of inference rules. The first is that Euler diagrams are very hard for human to draw in hand if there are many sets. The second, that is more essential than the first, is that Euler diagrams can represent relation among only three sets precisely. It is very difficult for Euler diagrams to represent all combinations among more than three sets.

In spite of the drawbacks, Euler diagrams are very useful for us to grasp relations between sets. Therefore, we try to extend Euler diagrams and to use for visualizing relations between large amount of sets. For the first drawback, we develop automatic drawing system for diagrams. On the second drawback, we will draw intuitive diagrams instead of precise diagrams. So, we combine precise diagrams on less than four sets and intuitive diagrams on more than three

sets so that we draw diagrams on relations among sets with no misunderstanding.

In this paper, we consider a general logic program as a directed graph, and visualize it by a method based on Euler diagrams.

A general logic program is defined as a set of general program clauses. A general program clause is defined as the following:

$$A \leftarrow B_1, \dots, B_m, \sim C_1, \dots, \sim C_n (m \geq 0, n \geq 0)$$

where $A, B_1, \dots, B_m, C_1, \dots, C_n$ are atoms and \sim means negation. The left side of \leftarrow is called as “head”, and the right side is called as “body”.

In this paper, for the first step to visualize a general logic program, we deal with the clauses of which body consists of a literal. And to make a problem simple, we assume all literals have no variable. So, the clauses we deal with in this paper like the following:

$$A \leftarrow B$$

or

$$A \leftarrow \sim C$$

In section 2, we explain basic ideas of inclusion diagrams which we propose in this paper. In section 3, we show a prototype system of inclusion diagrams and mention some implementation issues. We mention related works in section 4. And we give a conclusion in section 5.

2. Ideas

2.1. Euler diagrams

Euler diagrams represent relations between sets. For example, the region in the circle of figure 1 represents a set A . The region out of the circle and in the rectangle represents a set of not A .

A general program clause can be represented by Euler diagrams. For example, there is a general program clauses

$$A \leftarrow B$$

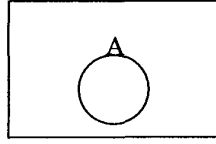


Figure 1. Euler diagrams

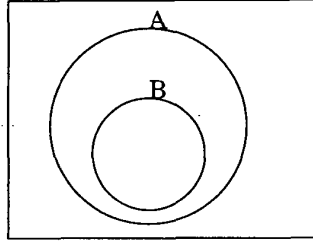


Figure 2. Euler diagrams for a general program clause

As well-known, this can be written $A \vee \neg B$. If A is a set of cases in which A is true and B is a set of cases in which B is true, then the clause is equal to:

$$A \supseteq B$$

Thus, this can be represented by Euler diagrams as figure 2.

2.2. Inclusion diagrams

Euler diagrams are very useful to grasp relations between sets intuitively. However, they have the following problems to visualize general program clauses.

- In Euler diagrams, we divide a space into two regions by a circle that represents a set. If there are several sets, we expect a space will be divided into the number of the combination of the sets. However, for more than three sets, Euler diagrams cannot represent all combinations.
- In Euler diagrams, a “not” set like $\sim A$ is represented as the region of out side of a circle. Thus, it is not easy to draw a diagram like $B \leftarrow \sim A$.

Therefore, we design inclusion diagrams as the following.

1. A circle corresponds to a literal in a general program clause. The inside of a circle represents the correspond literal is true. The outside of a circle represents the correspond literal status is unknown. For example, the

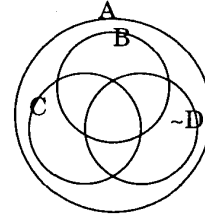


Figure 3. An inclusion diagrams for three nodes with negation

inside of the circle corresponds to a literal A means the cases of A is true. Similarly, the inside of the circle corresponds to a literal $\sim A$ means the cases of A is false.

2. We prefer intuitive understanding to preciseness. Therefore inclusion diagrams does not guarantee to represent precise information about sets if there are more than three sets in a diagram. They may allow users to grasp relations between sets intuitively. But, if there are less than four sets, inclusion diagrams can represent precise information about the sets. Users can understand precise relations among the sets by the diagram. Also, a positive literal and a negative literal on the same atom should not have intersections in logic, but we don't mind if they have intersections in a diagram.
3. To avoid drawing complex diagrams, we make several diagrams to represent the whole sets represented by general program clauses. Each of the diagrams represents some sets, and every set is represented in at least one of the diagrams. The details of how we decide which set will be drawn in which diagram will be mentioned in section 3.

For example, if there are general program clauses:

$$A \leftarrow B$$

$$A \leftarrow C$$

$$A \leftarrow \sim D$$

then an inclusion diagram is figure 3. B , C and $\sim D$ are drawn similarly.

In the case of four sets, we make figure 4, for example. This figure gives us intuitive understanding about four sets, but it is not precise. The regions correspond to $A \wedge C$ but don't care about B and D , and $B \wedge D$ but don't care about A and C are not represented.

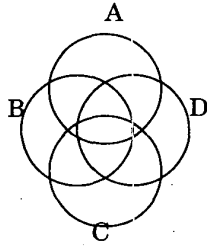


Figure 4. An inclusion diagrams for four nodes

Here, we should show whether inclusion diagrams can represent any logic program or not. It is trivial that any logic program is represented as directed graphs. In the directed graphs, a literal is represented as a node. Therefore, we consider whether inclusion diagrams can represent any directed graph produced by logic programs.

Inclusion diagrams draw the relation between nodes only by the inclusion of circles. So, they cannot draw general directed graphs. When there are loops in directed graphs, we cannot generate inclusion diagrams. Therefore, consider whether logic programs produce loops in the correspond directed graphs or not.

Case 1: loops with direction

For example, the following program clauses produce a loop.

$$A \leftarrow B$$

$$B \leftarrow A$$

We assume A is a set of cases in which A is true and B is a set of cases in which B is true. Then, above clauses can be rewritten as:

$$A \supseteq B \wedge B \supseteq A$$

These mean $A = B$. Therefore, we regard A and B as one literal. Thus we can avoid loops with direction.

Case 2: loops with no direction

There are cases that program clauses produce a loop with no direction. For example,

$$A \leftarrow B$$

$$B \leftarrow C$$

$$A \leftarrow C$$

produce a loop with no direction. In this case, we are puzzled as to where we draw C . But we think a proper way to

draw three clauses is that we ignore the third clauses, since the third clauses is lead by the first and second. Therefore if we have several passes between two literal, that is there is a loop with no direction, we cut the shorter pass then the loop will be removed.

Now, we know we can make inclusion diagrams for any program clauses of which body consists of 1 literal.

A general program clause can have negative literal in its body. Therefore, there are the cases that program clauses produce conflicts. In this paper, conflict means the situation which A and $\sim A$ also exist in same time. In a directed graph, it means A and $\sim A$ appear in the same pass. We can draw an inclusion diagram if there is a conflict in directed graph, but the diagram is not suite to our intuition. So, we remove clauses that cause conflicts. The details about conflicts will be discussed in section 3.1.

3. Implementation

We develop a prototype system that visualizes a logic program by our inclusion diagrams automatically. The system consists of two parts. The first part makes and manipulates directed graphs and the second part visualizes the graphs.

3.1. Graph manipulation part

The graph manipulation part of the prototype system

1. makes directed graphs from a set of general program clauses,
2. manipulates the graphs to draw diagrams,
 - (a) reduces loops,
 - (b) extracts conflicts, and
3. sets a level for each node of the graphs.

At first, we make directed graphs from a set of general program clauses. A node of the graphs corresponds to a literal in the general program clauses. An arc corresponds to a relation " \leftarrow ". If there is a rule $A \leftarrow B$, then we call A is a child of B and B is a parent of A . The node that has no child is called a leaf node. The node that has no parent is called a root node.

Then, we check and reform the graphs if there is any loop or conflict in the graphs.

Regarding a set of general program clauses as a directed graph, loops with direction in the clauses make a strongly connected component in the graph. We extract a strongly connected component from a directed graph and replace it as a node. The algorithm to find a strongly connected component in a directed graph is known[8].

We deal with loops with no direction in the drawing part. So we will discuss it in section 3.2.

To find a conflict we mentioned in section 2.2, we check if we can reach from a positive literal to a negative literal of the same atom or vice versa. For example, if there are A and $\sim A$ in the same directed graph and we can reach from $\sim A$ to A , they cause a conflict. In practice, after making directed graphs, we check that all negative literals have their positive literal as an ancestor. Because negative literals must be appeared in a body of a clause, so a node corresponds to a negative literal must be a root node. If we find a conflict, we cut the pass from the negative literal to the positive literal. Thus we can remove the conflict.

At last, we set a level for each node. The level decides which node will be drawn in which diagram, when we make several diagrams to represent a directed graph. A level of a node will be decided the following way:

1. The level of a leaf node is 0.
2. If the level of a node is n , then the level of parents nodes of it are $n + 1$.
3. If different number of levels will be assigned to a node, we choose bigger one.

3.2. Drawing part

We will draw a set of general program clauses as several diagrams. The number of diagrams must be not too many because many diagrams confuse users. But, we should also avoid making a complex diagram which has too many nodes in it. Therefore, basically, we draw nodes that are in the same level in a diagram. If users want to see the parents of one of the nodes, they select a menu for the node then we make a new diagram for the parents node. But, in the following cases, we draw parent nodes in the same diagram of the children nodes.

- There is only one node in a diagram.
- The all nodes drawn in a diagram have the same parents.

Now, we explain how we draw a diagram from a directed graph. A node is represented as a circle. First, we draw all leaf nodes. If there is only a leaf node or all leaf nodes have the same parents nodes, then we draw the parents nodes in the diagram. An algorithm to draw a diagram is shown in table 1. This algorithm will draw circles until the condition is not satisfied. So, in some cases there will be too many nested circles in a diagram. Therefore we explicitly restrict the number of recursive calls of procedure draw. Currently we restrict the number as 5 experimentally.

In the procedure drawcircles, we may draw several nodes. The node in the list of the parameter of drawcircles should not include the other node in the list. If there is such a node, it means there is a loop with no direction. Therefore, we check there is such a node in the list at the first of the procedure drawcircles. If there is, we remove the included node from the list.

Then, we draw the nodes in the list like an Euler diagram. If there are A and C in the list, a diagram is figure 5. Similarly, 10 nodes should be represented as figure 6. To make this figure, we draw 10 circles shifting a center of circles on a circle by $2 * \pi / 10$ radian.

3.3. Examples

In this section, we show two example diagrams generated by our prototype system. Our prototype system is implemented by Java.

Example1 shows how negative literals are drawn in a diagram. The diagram of this example is shown in figure 7. In this example, $B \leftarrow \sim A$ and $A \leftarrow B$ cause a conflict. So, these two clauses are not shown in a diagram. The inmost circles of the diagrams are A , C , D and $\sim C$. As we can see in the diagram, negative literals are drawn as same as positive literals. There should be no intersection between C and $\sim C$, but in the diagram there is. It is too complicated to draw precise diagrams. So, we don't mind if C and $\sim C$ have an intersection. We think such intersections are not obstacles to users' intuitive understanding.

Example2 is a a little big example to feel usefulness of the visualization. Table 3 is an example that consists of 14 rules. Even 14 rules, it is not easy to grasp an outline of rules written in text. Our prototype system visualizes this example as figure 8, 9, 10. The first figure is figure 8. C and D have parents that are not appeared in this figure. So, the menu "More..." is appeared to allow users show their parents. If we select C from the menu then figure 9 is appeared. If we select D , figure 10 is appeared.

4. Related Works

In this paper, we describe a visualization method for logic programs. There have been several studies on visualizing logic programs. However, almost of them propose visual logic programming[7, 5, 6, 1], that is, they visualize syntax of logic programs.

We are influenced by the studies in [2]. However, our work differs from logic reasoning with diagrams. A theoretical background on Euler diagrams are discussed in [4].

In this paper, we visualize a logic program as a directed graph. Although there are many studies in graph drawing[3], we don't find a study about graph drawing based on Venn/Euler diagrams.

Table 1. An algorithm to draw a diagram

```
list := a list of leaf nodes;
draw(list);

procedure draw(list)
begin
  n = the number of list;
  if n = 0 then return
  else if n = 1 then
    begin
      parent := a list of parents of list;
      draw(parent);
    end
  else if parents of all members of list are the same then
    begin
      parent := a list of parents of list;
      draw(parent);
    end
  end
drawcircles(list);
end
```

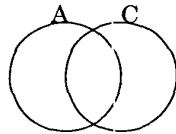


Figure 5. 2 nodes

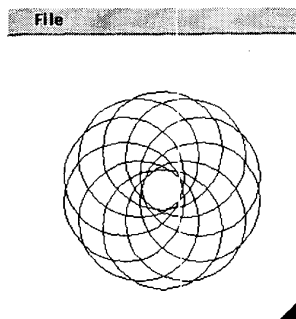


Figure 6. 10 nodes

Table 2. Example1

```
B ← ~ A
E ← ~ A
A ← B
B ← E
E ← C
E ← ~ C
E ← D
```

Table 3. example: 14 rules

```
G ← A   H ← F
H ← B   F ← H
J ← B   J ← H
A ← C   C ← I
A ← D   D ← J
A ← K   A ← L
F ← E   A ← M
```

5. Conclusion

In this paper, we describe a method to visualize a restricted logic program. Our method is based on Euler diagrams. We show examples generated by our prototype system. In future, we will expand this method to general logic programs.

References

- [1] J. Agusti, J. Puigsegur, and D. Robertson. A visual syntax for logic and logic programming. *Journal of Visual Languages and Computing*, 9(4):399–427, August 1998.
- [2] G. Allwein and J. Barwise, editors. *Logical Reasoning with Diagrams*. Oxford University Press, 1996.
- [3] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Plentice Hall, 1999.
- [4] E. Hammer and S.-J. Shin. Euler and the role of visualization in logic. *Languages, Logic and Computation: The 1994 Moraga Proceedings*, pages 271–286, 1994.
- [5] D. Ladret and M. Rueher. Vlp: a visual logic programming language. *Journal of Visual Languages and Computing*, 2(2):163–188, 1991.
- [6] M. Najork. Programming in three dimensions. *Journal of Visual Languages and Computing*, 7(2):219–242, 1996.
- [7] L. Pau and H. Olason. Visual logic programming. *Journal of Visual Languages and Computing*, 2(1):3–15, 1991.
- [8] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

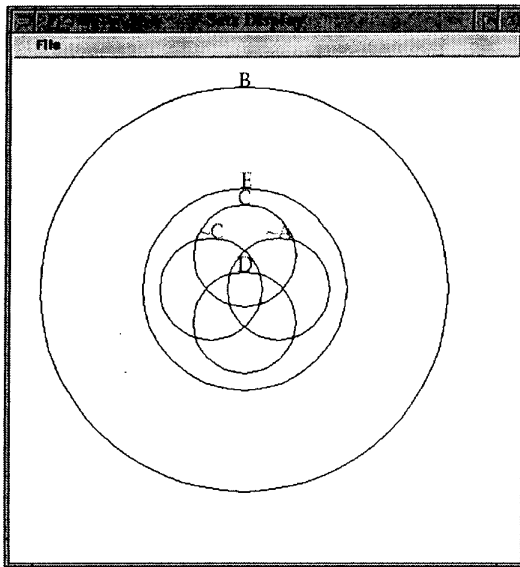


Figure 7. A diagram for example1

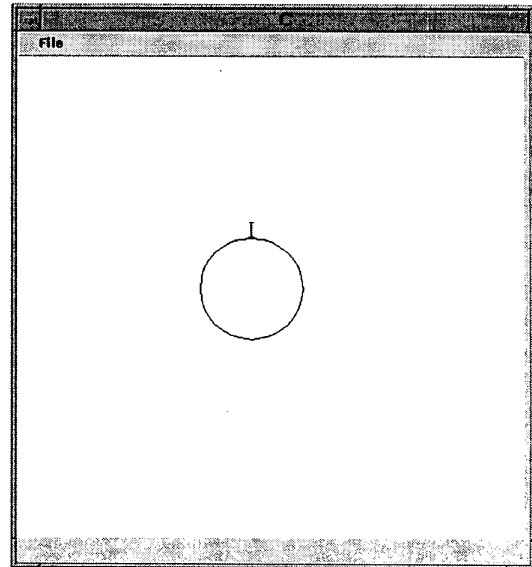


Figure 9. The diagram for node C of example2

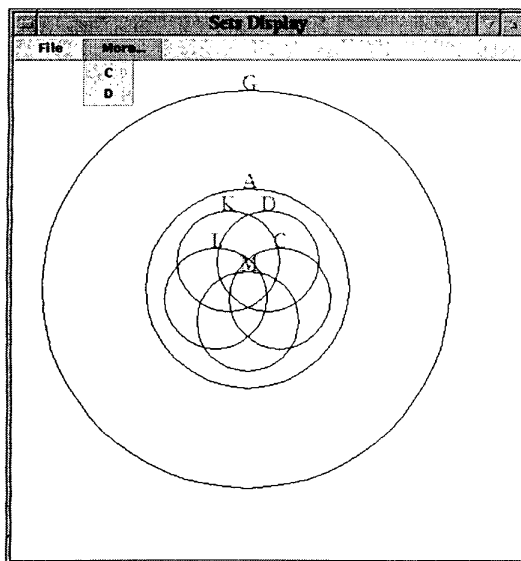


Figure 8. The first diagram for example2

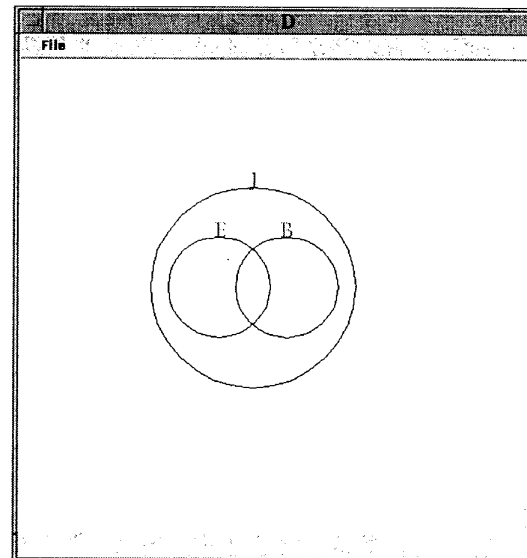


Figure 10. The diagram for node D of example2