Efficient Implementation of Non-Standard Connectives and Quantifiers in Deductive Reasoning Systems

Joongmin Choi and Stuart C. Shapiro
Department of Computer Science
State University of New York at Buffalo
Buffalo, New York 14260

Abstract

A rule use information (RUI) structure has been proposed to implement non-standard connectives and quantifiers that generalize conjunction, disjunction, implication, universal quantifier, and existential quantifier. The current implementation of the RUI structure in the SNePS Inference Package (SNIP) maintains a single RUI set for each rule, which causes a combinatorial number of substitution compatibility checks between RUIs as the number of antecedents in a rule and their instances increase.

In this paper, the problem of efficient RUI handling for deduction rules with non-standard connectives and quantifiers is addressed. Two kinds of algorithms are proposed, and methods of distributing RUIs over several sets are discussed. Complexity analyses and test results show that these schemes of RUI set distribution keep the complexity of RUI set handling to polynomial in terms of the number of RUIs and the number of substitution compatibility tests.

1 Introduction

Non-standard connectives and quantifiers are generalizations of the common connectives and quantifiers such as conjunction, disjunction, negation, implication, universal quantifier, and existential quantifier [16]. They are designed to provide closeness to human reasoning, structural simplicity, and expressibility in the areas of natural language understanding, knowledge representation, and reasoning [9, 14, 15]. Some of the connectives and quantifiers are briefly introduced below.

AND-OR is a single, parameterized connective which generalizes logical-and, logical-or, exclusive-or, nor, nand etc., and is symbolized as ${}_n\mathbb{X}_i^j$. The formula ${}_n\mathbb{X}_i^j(A_1, A_2, \ldots, A_n)$ is true when at least i and at most j of the arguments are true. Two rules of inference for the **AND-OR** connective are, (1) if it is known that exactly j of the arguments are true, then the remaining $n-j$ arguments must be false, and (2) if it is known that exactly $n-i$ of the arguments are false, then the remaining i arguments must be true. For instance, the statement "there are four men, Peter, John, Jason, and Bob, and exactly two of them are teachers" can be represented by ${}_4\mathbb{X}_2^2$ (*teacher*(*peter*), *teacher*(*john*), *teacher*(*jason*), *teacher*(*bob*)). If it is known that Jason and Bob are not teachers, we can conclude that Peter and John are teachers.

THRESH represented by ${}_n\Theta_i^j$ is the dual of **AND-OR**, and the formula ${}_n\Theta_i^j(A_1, A_2, \ldots, A_n)$ is true when either fewer than i or more than j of the arguments are true. If j is omitted, j is automatically set to $n-1$. **THRESH** is mainly used to represent the equivalence relation by ${}_n\Theta_1(A_1, A_2, \ldots, A_n)$ which indicates that the arguments are either all true or all false.

Implication ($\rightarrow$) is generalized by **or-entailment** ($\vee\!\rightarrow$) and **and-entailment** ($\&\!\rightarrow$). The formula $(A_1, A_2, \ldots, A_n) \vee\!\rightarrow (C_1, C_2, \ldots, C_m)$ means that the disjunction of the antecedents implies the conjunction of the consequents, and the formula $(A_1, A_2, \ldots, A_n) \&\!\rightarrow (C_1, C_2, \ldots, C_m)$ means that the conjunction of the antecedents implies the conjunction of the consequents.

Finally, **numerical quantifiers** represented by ${}_n\exists_i^j$ are defined to generalize the universal and existential quantifiers. The formula ${}_n\exists_i^j\bar{x}(P_1(\bar{x}), \ldots, P_k(\bar{x}) : Q(\bar{x}))$, where $\bar{x}$ is a sequence of variables, means that there are n sequences of constants each of which when substituted for $\bar{x}$ will make $P_1(\bar{x})$ & ... & $P_k(\bar{x})$ true, and at least i and at most j of these will also satisfy $Q(\bar{x})$. By **numerical quantifiers**, we can represent a sentence like "everyone has exactly one mother" by $\forall x\ [person(x) \rightarrow \exists_1^1 y\ (person(y) : mother(y, x))]$. This kind of rule is especially used for reasoning by exclusion, i.e. if everyone has exactly one mother, then Mary can't be John's mother if it is already known that Jane is John's mother.

Non-standard connectives and quantifiers have different rules of inference that determine how many antecedents must be consistently instantiated (by positive or negative instances) to derive the consequents. For instance, **and-entailment** requires all the antecedents be instantiated to deduce the consequents, but only one antecedent instantiation is enough for **or-entailment** to deduce the consequents. Also, the **AND-OR** connective ${}_n\mathbb{X}_i^j$ has many different rules of inference according to the values of the parameters n, i, and j.

This paper presents uniform and efficient methods of implementing non-standard connectives and quantifiers. Various kinds of rules of inference are uniformly manipulated by *rule use information* (RUI) set struc-

0073-1129-1/92 $3.00 © 1992 IEEE

tures that keep the instance information including variable substitutions and the number of antecedents that are consistently instantiated by positive or negative instances. Section 2 describes the RUI set structure with its advantages and problems, and discusses how the RUI set can uniformly handle different rules of inference for non-standard connectives. For efficient handling of the RUI set structure, we present two algorithms that distribute RUIs over several places. Section 3 describes the **P-tree** algorithm, and Section 4 describes the **S-indexing** algorithm. Test results are shown in Section 5, and related works are compared in Section 6.

2 Rule Use Information (RUI)

The RUI structure has been proposed and implemented in the SNePS Inference Package (SNIP) [8, 10, 13] to save instance information of the antecedents of a rule including variable substitutions and the number of positive and negative instantiations, and also to combine those instances that have consistent bindings for shared variables. A rule is associated with a set of RUIs called a "RUI set" that traces the history of instance handling. The current implementation of SNIP maintains a RUI set for each rule.

A RUI consists of 4 elements as below,

```
<RUI> ::= (<sbst> <pos count> <neg count>
           <flagged node set>)
```

where `<sbst>` represents variable substitutions from instances of antecedents, `<pos count>` denotes the number of antecedents known to be true, and `<neg count>` denotes the number of antecedents known to be false. The nodes of the `<flagged node set>` are the antecedents of the rule and a flag associated with each node indicates whether that node is known to be true or false.

As an example, consider a knowledge based system for reasoning about kinship facts. Such a system might need to have an `and-entailment` deduction rule such as Rule_1 for recognizing the husband relationship.

Rule_1: ∀x,y [`man`(x), `woman`(y), `married`(x,y) &→ `husband`(x,y)]

Suppose we want to derive all of the husband relationships from the following set of facts through backward chaining.

`man`(john) `man`(fred) `man`(bob) `man`(steve)
`woman`(mary) `woman`(jane) `woman`(deb)
`woman`(ada) `woman`(sue)
`married`(steve, sue)

The pattern `man`(x) has 4 instances, `woman`(y) has 5 instances, and `married`(x,y) has 1 instance. The RUI set of Rule_1 is initially empty, and is dynamically augmented as new instances of antecedents are processed. A resulting RUI set of Rule_1 after processing all the above instances is

```
    (
r1  ( {john/x} 1 0  {P1:true} )
r2  ( {fred/x} 1 0  {P1:true} )
r3  ( {bob/x} 1 0  {P1:true} )
r4  ( {steve/x} 1 0  {P1:true} )
r5  ( {mary/y} 1 0  {P2:true} )
r6  ( {john/x, mary/y} 2 0  {P1:true, P2:true} )
r7  ( {fred/x, mary/y} 2 0  {P1:true, P2:true} )
r8  ( {bob/x, mary/y} 2 0  {P1:true, P2:true} )
r9  ( {steve/x, mary/y} 2 0  {P1:true, P2:true} )
r10 ( {jane/y} 1 0  {P2:true} )
r11 ( {john/x, jane/y} 2 0  {P1:true, P2:true} )
r12 ( {fred/x, jane/y} 2 0  {P1:true, P2:true} )
r13 ( {bob/x, jane/y} 2 0  {P1:true, P2:true} )
r14 ( {steve/x, jane/y} 2 0  {P1:true, P2:true} )
r15 ( {deb/y} 1 0  {P2:true} )
r16 ( {john/x, deb/y} 2 0  {P1:true, P2:true} )
r17 ( {fred/x, deb/y} 2 0  {P1:true, P2:true} )
r18 ( {bob/x, deb/y}  2 0 {P1:true, P2:true} )
r19 ( {steve/x, deb/y} 2 0  {P1:true, P2:true} )
r20 ( {ada/y} 1 0  {P2:true} )
r21 ( {john/x, ada/y} 2 0  {P1:true, P2:true} )
r22 ( {fred/x, ada/y} 2 0  {P1:true, P2:true} )
r23 ( {bob/x, ada/y} 2 0 {P1:true, P2:true} )
r24 ( {steve/x, ada/y} 2 0  {P1:true, P2:true} )
r25 ( {sue/y} 1 0  {P2:true} )
r26 ( {john/x, sue/y} 2 0  {P1:true, P2:true} )
r27 ( {fred/x, sue/y} 2 0  {P1:true, P2:true} )
r28 ( {bob/x, sue/y} 2 0 {P1:true, P2:true} )
r29 ( {steve/x, sue/y} 2 0  {P1:true, P2:true} )
r30 ( {steve/x, sue/y} 1 0 {P3:true} )
r31 ( {steve/x, sue/y} 2 0 {P1:true, P3:true} )
r32 ( {steve/x, sue/y} 2 0 {P2:true, P3:true} )
r33 ( {steve/x, sue/y} 3 0 {P1:true,P2:true,P3:true})
    )
```

Here, `P1`, `P2`, and `P3` represent `man`(x), `woman`(y), and `married`(x,y), respectively. Note that the same RUI set, except for the order of RUIs, will result no matter which instance is processed first.

Resolving binding conflicts of shared variables is done by checking substitution compatibility between RUIs. Two RUIs are said to be "compatible" when the substitutions of the two RUIs are consistent, which means that a common variable in both substitutions is bound to the same value. For example, a RUI `r4` is compatible with `r30`, but not compatible with `r1`. Any two RUIs in a RUI set that have compatible substitutions and have disjoint `<flagged node set>`s are combined to create a merged RUI. When two RUIs r_a and r_b produce a merged RUI r_m, the value of `<pos count>` of r_m can be obtained by the summation of the values of `<pos count>`s of r_a and r_b. The value of `<neg count>` is similarly calculated. Also the value of `<sbst>` of r_m will be a union of the values of `<sbst>`s of r_a and r_b. The value of `<flagged node set>` is similarly obtained. For instance, `r1`, and `r5` are combined to produce `r6`, and `r29` and `r30` are merged to get `r33`. Eventually, `husband`(steve,sue) is derived from `r33` whose `<pos count>` value is the same as the number of antecedents of the rule.

2.1 Advantages of the RUI set

The advantages of employing the RUI set structure in deductive problem solving can be described in 4 ways.

First, non-standard rules of inference can be uniformly implemented. Uniformity in manipulating

various different kinds of rules of inference can be achieved by exploiting **<pos count>**, **<neg count>**, and **<flagged node set>** fields of the RUI structure. For example, an **and-entailment** rule can deduce the consequents if there is a RUI whose **<pos count>** value equals the number of antecedents in the rule. Also, two rules of inference for the **AND-OR** rule ${}_{n}\bigvee\!\!\bigwedge_{i}^{j}(A_1, A_2, \ldots, A_n)$ can be stated as, (1) if there is a RUI whose **<pos count>** value is equal to j, those arguments that are not in its **<flagged node set>** field are proved to be false, and (2) if there is a RUI whose **<neg count>** value is equal to $n - i$, those arguments not in its **<flagged node set>** fields are proved to be true. In the same fashion, expressions like "exactly two of n arguments", "not all of n arguments", or "at least 3 of n arguments" can also be expressed easily.

Second, we can reuse previous traces of rule handling steps saved in the RUI set for subsequent deductions on the same rule. Duplicate rule activation steps, including pattern matchings and binding conflict resolutions, are avoided. For example, suppose a new fact **married**(john,mary) is added to the knowledge base to find more husband relationships. All we have to do now is to create a new RUI **r34** for the instance **married**(john,mary), and check substitution compatibility between **r34** and each of RUIs in the **RUI** set of Rule_1 to produce the following merged RUIs.

```
r34 ({john/x, mary/y} 1 0 {P3:true})
r35 ({john/x, mary/y} 2 0 {P1:true,P3:true})
r36 ({john/x, mary/y} 2 0 {P2:true,P3:true})
r37 ({john/x, mary/y} 3 0 {P1:true,P2:true,P3:true})
```

r34 is combined with **r1**, **r5**, and **r6** to generate **r35**, **r36**, and **r37**, respectively. **r37** contributes to the derivation of **husband**(john,mary). This reusability enables the system to learn from experience [1, 2].

Third, the RUI set structure facilitates a full-degree of AND-parallelism. A major issue in the AND-parallel computation of a conjunctive rule is the overhead of resolving binding conflicts of shared variables. By maintaining a separate process for a rule that maintains a RUI set structure, processes corresponding to its antecedent patterns just save instances that are pattern matched and send them to the rule process without worrying about checking compatibilities of shared variables which is later performed at the rule process. There is no need to order antecedent patterns according to variable specifications [4], and no need to have complex communication mechanisms among antecedents.

Fourth, the RUI set structure is also used to facilitate the implementation of bi-directional inference which combines backward and forward chaining [17], and the implementation of recursive rule inference that does not cause infinite loops [10].

2.2 Efficient RUI Set Handling

One problem of maintaining the RUI set is efficiency, which resulted from the single RUI set management that keeps one sequential RUI set for each rule. In general, reasoning by maintaining a single RUI set for each rule is intractable due to the combinatorial number of substitution compatibility checks between RUIs. The number of RUIs in a set also becomes large as the application problem size increases. Furthermore, many unnecessary RUIs are merged between patterns that shares no common variables, such as **man**(x) and **woman**(y), because the compatibility checks between RUIs of these patterns always succeed. It is recognized that updating a single RUI set has an exponential complexity, in average, in terms of the number of antecedents in a rule. (Details are in Section 3).

In this paper, techniques for distributing RUIs are presented for fast reasoning with non-standard connectives and quantifiers. Two algorithms are given for efficient RUI handling in this distributed RUI set scheme.

The first algorithm, using a **P-tree**, is designed for conjunctive deductions rules. A conjunctive deduction rule has conjunctions of two or more clauses in the rule premise which should be satisfied simultaneously with consistent variable bindings for shared variables to derive the consequents. Rules with **and-entailment** and **numerical quantifiers** belong to this category.

The second algorithm is designed for those connectives in which not all the antecedents are required to be instantiated to derive the consequents. For these non-conjunctive connectives, we present a scheme of **S-indexing** (S stands for 'substitution') in which RUIs are distributed by bound values for free variables. This scheme is applied to **AND-OR**, **THRESH**, and **or-entailment** connectives.

3 P-tree Algorithm

This section presents a technique for efficient RUI handling for conjunctive deduction rules expressed by **and-entailment** or **numerical quantifiers**. A binary pattern tree called a **P-tree** is compiled from a set of antecedents of a rule and RUIs are distributed over the nodes of the **P-tree**. A **P-tree** of a rule is defined as a binary tree in which, (1) a leaf node corresponds to an antecedent pattern, (2) a parent node is a conjunction of its children, and (3) the root node represents the whole conjunctions of the rule premise.

A **P-tree** is compiled by considering variable specification of each pattern in order to arrange those patterns with common variables to be adjacent in the tree. The algorithm takes a list of pattern-variable specifications as its input and produces a tree of patterns as its output.

As an example, consider an **and-entailment** rule

Rule_2: ∀v1,v2,v3,v4,v5 [A(v1,v2), B(v3,v4), C(v3,v5), D(v1,v3), E(v2,v5), F(v2,v3), G(v1,v4) &→ H(v1,v2,v3,v4,v5)]

The pattern-variable specification of a pattern P(x_1, ..., x_n) is defined as a list (P $x_1 \cdots x_n$). So, the pattern-variable list of the above rule will be

((A v1 v2) (B v3 v4) (C v3 v5) (D v1 v3) (E v2 v5) (F v2 v3) (G v1 v4))

The **P-tree** compilation algorithm is divided into

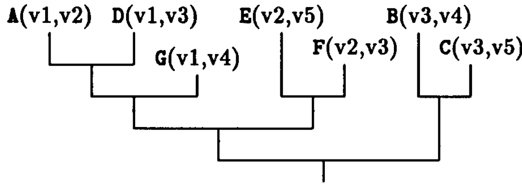


Figure 1: A **P-tree** for $Rule_2$

3 procedures. The first procedure `PatVar-to-VarPat` converts a pattern-variable list into a variable-pattern list. A variable-pattern specification has the form of (v $P_1 \cdots P_m$), where v is a variable and each P_i is an antecedent pattern that contains v. `PatVar-to-VarPat` generates the following variable-pattern list.

((v1 A D G) (v2 A E F) (v3 B C D F) (v4 B G) (v5 C E))

The second procedure `VarPat-to-PatSeq` builds a linear sequence of patterns from a variable-pattern list. This procedure arranges those patterns that have common variables to be close in the sequence. From the above variable-pattern list, `VarPat-to-PatSeq` works as follows. Initially, a pattern sequence is set to (A D G) since the first variable-pattern specification is (v1 A D G), and v1 is marked as processed. The union of variables of these patterns is (v1 v2 v3 v4). Now v2 is the first unprocessed variable, so (A E F) is the next candidate to be included in the sequence, but only E and F are inserted since A is already in the sequence. The resulting sequence is (A D G E F) with (v1 v2 v3 v4 v5) as the union of variables. Now v3 is the next unprocessed variable, so (B C D F) is the candidate to be included in the sequence, and B and C are inserted to make the final pattern sequence (A D G E F B C). Note that the order of the sequence is important.

The third procedure `PatSeq-to-PTree` builds a **P-tree** from a pattern sequence. A main step of this procedure is to extract the first two patterns from the sequence to make them adjacent in the tree if they share a common variable. Otherwise the first pattern is combined with the last pattern in the tree built so far. A **P-tree** for the above pattern sequence will be ((((A D) G) (E F)) (B C))). Figure 1 depicts this tree graphically.

The **P-tree** algorithm has reasonable complexity. Suppose p is the number of antecedent patterns in a rule and v is the average number of variables in a pattern. Then, `PatVar-to-VarPat` has the complexity of $O(p \cdot v)$, `VarPat-to-PatSeq` has $O(v)$, and `PatSeq-to-PTree` has $O(p)$. Hence, the overall complexity of the **P-tree** algorithm is $O(p \cdot v)$.

Distribution of RUIs is enabled by assigning a RUI set to each node in the **P-tree**. A RUI set of a leaf node is a set of RUIs directly built from the instances of the corresponding antecedent pattern. A RUI set of a non-leaf node is a set of RUIs successfully combined

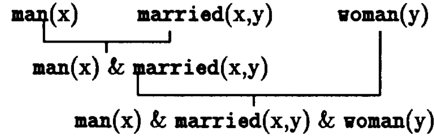


Figure 2: A **P-tree** for $Rule_1$

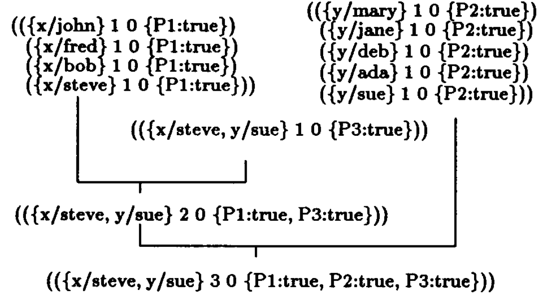


Figure 3: Distribution of RUIs over the **P-tree** nodes

from RUIs of its children nodes through substitution compatibility tests for shared variables. RUIs in the RUI set of the root node contribute to the derivation of the consequents when the value of **<pos count>** is equal to the number of antecedents.

As an example, a **P-tree** for the rule $Rule_1$ is drawn in Figure 2, and the distribution of RUIs over the nodes of this **P-tree** is shown in Figure 3. By distributing RUIs over **P-tree** nodes, the overall number of compatibility tests is drastically decreased, not to mention the total number of RUIs in the rule. As shown in Section 2, 33 RUIs and 49 compatibility tests are made by the single RUI set method, whereas the distributed RUI set method using **P-tree** needs only 12 RUIs and 9 compatibility tests. The differences of these measures will be significant as the problem size increases.

In Table 1, the complexity of handling the RUI set for conjunctive rules is compared between the single RUI set scheme and the distributed RUI set scheme. Two metrics in these performance measures are; (1) the number of total RUIs in the rule, and (2) the number of compatibility tests between RUIs during the inference. We assume that there are, in average, n antecedents in a rule and each antecedent has m instances. Notice that the average complexity is reduced from exponential to polynomial in terms of n.

Two application problems that have conjunctive rules with a large number of antecedents are shown here. Knowledge bases are suggested, and **P-trees** for conjunctive rules are compiled. Execution time comparisons are given in Section 5.

Map coloring problem: Coloring a given planar

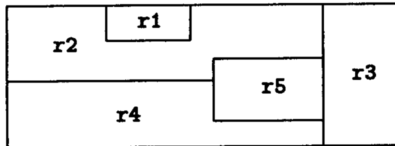
	The single RUI set scheme	
Cases	number of RUIs	number of tests
Best	$O(m \cdot n)$	$O(m^2 \cdot n^2)$
Worst	$O(m^n)$	$O(m^n)$
Average	$O(m \cdot 2^n)$	$O(m^2 \cdot 2^n)$

	The distributed RUI set scheme	
Cases	number of RUIs	number of tests
Best	$O(m \cdot n)$	$O(m^2 \cdot n)$
Worst	$O(m^n)$	$O(m^n)$
Average	$O(m \cdot n)$	$O(m^2 \cdot n)$

n: avg. no. of antecedents in a rule
m: avg. no. of instances for each antecedent

Table 1: Performance comparison of RUI handling for conjunctive rules

map with four colors (e.g. red, blue, green, and yellow) so that adjacent regions have different colors. Suppose we have the following planar map with 5 regions.



A way of expressing this map coloring problem is specifying a rule such that

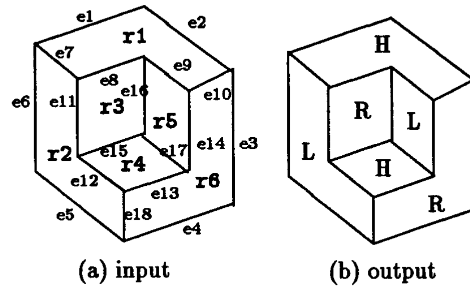
∀ r1,r2,r3,r4,r5
[**next**(r1,r2), **next**(r2,r3), **next**(r2,r4),
next(r2,r5), **next**(r3,r4),
next(r3,r5), **next**(r4,r5)
&⟶ **colormap**(r1,r2,r3,r4,r5)]

where given facts are

next(red,blue) **next**(red,green)
next(red,yellow) **next**(blue,red)
next(blue,green) **next**(blue,yellow)
next(green,red) **next**(green,blue)
next(green,yellow) **next**(yellow,red)
next(yellow,blue) **next**(yellow,green)

There are 7 patterns in a rule, and each pattern is matched with all 12 facts. Hence, it is easy to expect that the number of RUIs and the number of substitution compatibility tests to find all 72 possible colorings from this knowledge base will be exponential by a naive method, even though the size of the knowledge base is quite small.

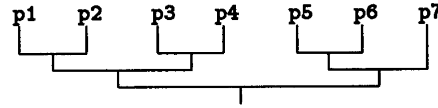
A **P-tree** for the rule is shown below. (Here, p1 corresponds to the first antecedent pattern of the rule,



(a) input (b) output

Figure 4: A sample scene

i.e. **next**(r1,r2), and **p2** corresponds to the second, i.e. **next**(r2,r3), and so on).



<u>Scene analysis problem</u>: Given a scene described in terms of its edges that can be positive, negative, or vertical according to their slopes, the task is to produce an interpretation of the scene in terms of face descriptions that may be left, right, or horizontal. An example input scene is given in Figure 4(a) (this scene is an abbreviated version of the scene in [3], page 67), and an interpretation is given in Figure 4(b).

A rule for defining this scene is

∀ r1,r2,r3,r4,r5,r6
[**pedge**(bg,r1), **nedge**(r1,bg), **vedge**(r6,bg),
pedge(r6,bg), **nedge**(bg,r2), **vedge**(bg,r2),
nedge(r2,r1), **pedge**(r1,r3), **nedge**(r5,r1),
pedge(r1,r6), **vedge**(r2,r3), **nedge**(r2,r4),
pedge(r4,r6), **vedge**(r5,r6), **pedge**(r3,r4),
vedge(r3,r5), **nedge**(r4,r5), **vedge**(r2,r6)
&⟶ **interpret**(r1,r2,r3,r4,r5,r6)]

Here, 'bg' stands for background. Each antecedent pattern corresponds to an edge of the scene. For example, the edge **e1** has positive slope and separates a face **r1** from the background, so the first antecedent is represent by **pedge**(bg,r1). Also **e3** is a vertical edge that separates **r6** from the background, so the third antecedent is represented by **vedge**(r6,bg). Other edge descriptions are similarly defined.

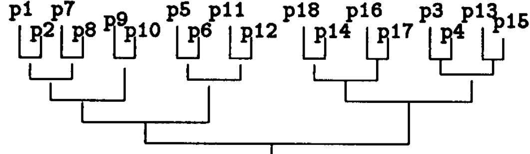
A set of interpretation rules characterizing the different kinds of edges are

vedge(L,R) **vedge**(R,L)
vedge(R,bg) **vedge**(bg,L)
pedge(H,R) **pedge**(R,H)
pedge(bg,H) **pedge**(R,bg)

nedge(L,H) **nedge(H,L)**
nedge(H,bg) **nedge(bg,L)**

Here, L, R, and H stand for left, right, and horizontal, respectively. Edges are characterized in terms of convexity or concavity of the adjoining regions. For example, **e16** becomes **vedge** since its two adjoining faces are R and L. From this knowledge base, the answer **interpret**(H, L, R, H, L, R) can be obtained, as shown in Figure 4(b).

A **P-tree** for the rule is shown below. (Here again, **p1** corresponds to the first antecedent pattern of the rule, i.e. **pedge**(bg,r1), and **p2** corresponds to the second, i.e. **nedge**(r1,bg), and so on).



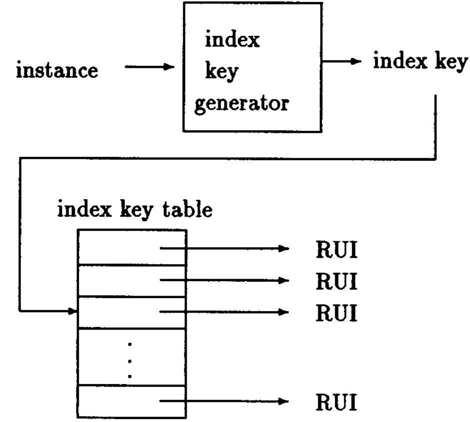


Figure 5: Overview of **S-indexing** mechanism

4 S-Indexing Algorithm

This section describes a method of efficient RUI set handling for non-conjunctive connectives including **AND-OR**, **THRESH**, and **or-entailment**. Non-conjunctive connectives are characterized by the fact that the number of antecedents that should be instantiated to derive the consequents is not always the same as the number of antecedents, but rather it varies depending on the connective's rules of inference. The **P-tree** method is not applicable to these non-conjunctive connectives, because the root node of a **P-tree**, which solely contributes to the derivation of the consequents, can only be reached by conjunctively instantiating all of the antecedents.

For non-conjunctive connectives, we now propose a scheme of **S-indexing** that distributes RUIs by bound values of variables in a rule. An index key is generated from the variable substitution of an instance in such a way that an instance with a variable substitution $\{b_1/v_1, b_2/v_2, \ldots, b_m/v_m\}$ produces an index key of $\langle b_1, b_2, \ldots, b_m \rangle$. Each rule manages an index key table so that a RUI can be accessed and modified by a corresponding index key from the table. Two instances with an identical variable substitution will have the same index key, and eventually will access the same RUI. When a new instance is processed, the system retrieves the corresponding RUI via the index key, changes the values of **<pos count>**, **<neg count>**, and **<flagged node set>** properly, and replaces the old RUI by the changed one. A benefit from this idea of indexing is that there is no need for checking binding conflicts between RUIs since different bindings lead to different RUIs, and consequently it is sufficient to maintain a single RUI for each different variable substitution. A schematic of the **S-indexing** mechanism is given in Figure 5.

As an example, consider an **AND-OR** rule

$${}_{4}\bigvee\!\!\!\!\bigwedge{}_{2}^{3}\ \{\ P_1(x,y),\ P_2(x,y),\ P_3(x,y),\ P_4(x,y)\}$$

and some facts

$$P_1(a,b)\quad P_2(a,b)\quad \neg P_3(b,c)\quad P_4(b,c)$$
$$\neg P_1(c,d)\quad \neg P_2(b,c)\quad \neg P_3(a,d)\quad P_4(a,d)$$

Here, we use $\neg$ as the negation symbol that is just an abbreviation of ${}_{1}\bigvee\!\!\!\!\bigwedge{}_{0}^{0}$. There are two variables in the **AND-OR** rule representation, so an index key will have the form of $\langle b_1, b_2 \rangle$, where b_1 is a bound value for x and b_2 is a bound value for y. When $P_1(a,b)$ is processed, a RUI $(\{a/x, b/y\}\ 1\ 0\ \{P1 : true\})$ is built and a new index key $\langle a,b \rangle$ is created in the table that will be used to access this RUI. The next instance $P_2(a,b)$ produces the same index key, so the previous RUI is retrieved and replaced by $(\{a/x, b/y\}\ 2\ 0\ \{P1 : true, P2 : true\})$. The third instance $\neg P_3(b,c)$ produces a different index key, resulting in the creation of a new key $\langle b,c \rangle$ and a corresponding RUI $\{\{b/x, c/y\}\ 0\ 1\ \{P3 : false\}\}$. Other instances are processed in the same way, and the result of RUI distribution after all instances are processed is shown below.

index keys	**RUIs**
$\langle a,b \rangle$	$(\{a/x, b/y\}\ 2\ 0\ \{P1 : true, P2 : true\})$
$\langle b,c \rangle$	$(\{b/x, c/y\}\ 1\ 2\ \{P2 : false, P3 : false, P4 : true\})$
$\langle c,d \rangle$	$(\{c/x, d/y\}\ 0\ 1\ \{P1 : false\})$
$\langle a,d \rangle$	$(\{a/x, d/y\}\ 1\ 1\ \{P3 : false, P4 : true\})$

Note that the rule of inference for **AND-OR** is applied to make $P_1(b,c)$ true for the index of $\langle b,c \rangle$, since the inference rule says at least 2 of 4 arguments should be true but 2 arguments are already known to be false (**<neg count>** is 2).

We now provide an application problem that uses non-conjunctive connectives (this problem is described in [20]).

Job puzzle problem:

"There are four people: Roberta, Thelma, Steve, and Pete. Among them, they hold eight different jobs. Each holds exactly two jobs. The jobs are: chef, guard, nurse, telephone operator, police officer (gender not implied), teacher, actor, and boxer. The job of nurse is held by a male. The husband of the chef is the telephone operator. Roberta is not a boxer. Pete has no education past the ninth grade. Roberta, the chef, and the police officer went golfing together. Now the question is : Who holds which jobs?"

There are some hidden information in this description of puzzle. For instance, Pete can't hold the job of teacher, police officer, or nurse since those jobs require high-level education. Also there are some general linguistic information such that an actor is recognized as a male by its suffix `-or`, and some common knowledge that Roberta and Thelma are female names and Steve and Pete are male names, and so on. A knowledge base for this puzzle are shown below including all extra information discussed above.

`person`(roberta) `person`(thelma)
`person`(steve) `person`(pete)
`female`(roberta) `female`(thelma)
`male`(steve) `male`(pete)
`job`(chef) `job`(guard)
`job`(nurse) `job`(operator)
`job`(police) `job`(teacher)
`job`(actor) `job`(boxer)

R_1: ∀p [`person`(p) $\vee\!\!\rightarrow$
${}_{8}\mathbb{X}_{2}^{2}$ (`hold`(p,chef), `hold`(p,guard), `hold`(p,nurse), `hold`(p,operator), `hold`(p,police), `hold`(p,teacher), `hold`(p,actor), `hold`(p,boxer))]
R_2: ∀j [`job`(j) $\vee\!\!\rightarrow$
${}_{4}\mathbb{X}_{1}^{1}$ (`hold`(roberta,j), `hold`(thelma,j), `hold`(steve,j), `hold`(pete,j))]
R_3: ∀p [`female`(p) $\vee\!\!\rightarrow$
${}_{3}\mathbb{X}_{0}^{0}$ (`hold`(p,nurse), `hold`(p,actor), `hold`(p,operator))]
R_4: ∀p [`male`(p) $\vee\!\!\rightarrow$ ${}_{1}\mathbb{X}_{0}^{0}$ (`hold`(p,chef))]
R_5: ${}_{1}\mathbb{X}_{0}^{0}$ (`hold`(roberta,boxer))
R_6: ${}_{3}\mathbb{X}_{0}^{0}$ (`hold`(pete,nurse), `hold`(pete,police), `hold`(pete,teacher))
R_7: ${}_{2}\mathbb{X}_{0}^{0}$ (`hold`(roberta,chef), `hold`(roberta,police))
R_8: ∀p [`person`(p) $\vee\!\!\rightarrow$
${}_{2}\mathbb{X}_{0}^{1}$ (`hold`(p,chef), `hold`(p,police))]

question: `hold`(x,y)?

Let us see how **S-indexing** distributes RUIs for this job puzzle problem. For brevity of explanation, we only consider the manipulation of the `AND-OR` rule of R_1

${}_{8}\mathbb{X}_{2}^{2}$ (`hold`(p,chef), `hold`(p,guard), `hold`(p,nurse), `hold`(p,operator), `hold`(p,police), `hold`(p,teacher), `hold`(p,actor), `hold`(p,boxer))

We name the first argument as P1, and the second argument as P2, and so on. Since there is one variable *p* in this rule, each index key will be a bound value for *p*. It is inferred from R_5 that Roberta is not a boxer, and this provides a negative instance information to `hold`(p,boxer) of the rule that is named P8, resulting in the creation of the following RUI with a new index key ⟨*roberta*⟩.

({roberta/p} 0 1 {P8:false})

Then, it is inferred from R_7 that Roberta is neither a chef nor a police officer. These instances have the same index key ⟨*roberta*⟩ as before, so the previous RUI is retrieved and replaced by

({roberta/p} 0 3 {P1:false, P5:false, P8:false})

Since Roberta is a female, it can also be inferred from R_3 that Roberta is neither a nurse, an actor, nor an operator. So the RUI for the index key ⟨*roberta*⟩ is replaced again by

({roberta/p} 0 6 {P1:false, P3:false, P4:false, P5:false, P7:false, P8:false})

At this point, the rule of inference for `AND-OR` is applied to derive that Roberta is both a guard and a teacher. Eventually, we will get the following answer at the end of the processing.

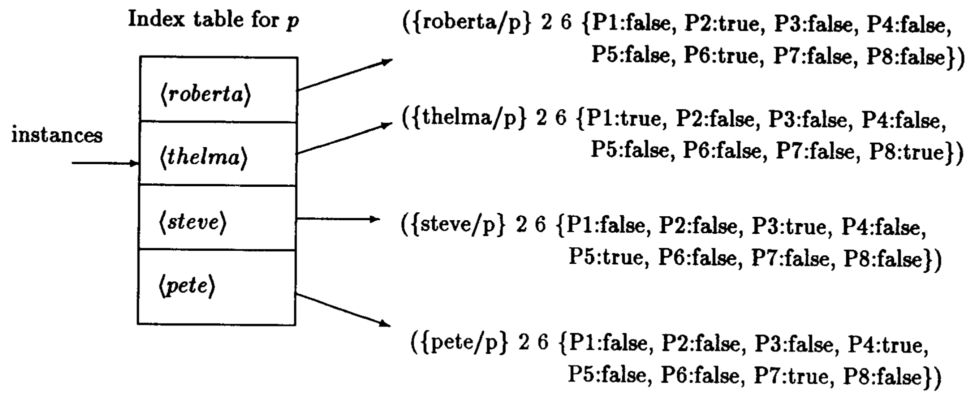
`hold`(roberta,teacher)
`hold`(roberta,guard)
`hold`(thelma,chef)
`hold`(thelma,boxer)
`hold`(steve,nurse)
`hold`(steve,police)
`hold`(pete,actor)
`hold`(pete,operator)

The distribution of RUIs is shown in Figure 6(a) for the `AND-OR` rule of R_1, and in Figure 6(b) for the `AND-OR` rule of R_2.

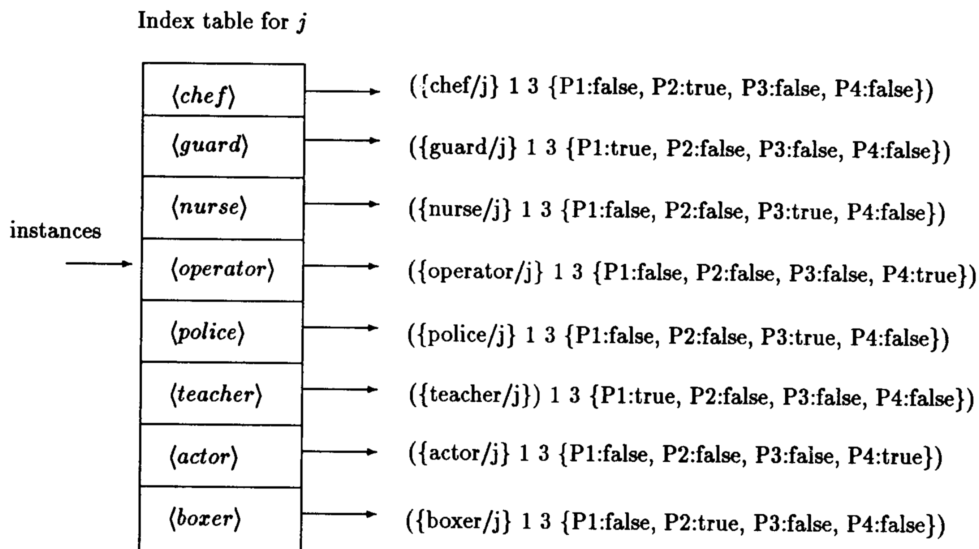
In Table 2, the complexity of handling the RUI set for non-conjunctive rules is compared between the single RUI set scheme and the distributed RUI set scheme. The same assumptions and measures mentioned in the previous section are used. You can notice in this table that the substitution compatibility tests are unnecessary in the **S-indexing** method.

5 Test Results

SNIP [8] was modified to employ the **P-tree** algorithm and the **S-indexing** scheme, implemented in COMMON-LISP. We ran the original and the modified versions of SNIP on the three examples described in the previous sections on a TI-Explorer lisp machine. (SNePS representations are not shown here). The map coloring and the scene analysis problems use the **P-tree** algorithm, and the **S-indexing** is applied to the job puzzle problem. Table 3 compares the test results between the single RUI set scheme and the distributed RUI set scheme in terms of execution time, the number of RUIs, and the number of compatibility checks for these three problems.



(a) **S-indexing** for the AND-OR rule of R_1



(b) **S-indexing** for the AND-OR rule of R_2

Figure 6: Distribution of RUIs by **S-indexing**

	The single RUI set scheme	
Cases	number of RUIs	number of tests
Best	$O(m \cdot n)$	$O(m^2 \cdot n^2)$
Worst	$O(m \cdot 2^n)$	$O(m^2 \cdot 2^n)$
Average	$O(m \cdot n^2)$	$O(m^2 \cdot 2^n)$

	The distributed RUI set scheme	
Cases	number of RUIs	number of tests
Best	$O(m)$	$O(1)$
Worst	$O(m \cdot n)$	$O(1)$
Average	$O(m \cdot n)$	$O(1)$

n: avg. no. of antecedents in a rule
m: avg. no. of instances for each antecedent

Table 2: Performance comparison of RUI handling for non-conjunctive rules

	The single RUI set scheme		
Problems	CPU time (in sec.)	number of RUIs	number of tests
Coloring	4563	18276	327212
Scene	9130	64372	454386
Puzzle	5 hrs	3608	189774

	The distributed RUI set scheme		
Problems	CPU time (in sec.)	number of RUIs	number of tests
Coloring	99	612	9936
Scene	17	84	103
Puzzle	146	88	0

Table 3: Test results comparison

Notice from the table that the **P-tree** method is much more effective for the scene analysis problem than for the map coloring problem. The main reason is that the number of antecedents of the scene analysis rule is much bigger than that of the map coloring rule. This phenomenon can be verified by the complexity analysis in Table 1 in which the number of antecedents is denoted by n, and the **P-tree** method reduces the average complexity of processing the RUI set from 2^n to n. From this observation, it can be said that the effect of the **P-tree** method will be severe especially for a rule with a large number of antecedents.

For the job puzzle problem, the single RUI set method was very inefficient, and it even didn't finish after more than 5 hours of running, as seen in the table. The **S-indexing** method significantly improves the performance by eliminating compatibility tests.

6 Related Work

The **P-tree** algorithm can be compared with the RETE algorithm [6] that was developed to reduce the overhead of pattern matching costs when many patterns and objects are in the system. Both methods focus on the control of rule execution by building a tree (or network) structure from antecedents of a rule (or productions), and then by feeding the instances (or working memory elements) into the structure to produce combined instance information that has consistent variable bindings. In production systems, this combined information consists of the 'conflict set'. Each node of the structure saves its instances as a way of avoiding duplicate processing, and also checking the compatibility of bindings is performed for shared variables between two adjacent nodes (in RETE, this process is called 'join').

However, there are some differences between the two approaches. The RETE algorithm is basically designed for production systems [5, 11] that employ only forward rule chaining. Therefore, in every cycle of the interpreter, all patterns in the production memory are compared to all elements in working memory to determine which productions can be triggered. In fact, this characteristic of ***all-patterns vs. all-objects*** match enables the system to compile a single RETE network from entire patterns. The RETE algorithm, however, is not applicable to backward chaining in which pattern matches are normally performed only for those patterns of a rule that is currently being triggered, and also only those facts that are pattern matched will send information to the requesting patterns. The **P-tree** algorithm is mainly applied to backward reasoning systems, although it can also be extended to handle forward chaining. There will be as many **P-trees** as the number of rules in the knowledge base, whereas the RETE algorithm will compile a single network for entire productions. A consequence from this fact is that a RETE network has to be re-compiled when a new production is added to the production memory, so no dynamic addition of productions is assumed in the RETE algorithm.

Studies on the control of backward inference mainly emphasize the selection of a rule among possibly many applicable rules at each branch in the inference tree. Smith [18] suggested an algorithm for finding optimal inference path in non-redundant knowledge base. Greiner's work [7] extends Smith's work using explanation-based learning [12] to find an optimal inference strategy for a redundant knowledge base. Greiner actually proves that finding an optimal path from a general redundant knowledge base is NP-complete, so his method makes a redundant knowledge base irredundant by removing some existing rules, and then applies Smith's algorithm. Also a method of choosing backward or forward directions during an inference is suggested in [19]. All of these approaches concentrate on the selection of a proper rule or choosing a direction for efficiency, but the issue of the overhead for executing the selected rule itself is largely ignored. We hope that more efficient infer-

ence engines can be built by combining the above rule selection algorithms with our rule activation methods.

Conclusion

We have presented methods for the efficient implementation of rules with non-standard connectives and quantifiers. Our main objective is to manage a uniform way of handling a variety of connectives, and at the same time to have a reasonably good performance for controlling rule activations with these connectives. A uniform manipulation of different connectives and quantifiers is achieved by keeping the RUI set structure for each rule. The RUI set structure provides a way of avoiding duplicate pattern matches and binding conflict resolutions of shared variables by saving the history of rule activations. For efficient handling of the RUI set, the **P-tree** and **S-indexing** algorithms have been designed and implemented. By using these algorithms, the complexity of processing the RUI set becomes polynomial in terms of the average number of antecedents in a rule and the average number of instances for each antecedent.

References

[1] Joongmin Choi and Stuart C. Shapiro. Experience based deductive learning. In *The Third International Conference on Tools for Artificial Intelligence*, San Jose, CA, 1991. IEEE. forthcoming.

[2] Joongmin Choi and Stuart C. Shapiro. Learning in deduction by knowledge migration and shadowing. In *AAAI-91 Workshop on Knowledge Acquisition: From Science to Technology to Tools*, Anaheim, CA, 1991. AAAI.

[3] H. Coelho, J. C. Cotta, and L. M. Pereira, editors. *How to Solve it with PROLOG*. Laboratório Nacional de Engenharia Civil, Ministério da Habitação e Obras Públicas, 3 edition, 1982.

[4] John S. Conery and Dennis F. Kibler. AND parallelism in logic programs. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 539–543, Karlsruhe, Germany, August 1983. William Kaufmann, Los Altos.

[5] Charles L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1979.

[6] Charles L. Forgy. RETE: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[7] Russell Greiner. Finding optimal derivation strategies in redundant knowledge bases. *Artificial Intelligence*, 50:95–115, 1991.

[8] Richard G. Hull. A new design for SNIP the SNePS Inference Package. SNeRG Technical Note 14, Dept. of Computer Science, State University of New York at Buffalo, 1986.

[9] João P. Martins and Stuart C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35:25–79, 1988.

[10] Donald P. McKay and Stuart C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 368–374, Vancouver, Canada, August 1981.

[11] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 42–47, Seattle, WA, July 1987. Morgan Kaufmann, Los Altos.

[12] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[13] Stuart C. Shapiro. Compiling deduction rules from a semantic network into a set of processes. In *Abstracts of Workshop on Automatic Deduction*, MIT, Cambridge, MA, 1977.

[14] Stuart. C. Shapiro. Numerical quantifiers and their use in reasoning with negative information. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 791–796, Tokyo, Japan, August 1979.

[15] Stuart C. Shapiro. SNePS semantic network processing system. In N.V. Findler, editor, *Associative Networks : Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[16] Stuart. C. Shapiro. Using non-standard connectives and quantifiers for representing deduction rules in a semantic network. Invited paper presented at Current Aspects of AI Research, a seminar held at the Electrotechnical Laboratory, Tokyo, 1979.

[17] Stuart C. Shapiro, João Martins, and Donald P. McKay. Bi-directional inference. In *Proceedings of the Fourth Annual Meeting of the Cognitive Science Society*, pages 90–93, Ann Arbor, MI, 1982.

[18] David E. Smith. Controlling backward inference. *Artificial Intelligence*, 39:145–208, 1989.

[19] Richard Treitel and Michael R. Genesereth. Choosing directions for rules. *Journal of Automated Reasoning*, 3:395–431, 1987.

[20] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-hall, Inc., Englewood Cliffs, NJ, 1984.