Research Contribution

Knowledge Table: An Approach to Speeding up the Search for Relational Information in Knowledge Base

TORAMATSU SHINTANI*

For efficient knowledge utilization, it is necessary to keep relationships between knowledge such as isa (or class-inclusion) relationships, hasa (or part-whole) relationships, and data dependencies. These relationships are useful for managing and manipulating knowledge to solve complex problems in connection with belief revision, default reasoning, multiple inheritance within networks, and so on. Unfortunately, keeping and using them can be ineffective and expensive when large numbers of relationships are involved. In this paper, a knowledge table is introduced as a structure that allows effective searching for relationships in knowledge bases. The knowledge table is a table-like representation for keeping relationships that precludes the need for depth-first searching. To create the table, we use logical bitwise computations for searching and for representing relationships. This greatly speeds up the search for the relationships and makes it easy to find loop structures among the relationships.

1. Introduction

Recently the importance of knowledge utilization as well as knowledge representation and acquisition has been recognized and discussed [14]. To realize efficient knowledge utilization, it is necessary to keep and manage relationships between knowledge in an effective manner. Current Artificial Intelligence (AI) programs contain well-known relationships such as isa (class-inclusion), hasa (part-whole) [19, 23] and data dependencies [6, 9]. These relationships are useful for managing and manipulating knowledge to solve complex problems in connection with belief revision [6], default reasoning [8], multiple inheritance within networks [3], and so on. For example, belief revision [2, 11] has two main processes for maintaining a knowledge base: searching for current false assumptions and removing them. Searching can be performed effectively by using dpendency-directed backtracking (or nonchronological backtracking) [12]. When assumptions are removed in order to maintain a knowledge base, it is necessary to remove or modify other assumptions that depend on them. To maintain the relationships, we need to represent graphs in a computer. There are two common representations for a graph: adjacency matrices and lists [1]. Each representation has advantages and disadvantages as shown in Table 1.

The adjacency list representation is convenient for representing these relationships, and is frequently used for representing relationships in AI programming.

*International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED.

Table 1 Advantages and disadvantages of the two graph representations.

	Main advantages	Main disadvantages
Matrix	(1) It is convenient for getting knowledge on whether certain arcs are present.	(1) Keeping a graph takes up much storage area. (2) Initializing a matrix takes time.
Lists	(1) It does not need much storage area to represent a graph structure. (2) It is convenient for representing directed tree structures.	(1) It is inconvenient for getting knowledge on whether certain arcs are present. (2) It is inconvenient for grasping the whole relational structure.

Generally, the representation is not effective when a large number of relationships is involved, since it is necessary to check the relationships by using a depth-first search. This consumes much time. The knowledge table proposed in this paper was designed to speed up the search and manage relationships effectively and inexpensively in a knowledge base. We also propose a mechanism for the knowledge table, and present details of methods for constructing knowledge tables in C-Prolog [7] on the VAX11/780.

There are five sections. Section 2 outlines the basic idea of the knowledge table, and describes its basic structures and manipulations. Section 3 gives detailed procedures for manipulating the knowledge table. Section 4 describes procedures for detecting loop structures in a knowledge table. Section 5 gives experimental results of their performance. Section 6 consists of some concluding remarks.

2. The Knowledge Table

In this section, we propose a new representation for managing relationships. We call this representation a "*knowledge table.*" The knowledge table introduces the advantages of (1) the adjacency matrix, which enables us to check the adjacencies of objects easily, (2) the reachability matrix, which enables us to check reachability between objects without depth-first searching, and (3) the adjacency list, which saves storage for representing graph structures.

2.1 The Basic Structure

In order to realize the above features, we first represent the relations of knowledge by a table, which integrates an adjacency matrix and a corresponding reachability matrix, and then reduce it to a set of list expressions. For the reasons given above, we adopt the relational view of data in order to represent a knowledge table. The knowledge table keeps relations between objects (or knowledge) by representing information on the adjacency and reachability matrix of a

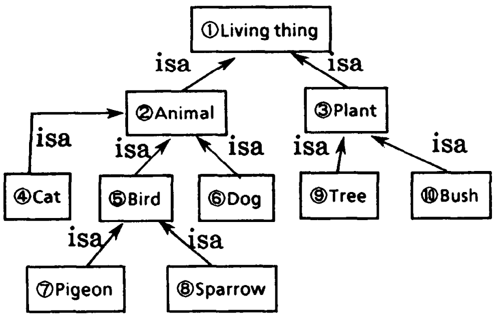

Fig. 1(a) Example of isa relationships.

	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
①Living thing										
②Animal	1									
③Plant	1									
④Cat	2	1								
⑤Bird	2	1								
⑥Dog	2	1								
⑦Pigeon	2	2	0	0	1					
⑧Sparrow	2	2	0	0	1					
⑨Tree	2	0	1							
⑩Bush	2	0	1							

Fig. 1(b) Example of a knowledge table.

directed graph. For example, the isa hierarchy in Fig. 1(a) can be represented in the knowledge table in Fig. 1(b), which has information on the adjacency and reachability of the isa hierarchical graph. Each element of the table has a number "1", "2", or "0", which indicates (1) *adjacency*, (2) *reachability*, or (3) *no relation* between the objects (or nodes), respectively. The unfilled elements in the table in Fig. 1(b) should have "0"s, but these are omitted because of the economical list representation of the table described below. Adjacency can be expressed by an arc (m, n) of a hierarchical graph, where node m is called a parent of node n, and node n is called a child of node m. If node m is not adjacent to node n and there is a path of arcs from m to n, then their relationship is represented by reachability, and node m is called an ancestor of node n and node n is called a descendant of node m. In Prolog programming, the 10-column and 10-row table can be represented by the following 10 assertions of an 11-place predicate [4]:

```
isa_table(living_thing, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(animal,       1, 0, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(plant,        1, 0, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(cat,          2, 1, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(bird,         2, 1, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(dog,          2, 1, 0, 0, 0, 0, 0, 0, 0, 0).
isa_table(pigeon,       2, 2, 0, 0, 1, 0, 0, 0, 0, 0).
isa_table(sparrow,      2, 2, 0, 0, 1, 0, 0, 0, 0, 0).
isa_table(tree,         2, 0, 1, 0, 0, 0, 0, 0, 0, 0).
isa_table(bush,         2, 0, 1, 0, 0, 0, 0, 0, 0, 0).
```

In this representation, the predicate isa_table represents the name of the table, and each row of the table can be uniquely identified by the first argument of these assertions. The other arguments represent the column positions of the table. This relational representation is not efficient, because it consumes too many computational resources, and it does not have sufficient flexibility for the table to be extended.

From a computational point of view, we can represent the same information by omitting unnecessary "0"s, as follows:

```
isa_table(table_column, [living_thing, animal, plant,
                         cat, bird, dog, sparrow,
                         tree, bush]).
isa_table(animal,       [1]).
isa_table(plant,        [1]).
isa_table(cat,          [6]).
isa_table(bird,         [6]).
isa_table(dog,          [6]).
isa_table(pigeon,       [266]).
isa_table(sparrow,      [266]).
isa_table(tree,         [18]).
isa_table(bush,         [18]).
```

In this representation the first assertion records a list in which the order of rows of the knowledge table is kept. The first argument of the other assertions identifies the

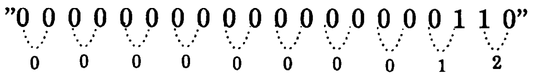

Fig. 2 The binary digit string.

rows of the table and the second argument, which is a list, represents the significant relational information of the rows. We call the list *an adjacency-reachability list*. In this case, it is not necessary to represent the first row "living_thing" of the table because it has no significant relational information. The fourth row of the table is represented by the assertion "isa_table(cat, [6])". Its second argument, which is the adjacency-reachability list "[6]", represents the sequence of column elements "2, 1, 0, 0, 0, 0, 0, 0, 0, 0" of the row. That is to say, the sequence of elements of the row can be represented by one decimal digit in the adjacency-reachability list. We consider the sequence of the elements to be the binary digit string "00000000000000000110", where one element of the row is indicated by two binary digits and arranged in the opposite order. The first, second, and remaining elements of the row, namely 2, 1, and 0 . . . 0, are represented by the last two binary digits of the string "10", the second last pair of binary digits "01", "00" . . . "00", as in Fig. 2. Then we can represent the sequence of column elements of the row by using the decimal digit 6, since the binary digit string "00000000000000000110" is considered to be the binary number "110", which is the decimal number 6.

Using the adjacency-reachability list we can overcome the disadvantages and exploit the advantages of the two representation for graphs, as shown in Table 1. In the following sections. We shall give details of its manipulations.

2.2 The Basic Idea of the Manipulations

To manipulate a knowledge table, we need to realize the following basic functions for adjacency-reachability lists of the table: (1) setting, (2) checking, and (3) resetting of Nth bit of a binary digit string.

Here, we describe the outline of implementation for these basic functions in C-Prolog. For case (1), we can use the predicate "$\vee$", which achieves integer bitwise disjunction. We can set a bit at the Nth position of a binary digit string S_d by bitwise disjunction between S_d and integer 2^{N-1}.

For case (2), we can use the predicate "$\wedge$", which achieves integer bitwise conjunction. To check a bit at the Nth position of a binary digit string S_c, we first get a number N_c by bitwise conjunction between S_c and integer 2^{N-1}, and then check it as follows: The bit is set at the Nth position of the string S_c if the number N_c is 1; otherwise ($N_c=0$), the bit is not set.

For case (3), we can use the predicates "$\wedge$" and "$\setminus$". The predicate "$\setminus$" achieves integer bitwise negation. we can reset a bit at the Nth position of a binary digit string S_r by bitwise conjunction between S_r and integer N_r, where N_r is obtained by applying integer bitwise negation to integer 2^{N-1}.

To clarify the above description, a few examples are given in Fig. 3. Example 1 shows the setting of a bit at the fifth position of the binary number "1001010" (that is, the decimal number 74). The result is the decimal number 90, which is "1011010" as a binary number. Example 2-1 shows that the fourth bit of the decimal number 74 is set. Example 2-2 shows that the fifth bit of the decimal number 74 is not set. Example 3 shows the resetting of a bit at the fourth position of the decimal

```
| ?- N is 2 ^ (5-1), Result is 74 \/ N.          % Example 1

N = 16
Result = 90

yes
| ?- N is 2 ^ (4-1), Result is 74 /\ N.          % Example 2-1

N = 8
Result = 8

yes
| ?- N is 2 ^ (5-1), Result is 74 /\ N.          % Example 2-2

N = 16
Result = 0

yes
| ?- N is 2 ^ (4-1), Result is (\N) /\ 74.       % Example 3

N = 8
Result = 66

yes
```

Fig. 3 Examples of basic manipulations for knowledge tables.

number 74. The result is the decimal number 66, which is "1000010" as a binary number. By extending these basic functions, we can get functions for a knowledge table that can manipulate two bits of a binary digit string at a time.

3. Manipulating the Knowledge Table

A knowledge table is composed of a set of assertions whose second arguments are adjacency-reachability lists. An adjacency-reachability list is composed of some positive integers. These integers represent elements of columns (or rows) of a knowledge table.

Here, for convenience of description, we define terms for the knowledge table. We call integers in an adjacency-reachability list *list-elements*. Elements are elements of a knowledge table, which identify its rows (or columns). Adjacency-reachability lists are kept orderly. These lists are arranged according to the order of rows (or columns) of a knowledge table. To keep the order, we use a list whose elements are the names of the rows (or columns) that have adjacency-reachability lists. We call the list *the L_k list*.

In this section, we define and show the frameworks of procedures for managing a knowledge table by using a C-Prolog syntax and a few of its built-in (system-defined) procedures (or predicates). Since Prolog is a simple but powerful and practical tool for programming in logic, we can show the procedures with clear, readable, and concise Prolog programs.

3.1 Adding a New Relation

We show the procedure for adding a new relationship to a knowledge table, that is, the addition of a new arc (n, m) where n is the tail node and m is the head node of the arc (n, m). We can define it as follows:

Procedure A1
```
add_arc_to_Knowledge_Table(Head_node, Tail_node,
                                  Knowledge_table):-
   (exist_node(Tail_node, Knowledge_table),
    find_position(Tail_node, Knowledge_table,
                  T_position)
    ;
    make_position(Tail_node, Knowledge_table,
                  T_position),         %Procedure A2
    record_arc_in_Knowledge_table(Tail_node,
                  T_position,[ ], Knowledge_table)),
   (exist_node(Head_node, Knowledge_table),
    find_position(Head_node, Knowledge_table,
                  H_position),
    ;
    make_position(Head_node, Knowledge_table,
                  H_position)),
   (take_adjacency_reachability_list(Tail,
                  Knowledge_table, A_r_list_of_T),
    transform_01_to_10(A_r_list_of_T,
                  A_r_list_of_T_2),    %Procedure A3
    set_rtype_at_T_position(A_r_list_of_T_2,
          T_Position, 1,
          New_list_of_Head)      %Procedure A4
    ;
    set_rtype_at_T_position([ ], T_Position, 1,
                          New_list_of_Head)),
   (find_sub_knowledge(Head_node, Knowledge_table,
          Descendants),          %Procedure A5
    set_type_at_T_positions(Descendants, T_position,
          Knowledge_table, 2)    %Procedure A1.1
    ;
    true),
   record_arc_In_Knowledge_table(Head, node,
                  H_position, New_list_of_Head,
                  Knowledge_table).
```
Procedure A1.1
```
set_type_at_T_positions([ ], T_positions,
                         Knowledge_table, Type).
set_type_at_T_positions([N1|N2], T_positions,
                         Knowledge_table, Type):-
   find_position(N1, Knowledge_table, N1_position),
   take_adjacency_reachability_list(N1, A_r_list),
   set_rtype_at_T_position(A_r_list, T_position, Type,
                  New_list),         %Procedure A4
   record_arc_in_Knowledge_table(N1, N1_position,
                  New_list, Knowledge_table),
   set_type_at_T_positions(N2, T_positions,
                         Knowledge_table, Type).
```
Procedures have comments which are made between the operator "%" and the end of a line. For example, Procedure A1 has five comments, which show the names of procedures to be referred to.

In Procedure A1, the predicate

exist_node(Node, Knowledge_table)

is used to test whether a node currently belongs to a knowledge table, where the first and second arguments represent the node and the knowledge table respectively. The predicate

find_position(Node, knowledge_table, Position)

finds the position number of a node in the L_k list of a knowledge table, where the first, second, and third arguments represent the node, the knowledge table, and the position number, respectively. The predicate

take_adjacency_reachability_list(Node, Table, List)

retrieves the adjacency-reachability list of a node from a knowledge table, where the first, second, and third arguments represent the node, the knowledge table, and the list, respectively. The predicate

record_arc_in_Knowledge_table(Node, P, A_r_list, Table)

is used to record the adjacency-reachability list *A_r_list* of a node *Node* is a knowledge table *Table* by inserting the node at the *P*th position in the L_k list of the knowledge table and by asserting the adjacency-reachability list *A_r_list* of the node, where the first and fourth arguments represent the node and the knowledge table, respectively.

3.2 Retrieving and Removing Relations

Procedure A5 in Procedure A1 is used to retrieve the sub-nodes of node *Nj* from a knowledge table. The procedure can be specified as in Appendix A. The mechanism of this procedure is to check all the adjacency-reachability lists of nodes in a knowledge table to find whether the relation type "1" or "2" exists at the *N*th element position in each of the adjacency-reachability lists, where the position number *N* is determined by the position of node *Nj* in the L_k list of the table. This mechanism is equivalent to checking a row of a knowledge table sequentially to find whether a relation type exists at the *N*th column position of the row.

The procedure can be changed simply to retrieve the children of a node from a knowledge table by modifying the predicate "find_sub_knowledge2" so that it finds only the relation type "1".

Procedure A6 for finding super nodes can be specified as in Appendix A. Using the predicate "find_super_knowledge", we can find the parents of a node if the third argument is "1". We also find the ancestors of a node if the third argument is "2". This procedure is equivalent to checking a column of a knowledge table sequentially to find whether a relation type exists at each row position of the column.

To clarify the description, a few examples are presented in Fig. 4. The predicate "add_to(Tail, Heads, Kt)" is equivalent to the predicate "add_arc_to_Knowledge_Table" in Procedure A1, and the nodes *Heads* are considered to be parents of the node *Tail* in knowledge table *Kt*. The predicate "children(Node, Children, Kt)" retrieves the children of a node from knowledge table *Kt*, whereas the predicate "descendants(Node, Descendants, Kt)" retrieves descendants of a node from *Kt*. These two predicates are obtained from Procedure A5 by modifying it. The predicate "subs" is equivalent to the predicate "find_sub_knowledge" in Procedure A5. The predicate "parents(node, Parents, Kt)" retrieves parents of a node from knowledge table *Kt*, whereas the predicate "ancestors(node, Ancestors, Kt)" retrieves ancestors of a node from *Kt*. These two predicates are specified simply by using the predicate "find_super_knowledge" in Procedure A6. The predicate "supers(Node, Supers, Kt)" retrieves all the super knowledge of the node from knowledge table *Kt*. This predicate is also obtained from Procedure A6 by modifying it.

In order to remove a node and its relations from a knowledge table, we can specify the following procedure by using Procedures A5 and A1.1.

```
remove_from(node, Knowledge_table):-
  find_position(Node, Knowledge_table, Position),
  find_sub_knowledge(Node, Knowledge_table,
          Descendants),          %Procedure A5
  remove_adjacency_reachability_list(Node,
          Knowledge_table),
```

```
| ?- add_to(n3,[n1,n2],isa).
n3

yes
| ?- add_to(n7,[n3,n4,n5],isa).
n7

yes
| ?- add_to(n8,[n7,n6],isa).
n8

yes
| ?- children(n4,X,isa).

X = [n7]

yes
| ?- descendants(n4,X,isa).

X = [n8]

yes
| ?- subs(n2,X,isa).

X = [n3,n7,n8]

yes
| ?- parents(n7,X,isa).

X = [n3,n4,n5]

yes
```

```
| ?- ancestors(n7,X,isa).

X = [n1,n2]

yes
| ?- supers(n8,X,isa).

X = [n1,n2,n3,n4,n5,n7,n6]

yes
```

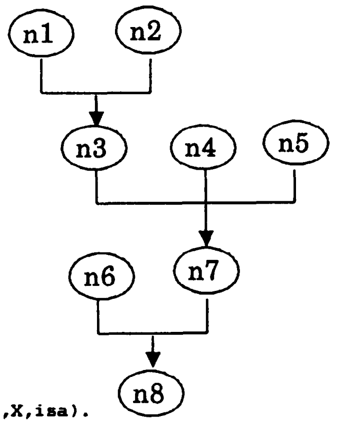

Fig. 4 Examples of retrieving relations from a knowledge table.

```
set_type_at_T_positions(Descendants, Position, 0).
                                    %Procedure A1.1
```

In the procedure, the predicate

```
remove_adjacency_reachability_list(Node,
                                    Knowledge_table)
```

is used to remove the adjacency-reachability list of the node in the knowledge table.

3.3 Propagating Reachabilities

Procedure A3 in Procedure A1 is needed to propagate reachabilities effectively from parent nodes to its sub nodes. It transforms any "01" element (that is, adjacency) to a "10" element (that is, reachability) in an adjacency reachability list. Procedure A3 can be specified as follows:

Procedure A3

```
transform_01_to_10([ ], [ ]).
transform_01_to_10([E1|E2], New_list):-
   change_01_to_10(E1, New_E1),   %Procedure A3.1
   New_list=[New_E1|New_list2],
   transform_01_to_10(E2, New_list2).
```

Procedure A3.1

```
change_01_to_10(N1, N2):-
   Num=178956970,
   N2 is ((N1 « 1)∧Num)∨(N1∧Num)).
```

In Procedure A3.1, the decimal number

"178956970"

is used to represent the binary number

"1010101010101010101010101010".

Using this number, we can easily transform "01" elements to "10" elements in an adjacency-reachability list through efficient bitwise computations in Procedure A3.1. If we do not use the bitwise computations, we need to check and change all the bit elements of every list-element of an adjacency-reachability list. The bitwise computations help to speed up the search for elements.

procedure A4 in Appendix A is used to set a relation type of "0", "1", or "2" at the *N*th element position of an adjacency-reachability list. This procedure is also done effectively by using bitwise computations.

In C-Prolog [7] for the VAX11/780, one integer (positive decimal number) can represent 14 elements of a knowledge table, since a decimal number is treated as a sequence of 28 bits. If the number of the column (or row) elements of a knowledge table is greater than 14, the adjacency-reachability lists of the knowledge table are composed of more than one decimal number, as follows:

```
table(. . . , [77, . . . , 256]).
table(fish, [266, 6557, 256, . . . , 181]).
```

3.4 Reducing Memory Space

Procedure A2 is used to reduce memory space for representing the knowledge table. It does so by eliminating unnecessary elements that are used for keeping "0" relationships in adjacency-reachability lists. The procedure can be specified as follows:

Procedure A2

```
make_position(Node, Knowledge_table, Position):-
   take_LK_list(Knowledge_table, LK_list),
   (LK_list=[ ], Position=1
   ;
   (find_sub_knowledge(Node, Knowledge_table,
              Descendants),             %Procedure A5
    one_left_shift(Descendants, Knowledge_table),
    ;
    length(LK_list, Length),
    Position is Length+1)).
```

The predicate

```
one_left_shift(Node_set, Knowledge_table)
```

is used to left-shift columns elementwise. It shifts the adjacency-reachability lists by one place for all the nodes in the set *Node_set* in a knowledge table, where the second argument represents the knowledge table. An example is given to clarify the above description. we consider the directed graph in Fig. 5, which is represented by the knowledge table in Fig. 6.

We add to the knowledge table a new node "w", which is a parent of the node "a" already kept in the table. We put node "w" at the last position in the L_k list. We call this extension of a knowledge table *tail-ex-*

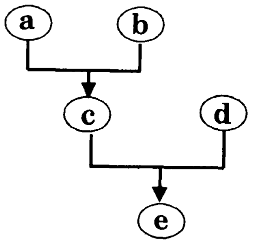

Fig. 5 Example of a directed graph.

	a	b	c	d	e
a					
b					
c	1	1			
d					
e	2	2	1	1	

Fig. 6 Knowledge table of the graph in Fig. 5.

	a	b	c	d	e	w
a	0	0	0	0	0	1
b						
c	1	1	0	0	0	2
d						
e	2	2	1	1	0	2
w						

Fig. 7 Tail-extended knowledge table of Fig. 6.

	w	a	b	c	d	e
w						
a	1					
b						
c	2	1	1			
d						
e	2	2	2	1	1	

Fig. 8 Head-extended knowledge table of Fig. 6.

Table 2 number of elements required for adding rel (T, H).

	$\lvert N_{sub}\rvert \neq 0$		$\lvert N_{sub}\rvert = 0$	
	$\lvert N_{sup}\rvert \neq 0$	$\lvert N_{sup}\rvert = 0$	$\lvert N_{sup}\rvert \neq 0$	$\lvert N_{sup}\rvert = 0$
Tail-extension	$(\lvert N_{sub}\rvert + 1) \times (\lvert N_{sub}\rvert + 2)$	$(\lvert N_{sub}\rvert + 1) \times (\lvert N_{sub}\rvert + 2)$	$\lvert N\rvert - \lvert N_{sup}\rvert + 1$	N_{posi}
Head-extension	$\lvert N_p\rvert$	$\lvert N_p\rvert + 1$	$\lvert N_p\rvert$	$\lvert N_p\rvert + 1$

tension of a table. Then, node "w" can be added to the knowledge table as in Fig. 7 by using Procedure A1 without using Procedure A2. The knowledge table in Fig. 7 has many unnecessary "0" elements to keep node "w".

On the other hand, to add the new node "w" to the knowledge table in Fig. 6, we can put it at the first position in the L_k list. We call this extension of a knowledge table *head-extension of a table*. Then, node "w" can be added to the knowledge table by using Procedure A1 together with Procedure A2, as in Fig. 8. Unnecessary "0" elements are eliminated in the knowledge table in Fig. 8, in contrast to that in Fig. 7, which is generated by the tail-extension method.

Table 2 summarizes the number of elements required to keep relationships when a new relationship rel (T, H) is added to a knowledge table, where "rel" indicates the name of the added relationship between T and H that has a direction from T (tail node) to H (head node); in other words, T is a parent of H. In Table 2, $\lvert N\rvert$ is the number of nodes in the knowledge table. $\lvert N_p\rvert$ is the number of nodes that have adjacency-reachability lists (or parents) in the knowledge table. $\lvert N_{sub}\rvert$ is the number of sub nodes of node H, which are descendants and children of node H. $\lvert N_{sup}\rvert$ is the number of super nodes of node H, which are ancestors and parents of node H. N_{posi} is the number of column position of node T in the knowledge table. These numbers are specified before rel(T, H) is added to the table, and their relationships are as follows:

$$\lvert N\rvert > \lvert N_p\rvert \geq \lvert N_{sub}\rvert, \quad \lvert N\rvert > \lvert N_{sup}\rvert, \quad \text{and} \quad \lvert N\rvert > N_{posi}.$$

If sub nodes of node H exist, it is generally more efficient to keep node T in the table by using the head-extension method to reduce the memory space for the table. Otherwise it should be kept in the table by using the tail-extension method, since the head-extension method needs more computational resources (such as applying the predicate shifted once to the left) than the tail-extension method. Procedure A2 is used to realize these ideas and save memory space when a node is added to the knowledge table.

4. Detecting Loop Structures

Procedure A1, described in Section 3.1, can detect a loop structure simply by modifying the function of the predicate "set_type_at_T_positions" (that is, Procedure A1.1) in the procedure. The predicate

```
set_type_at_T_positions(Nodes, Nth,
                        Knowledge_table, 2)
```

is used to set the reachability relationship "2" at the *N*th element position of adjacency-reachability lists of the nodes in the knowledge table. In this process, if a relationship number already exists at the position, it sets the relationship to "0" before setting the number "2" at the position. This setting is performed by using an integer bitwise disjunction, as discussed in Section 2.2.

If we only use the integer bitwise disjunction to set the number "2" instead of the original function, we can easily find a loop structure by checking whether the number "3" exists or not in the elements of adjacency-reachability lists. The number "3" appears in an element of an adjacency-reachability list of a node when the node has descendants some of which are also its parent nodes. The new relationship "3" is obtained by an integer bitwise disjunction between the numbers "1" and "2", which are used to indicate adjacency and reachability, respectively, in a knowledge table. We can easily check whether the number "3" exists or not by using the predicate

```
find_3_in_bit_string(Number)
```

where the argument is a decimal number used as a list-element of an adjacency-reachability list. It can be defined as follows:

```
find_3_in_bit_string(Number):-
  Test is (Number∧178956970)∧((Number≪1)
                                          ∧Number),
  Test>0.
```

The number 178956970 is discussed in Section 3.3.

An example is given to clarify the description and the process of making a knowledge table. Let us keep the graph as in Fig. 9 in a knowledge table, which contains a loop structure. We can keep the graph in a knowledge table by executing the directives in the following order:

```
:-add_to(c, [a, b], graph).   %Step 1
:-add_to(e, [c, d], graph).   %Step 2
:-add_to(a, [e], graph).      %Step 3
```

The knowledge table is constructed in the same order as in Fig. 9. In Fig. 9, setting reachabilities means setting the reachability relationship "2" in the elements of a knowledge table. TABLEs (3) and (5) are obtained by executing Procedures A3 and A4 in Procedure A1, which adds relations to a knowledge table. TABLE (6) is obtained by executing Procedures A5 and modified A1.1 in Procedure A1. These procedures are used to propagate reachabilities from parent nodes to their sub nodes. First, as in TABLE (6), the number "3" appears in the fifth column of the first row for node "a" when it propagates reachabilities from the node "a" to its sub nodes "a", "c", and "e", which can be found by using Procedure A5. Here we can check and find the loop structure by applying the predicate "find_3_in_bit_string" to the list-elements of the adjacency-reachability list of node "a". In Section 3.4, we discussed the

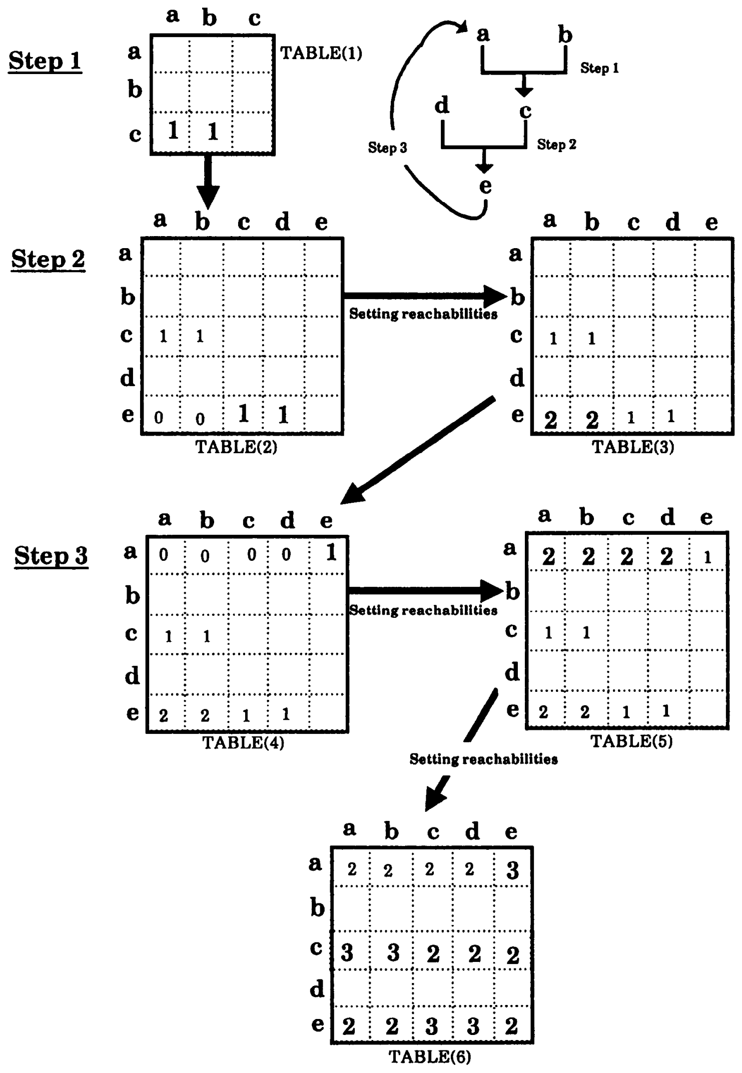

Fig. 9 The process of construction a knowledge table and finding a loop structure.

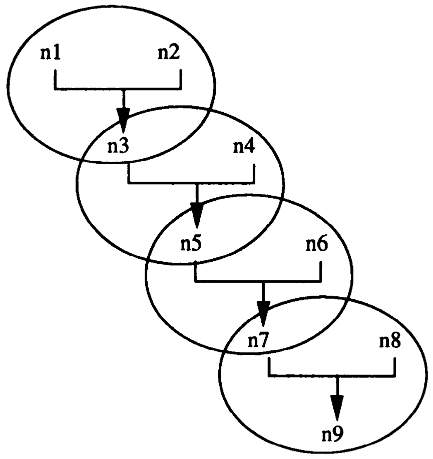

Fig. 10 Example of four relations.

number of elements necessary to keep a relationship in a knowledge table, and we summarized the discussion in Table 2. However, this does not hold if the knowledge table contains a loop structure.

5. Timing Data

We conducted a few experiments to see how fast the procedure runs. The program was written in C-Prolog for a VAX11/780. The experiments were conducted by setting up a sequence of linked tri-node relations. For example, the experiment for four relations is shown graphically in Fig. 10. One relation is composed of three nodes, where one is a child of the others, such that the next relation is also composed of three nodes, one parent of which is the previous child node. The relations in Fig. 10 were made by applying the "add_to" directives as follows:

```
:-add_to(n3, [n1, n2], graph).
:-add_to(n3, [n1, n2], graph).
:-add_to(n3, [n1, n2], graph).
:-add_to(n3, [n1, n2], graph).
```

To compare this with the knowledge table approach, we made another representation for keeping relations, which was realized by the program P in Appendix B. Program P functions like Procedure A1. However, it does not keep relations in the sameway as a knowledge table, but keeps them as binary assertions. It executes a depth-first search to find a relation. In Prolog programming, the representation in program P is usually used to represent relations. For example, in program P, the relations in Fig. 10 are represented by the following assertions:

```
graph(n3, n1).
graph(n3, n2).
graph(n5, n3).
graph(n5, n4).
graph(n7, n5).
graph(n7, n6).
graph(n9, n7).
graph(n9, n8).
```

Table 3 summarizes the results of the experiments. The compilation time means the CPU time necessary to load each of the files that contain add_to directives or assertions. The compilation time corresponds to the time needed to make a knowledge table or to assert binary representations for keeping the relations. The memory space means the memory space necessary to keep the relations. The supers and subs are predicates specified in Section 3.2. The search time for retrieving these relations is measured by the CPU time. The predicate supers was used to find all the super-nodes of the bottom node, which has no children. The predicate subs was used to find all the sub-nodes of the root node, which has no parents. In Table 3, the symbol "※" shows that the program P cannot search the super-nodes, because it causes local stack overflow of the C-Prolog system during execution.

The practical program we realized for a knowledge table includes an extra mechanism for checking and keeping loop structures, which is discussed in Section 4. In other words, the compilation time for a knowledge table includes the time taken to check loop structures.

Although the knowledge table method requires more memory space, as shown in Fig. 11, and more compilation time, which is required when a knowledge relation is added to a table, it is much faster than the binary

Table 3 Results of experiments.

Numbers of relations	Binary representation				Knowledge table			
	Compilation time (sec)	Memory spaces (byte)	Search time (sec)		Compilation time (sec)	Memory spaces (byte)	Search time (sec)	
			supers	subs			supers	subs
10	0.3	760	1.00	0.50	1.67	1.436	0.24	0.19
50	1.25	3,640	6.76	3.88	17.68	8,368	1.67	0.86
100	2.27	7,240	18.57	10.72	51.75	20,900	4.70	1.72
150	4.37	10,840	35.31	20.69	110.73	37,716	9.80	2.86
200	5.45	14,440	56.79	34.06	183.82	58,816	15.86	3.98
250	7.47	18,040	※	51.41	262.78	84,200	23.89	5.14

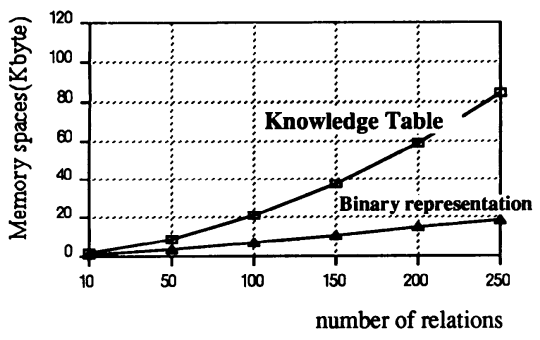

Fig. 11 Memory spaces.

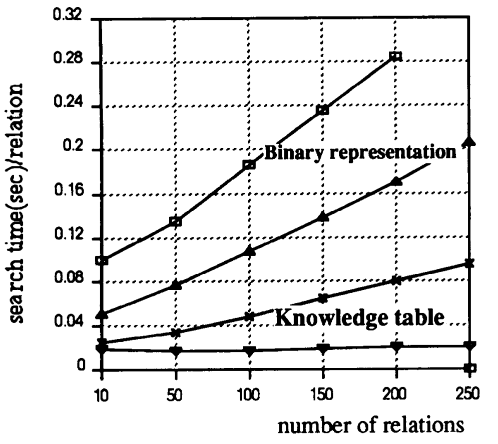

Fig. 12 Search time per relation.
□ supers (Binary representation)
△ subs (Binary representation)
× supers (Knowledge Table)
▽ subs (Knowledge Table)

representation method in finding the super-or sub-nodes of a node. This speed is particularly important when searching through a large knowledge base. Fig. 12 shows the search time (in seconds) per relation for the binary representation and for the knowledge table. The graphs in Figs. 11 and 12 are obtained by using the data in Table 3. We emphasize that the time required for finding sub-nodes in the knowledge table is linearly proportional to the number of relations. The ideal search performance is realized by fully exploiting the merits of the bitwise computation mentioned in Section 2.2. In other words, the simple computation makes it possible to check the relationships between nodes effectively.

For representing relationships in C-Prolog programming for the VAX11/780, the program for a knowledge table is more complicated than program P in Appendix B. The complication is partly due to the bitwise computations. If a row of a knowledge table can be represented with one integer, the program can be simplified and made faster than the current program for a knowledge table. If the number of the column elements of a knowledge table is greater than 14, the adjacency-reachability lists of the knowledge table are composed of more than one decimal number. The complication of the program is due to the management of the resulting lists. Despite this inessential complication, which is constrained by the design of the specific machine, the program is more efficient than program P in searching for super- or sub-nodes of a node. This shows the efficiency of bitwise computations. The search time for finding super nodes in the knowledge table is also faster than that of binary representation, as shown in Fig. 12. If we use a "bignum", the performance of the supers is effectively improved, since we can represent the rows of the knowledge table by using only one bignum. The bignum is an integer of potentially unbounded size, which is usually implemented and used in Lisp systems.

In practice, it is convenient to use both the knowledge table and the binary representation in an application, since the knowledge table requires more compilation time for the relationships, as shown in Table 3. The binary representation is applied for executing the applications as an interpretive mode in which the relationships are recorded quickly. The knowledge table is used to speed up the execution by compiling the relationships in the application. If the application and the knowledge table are implemented by using a parallel programming language such as GHC [13], the mechanism for the knowledge table can be used effectively, since the compilation process can be accomplished as a background task.

6. Conclusions

In this paper, we have proposed a mechanism for a knowledge table, and related procedures. The knowledge table has been designed to manage effectively many relationships between items of knowledge in a knowledge base. Managing the relationships is useful for belief revision, searching for objects from super classes in isa hierarchies, and so on. To achieve this, we have introduced a new representation for a knowledge table, which adopts some of the advantages of table representation and list representation for the relationships, and uses logical bitwise computations to search for them. The computations greatly speed up the search for the relationships. The table-like representation has the following advantages: (1) it is easy to check adjacency and reachability among knowledge items, (2) it is easy to find loop structures among relationships, and (3) the space needed for keeping the relationships is less than in a representation using complete tables. These advantages result from combining table representation and list representation. Research is in progress to realize the following mechanisms [10, 11] as applications of knowledge tables: (1) keeping data dependencies for belief revision, (2) keeping information to control message-passing between objects in object-oriented programming, (3) keeping hierarchical taxonomies to control inheritances in frame-oriented programming, and (4) keeping information to generate explanations for results in rule-oriented programming.

Acknowledgment

The author would like to acknowledge the continuing guidance and encouragement of Dr. Mitsuhiko Toda, IIAS-SIS. He is also deeply grateful to Dr. Kozo Sugiyama, IIAS-SIS, for reading the draft of this paper and making many valuable comments.

References

1. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974), 55-60, 172-223.
2. DOYLE, J. The Ins and Outs of Reason Maintenance, IJCAI-83, (1983), 349-351.
3. ETHERINGTON, D. B. and REITER, R. On Inheritance Hierarchies with exceptions, *AAAI* (1983), 104-108.
4. KOWALSKI, R. Logic for Problem Solving, Elsevier North Holland, 1977.
5. MCDERMOTT, D. and DOYLE, J. Non-monotonic Logic I, MIT, Technical Report Memo 486, 1978.
6. MCDERMOTT, D. Contexts and Data Dependencies: A Synthesis, *IEEE Trans. PAMI*, PAMI-5, No. 3 (1983), 237-246.
7. PEREIRA, F. (Ed.), C-prolog User's Manual, Version 1.4a (Sept. 1983).
8. REITER, R. A Logic for Default Reasoning, Artificial Intelligence 13 (1980), 81-131.
9. RICH, R. Artificial Intelligence, *McGraw-Hill*, 1983.
10. SHINTANI, T., KATAYAMA, Y., HIRAISHI, K. and TODA, M. KORE: A Hybrid Knowledge Programming Environment for Decision Support Based on a Logic Programming Language, Proceedings of LPC '86 (Springer, Lecture Notes in Computer Science 264) (1986), 22-33.
11. SHINTANI, T. An Approach to Nonmonotonic Inference Mechanism in Production System KORE/IE, Proceedings of LPC '88 (Springer, Lecture Notes in Computer Science 384) (1989), 38-52.
12. STALLMAN, R. and SUSSMAN, G. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer Aided Circuit Analysis, Artificial Intelligence, 9, 2 (1977), 135-196.
13. UEDA, K. Guarded Horn Clauses, ICOT Technical Report TR-103, 1985.
14. HAYES-ROTH, F. The Knowledge-Based Expert System, A Tutorial, COMPUTER (September 1984), 11-28.

(Received April 16, 1987; revised December 4, 1989)

Appendix A

Procedure A4

```
set_rtype_at_T_position(A_r_list1, Position,
                          Relation_type, A_r_list2):-
    element_position(Position, List_Element, Element),
                                          %Procedure A0
    pick_up_list_element(A_r_list1, List_Element,
                          Object),
    Bit1 is 2*Element,
    Bit2 is Bit1 - 1,
    (Relation_type =:= 0,
     reset_bit(Object, Bit1, Object1),  %See Section 2.2
     reset_bit(Object, Bit2, Object2)
     ;
     Relation type =:= 1,
     reset_bit(Object, Bit1, Object1),
     set_bit(Object, Bit2, Object2)      %See Section 2.2
     ;
     Relation type =:= 2,
     set_bit(Object, Bit1, Object1),
     set_bit(Object, Bit2, Object2)),
    replace_list_element(A_r_list1,
           List_element_number, Object2, A_r_list2).
```

Procedure A0

```
element_position(N, M, L):-
      (N =< 14, M = 1, L = N
       ;
       X is N//14, Y is N mod 14,
       (Y =:= O, M = X, L = 14
        ;
        M is X + 1, L = Y)).
```

Procedure A5

```
find_sub_knowledge(Node, Knowledge_table, Subs):-
   find_position(Node, Knowledge_table,
                                   Node_position),
   take_LK_list(Knowledge_table, LK list),
   (LK list = [ ], Subs = [ ]
    ;
    find_sub_knowledge2(Node, Node_position,
                                   LK_list, Subs)).

find_sub_knowledge2(Node, Node_position, [ ], [ ]).
find_sub_knowledge2(Node, Node_position, [E1|E2],
                                         Subs):-
   (exist_relationship(E1, Node_position, 0)
                                   %Procedure A5.1
    find_sub_knowledge2(NOde, Node_position, E2,
                                         Subs)
    ;
    find_sub_knowledge2(Node, Node_position, E2,
                                        Subs2),
    Subs = [E1|Subs2]).
```

Procedure A5.1

```
exist_relationship(A_r_list, Position, Relation_type):-
   element_position(Position, List_Element, Element),
                                     %Procedure A0
   pick_up_list_element(A_r_list, List_Element,
                                         Object),
   Relation_type is (Object >> 2*(Element - 1))∧3.
```

Procedure A6

```
find_super_knowledge(Node, Knowledge_table,
                           Relation_type, Supers):-
   take_A_r_list(Node, Knowledge_table, A_r_list),
   take_LK_list(Knowledge_table, LK_list),
   find_relationship(A_r_list, LK_list, Relation_type,
                  Supers).          %Procedure A6.1
```

Procedure A6.1

```
find_relationship([ ], LK list, Relation_type, [ ]).
find_relationship([L1|L2], LK list, Relation_type,
                                        [K1|K2]):-
   pick_list_elements(LK_list, 14, LK_elements, Rest),
                                   %Procedure A6.2
   find_LK_elements(L1, LK_elements, Relation_type,
                  K1, 1),          %Procedure A6.3
```

```
  find_relationship(L2, Rest, Relation_type, K2, 1).
```

Procedure A6.2

```
pick_list_elements(List, 0, [ ], List).
pick_list_elements(List, Num, List, [ ]):-
  length(List, Length),
  Num > Length.
pick_list_elements([L1|L2], Num, [L1|Result], Rest):-
  Num2 is Num - 1,
  pick_list_elements(L2, Num2, Result, Rest).
```

Procedure A6.3

```
find_LK_elements(List_element, LK_elements,
  Type, [ ],                                  15).
find_LK_elements(List_element, [ ], Type, [ ], Num).
find_LK_elements(List_element, [LK1|LK2], Type,
                                       Result, Num):-
  Num2 is Num + 1,
  (Type is (List element » 2*(Num - 1))∧3,
   find_LK_elements(List_element, LK2, Type,
                                     Result2, Num2),
   Result = [LK1|Result2]
   ;
   find_LK_elements(List_element, LK2, Type,
                                     Result, Num2)).
```

Appendix B

The program P

```
add_to(Node, [ ], KT).
add_to(Node, [P1|Ps], KT):-
  write(Node), nl,
  TERM =. .[KT, Node, P1],
  assert(TERM),
  add_to(Node, Ps, KT).
parents(X, Ps, KT):-
  TERM =. .[KT, X, P],
  bagof(P, TERM, Ps).
children(X, Cs, KT):-
  TERM =. .[KT, C, X],
  bagof(C, TERM, Cs).
supers(X, S, KT):-
  (parents(X, Ps, KT),
   supers2(Ps, S2, KT),
   append(Ps, S2, S)
   ;
   S = [ ]).
supers2([ ], [ ], KT).
supers2([P1|Ps], A, KT):-
  supers(P1, A1, KT),
  supers2(Ps, As, KT),
  append(A1, As, A).
subs(X, S, KT):-
  (children(X, Ps, KT),
   subs2(Ps, S2, KT),
   append(Ps, S2, S)
   ;
   S = [ ]).
subs2([ ], [ ], KT).
subs2([P1|Ps], A, KT):-
  subs(P1, A1, KT),
  subs2(Ps, As, KT),
  append(A1, As, A).
append([ ], X, X).
append([X|Y], Z, [X|R]):-
  append(Y, Z, R).
```