Hh Handout 1

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

MANAGING UNCERTAINTY IN EXPERT SYSTEMS

by

Jerzy W. Grzymala-Busse
University of Kansas

KLUWER ACADEMIC PUBLISHERS
Boston/Dordrecht/London

quantifiers ($\forall$, $\exists$),

parentheses and *commas* ((,), ,).

Symbol $\forall$ means *universal quantifier*, "for all", while $\exists$ means *existential quantifier*, "there exists". The expression to which a quantifier is applied is the *scope* of the quantifier. An occurrence of an individual variable x is *bound* if and only if it is either an occurrence $(\forall x)$ or $(\exists x)$ or within the scope of a quantifier $(\forall x)$ or $(\exists x)$. Any other occurrence of a variable is a *free* occurrence. For example, in the expression $(\forall x)(R(x) \wedge S(y))$, the scope of the quantifier is $R(x) \wedge S(y)$, and thus x is a bound variable in both of its occurrences. The variable y occurs free, since even if it is within the scope of a quantifier, the quantifier is not on y.

Thus, the following are expressions of predicate calculus:

Sue = likes (someone), i.e., someone likes Sue,

isnice (Ann),

Sue = likes (Ann) $\wedge$ Ann = likes (Sue),

$(\forall x)\ (1 \geq f(x))$,

$(\exists x)\ (x \geq f(2))$,

$(\forall x)\ (\exists y)\ ((1 \geq f(x)) \wedge (y \geq f(x)))$.

Predicate calculus is of first order if quantifiers $\forall$ and $\exists$ are taken over individuals (i.e., they are of the form $(\forall x)$, ($\exists$someone)) but not over functions or predicates.

The set of all the preceding symbols is called the *alphabet* of the first-order predicate calculus. *Terms* are precisely those strings over the alphabet that may be obtained by finitely many applications of the following rules:

1. Every constant is a term,
2. Every variable is a term,
3. If $t_1, t_2, \ldots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, t_2, \ldots, t_n)$ is a term.

From the preceding examples, none is a term. Examples of terms are Ann, someone, likes(Ann).

Atomic formulas are expressions of the form

$$P(t_1, t_2, \ldots, t_n),$$

where P is an n-ary predicate symbol, $t_1, t_2, \ldots, t_n$ are terms, and $n \geq 1$. For example, isnice (Ann), $1 \geq 0$, $2 = 2$ are atomic formulas.

The *well-formed formulas* are expressions that can be built from the atomic formulas by the use of finitely many times connective symbols and quantifier symbols.

The preceding definitions describe the *syntax* (grammar) of the first-order predicate calculus. Credit for the *semantics* for the first-order predicate calculus belongs to A. Tarski (1933). The concepts of truth and falsity in the first-order predicate calculus are furnished by the following definitions.

A *domain* is any nonempty set. Given a set X of well-formed formulas of the first-order predicate calculus, an *interpretation* of X is a domain D together with an assignment to each n-ary predicate symbol of an n-ary predicate on D, to each n-ary function symbol of an n-ary function on D, to each individual constant symbol of a fixed element of D, and to the equality symbol = the identity predicate = in D, where for $a, b \in D$, $a = b$ is true if and only if a and b are the same.

A well-formed formula of the first-order predicate calculus is *satisfiable in a domain* D if and only if there exists an interpretation with domain D and assignments of elements of D to the free occurrences of individual variables in the formula such that the resulting proposition is true. A well-formed formula is *valid in a domain* D if and only if for every interpretation with domain D and every assignment of elements of D to the free occurrences of individual variables in the formula the resulting proposition is true. A well-formed formula is *satisfiable* if and only if it is satisfiable in some domain, and it is *valid* if and only if it is valid in all domains. In propositional calculus valid formulas are tautologies.

Predicate calculus is characterized by completeness and soundness, but not by decidability.

2.2 Production Systems

Production systems were first proposed by E. Post in 1943, but their current form was introduced by A. Newell and H. A. Simon in 1972 for psychological modeling and by B. G. Buchanan and E. A. Feigenbaum in 1978 for expert systems.

A production system consists of

1. A *knowledge base*, also called a *rule base*, containing *production rules* (or *rules*, or *productions*),
2. A *data base*, containing *facts*,
3. A *rule interpreter*, also called a *rule application module*, to control the entire production system.

2.2.1 Production Rules

Production rules are units of knowledge of the form

If conditions

then actions.

The *condition part* of the production rule is also called the *IF part, premise, antecedent*, or *left-hand side* of the rule, while the *action part* of the rule is also called the *THEN part, conclusion, consequent, succedent*, or *right-hand side* of the rule. Actions are executed when conditions are true and the rule is fired.

The name *production rule* covers a whole spectrum of different concepts. The first way to classify production rules is with respect to the restrictions on logical connectives between conditions and actions. Usually production rules are of the following form:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A_1 \vee A_2 \vee \ldots \vee A_n,$$

where $m, n \geq 1$ (see Subsection 2.3.2 and the definition of the *Kowalski form*).

An atomic formula C_i, $i = 1, 2, \ldots, m$, or A_j, $j = 1, 2, \ldots, n$, may be represented by a triple (entity, attribute, value); for example:

(person, weight, light),

(Jan, isnice, true),

or

(mycar, battery, weak).

In some cases (e.g., when the entity is known, is not essential, or is unique) triples may be replaced by pairs (attribute, value); for example:

(weight, small),

(isnice, true),

or

(battery, weak).

In the preceding examples attributes are unary functions or predicates, called *nominal descriptors* in Dietterich and Michalski (1983). Attributes that are k-ary functions or predicates for $k > 1$ are called *structured descriptors* in Dietterich and Michalski (1983). Examples of atomic formulas expressed by structured descriptors are

((node1, node2), arcs, 3),

((Lawrence, Topeka), distance, 20),

or

((Ann, Sue), likes, true).

In many expert systems production rules the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A_1 \vee A_2 \vee \ldots \vee A_n,$$

is reduced further to the list of the following forms

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A_1$$

or

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A_2$$

or

. . . .

or

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A_n.$$

The form $C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow A$ is the *Horn clause form*. Using this form, another production rule of the form

$$C_1 \vee C_2 \rightarrow A$$

should be substituted by the following two production rules:

$$C_1 \rightarrow A$$

and

$$C_2 \rightarrow A.$$

Another way to classify production rules is according to the kind of action. In general, an action is to *change* the content of the data base. Production rules in which actions are restricted exclusively to *add* facts to the data base are called *inference rules*.

In the case of production systems dealing with uncertainty, even more types of production rules are determined by the corresponding approaches to uncertainty. That will be discussed in the following chapters.

Note that production rules have *names* and may be equipped with additional features, not always following from uncertainty. One of them may be a *time tag*, to carry the information about the last time the production rule was used by the interpreter.

2.2.2 Data Base

A data base holds facts, usually in the form of triples (entity, attribute, value), where the entity is a constant; for example:

(Jan, weight, light)

or

(car# 42, battery, weak).

The content of the data base, in general, is changed cyclically by an interpreter. The facts may have time tags, so that the time of their insertion into the data base can be determined.

2.2.3 Rule Interpreter

The rule interpreter works iteratively in recognize-and-act cycles. In such a cycle, the interpreter first *matches* the condition part of the production rules against the facts in a data base, recognizing all *applicable rules*. Then it selects one of the applicable rules and *applies* the rule (*fires* or *executes* it). As a result, the action part of the production rule is inserted into the data base and the content of the data base is changed by the rule. Then the interpreter goes to the next recognize-and-act cycle. The interpreter stops its cycling when the problem is solved or a state is reached in which no rules are applicable.

2.2.3.1 Pattern Matching

Usually, the terms of conditions and actions of rules are written as triples (entity, attribute, value), where entities are variables, and facts are represented by similar triples, although of a different type, because the entities are constants. The problem of *pattern matching* arises, that is, matching triples of different types. For example:

(person, yearly income, greater than $15,000)
$\wedge$ (person, value of house, greater than $30,000)
$\rightarrow$ (person, loan to get, less than $3,000)

is a production rule, while

(John, yearly income, greater than $15,000)

and

(John, value of house, greater than $30,000)

are facts.

Before matching may be performed, the variable "person" must be assigned a constant value. The assignment of the constant "John" to variable "person" makes the first two patterns in the production rule identical to the corresponding facts. Thus, the firing of the production rule in forward chaining causes the new fact (John, loan to get, less than $3,000) to be added to the data base.

During a cycle performed by an interpreter, most of the time is spent on pattern matching. Thus it is important to find an efficient algorithm for pattern matching. The most popular one is the Rete match algorithm (Forgy, 1982). This algorithm takes advantage of pattern similarity and temporal redundancy. The former means that a rule is tested against the same contents of data base, and some matching can be done at the same time because many patterns are similar. The latter follows from the fact that the contents of the data base, although changed after each cycle, are modified only a little. Thus, for two consecutive cycles, most of the information necessary for pattern matching may be saved. The Rete matching algorithm is used in the rule-based language OPS5, a language used for programming expert systems.

2.2.3.2 Conflict Resolution

Recognition may be divided into *selection* and *conflict resolution*, where "selection" means the identification of all applicable rules, based on pattern matching, and "conflict resolution" means the choice of which rule to fire. Some approaches to conflict resolution are listed here—they may be used in combination.

- *The most specific rule.* Thus, if the facts in the data base are P and Q and the rules are $P \rightarrow R$ and $P \wedge Q \rightarrow S$, then both rules are applicable, and the second should be fired, because its condition part is more detailed,
- *The rule using the most recent facts.* Facts must have time tags,
- *Highest priority rule.* Rules must have assigned priorities,
- *The first rule.* Rules are linearly ordered and the least applicable rule is fired,
- *No rule is allowed to fire more than once on the basis of the same contents of the data base.* This eliminates firing the same rule all the time.

2.2.4 Forward Chaining

Forward chaining is also called *data-driven, bottom-up,* or *antecedent chaining.* During the selection time of each cycle, the interpreter is looking for applicable rules by matching condition parts of rules with the current contents of the data base. It is necessary to recognize when to stop applying rules. The condition to terminate the process is either when the goal is reached or when all possible facts are already inferred from the initial data base.

Consider the following trivial example of a production system that might be used in troubleshooting car problems. For simplicity, variables in rules are ignored in the example, so that pattern matching is not necessary. The rules are

R1. **If** the ignition key is on
and the engine won't start,
then the starting system (including the battery) is faulty.

R2. **If** the starting system (including the battery) is faulty
and the headlights work,
then the starter is faulty.

R3. **If** the starting system (including the battery) is faulty
and the headlights do not work,
then the battery is dead.

R4. **If** the voltage test of the ignition switch shows 1 to 6 volts,
then the wiring between the ignition switch and the solenoid is OK.

R5. **If** the wiring between the ignition switch and the solenoid is OK,
then replace the ignition switch.

Initially, the data base contains the following facts:

A. The ignition key is on,

B. The engine won't start,

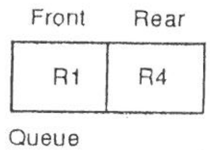

Figure 2.1 The Queue before the First Rule Firing

C. The headlights work,

D. The voltage test of the solenoid shows 1 to 6 volts.

In the example it is assumed that forward chaining is applied, and that a rule is applicable only if its condition part is true and if its action part adds a new fact to the data base. Moreover, rules are ordered according to their names (i.e., R1 precedes R2, R2 precedes R3, etc.) and they are scanned by the interpreter in that order. Applicable rules are successively inserted into a queue on a first-come, first-served basis. However, if a rule is already in the queue, then it is not inserted again. Conflict resolution is taken into account by firing the rule removed from the front of the queue. The goal is inferring all possible facts from the initial data base.

Initially, the applicable rules are R1 and R4, and they are inserted at the end of the queue, as illustrated by Figure 2.1.

Rule R1 is removed from the front of the queue and fired. Thus the new fact

E. The starting system (including the battery) is faulty

is added to the data base.

In the second cycle produced by the interpreter, rules are classified as follows:

- Rule R1 is no longer applicable, since its action part would add E to the data base, and it is already there.
- Rule R2 is applicable, and it is added to the end of the queue.

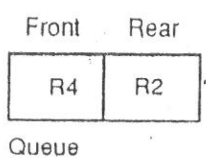

Figure 2.2 The Queue before the Second Rule Firing

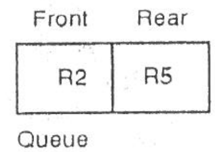

Figure 2.3 The Queue before the Third Rule Firing

- Rule R3 is not applicable; rule R4 is applicable, but it is already in the queue; rule R5 is not applicable.

Finally the queue contains two rules, R4 and R2, as illustrated by Figure 2.2. Rule R4 is removed from the front of the queue and fired, so that the new fact

F. The wiring between the ignition switch and the solenoid is OK

is added to the data base.

During the third cycle, after scanning the rules, rule R5 is inserted at the end of the queue (see Figure 2.3). Rule R2 is removed from the front of the queue and fired, producing the new fact

G. The starter is faulty,

which is then added to the data base.

As the interpreter discovers in the next cycle, no new rule may be inserted at the end of the queue, as the queue is represented by Figure 2.4. Rule R5 is removed from the front of the queue and fired, producing the new fact

H. Replace the ignition switch,

which is then added to the data base.

In the next cycle there are no applicable rules, nor are there any rules remaining in the queue from earlier cycles, so the interpreter halts the computation.

It can be seen that rules R1, R2, R3, R4, and R5 might be fired in different

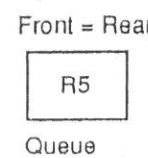

Figure 2.4 The Queue before the Fourth Rule Firing

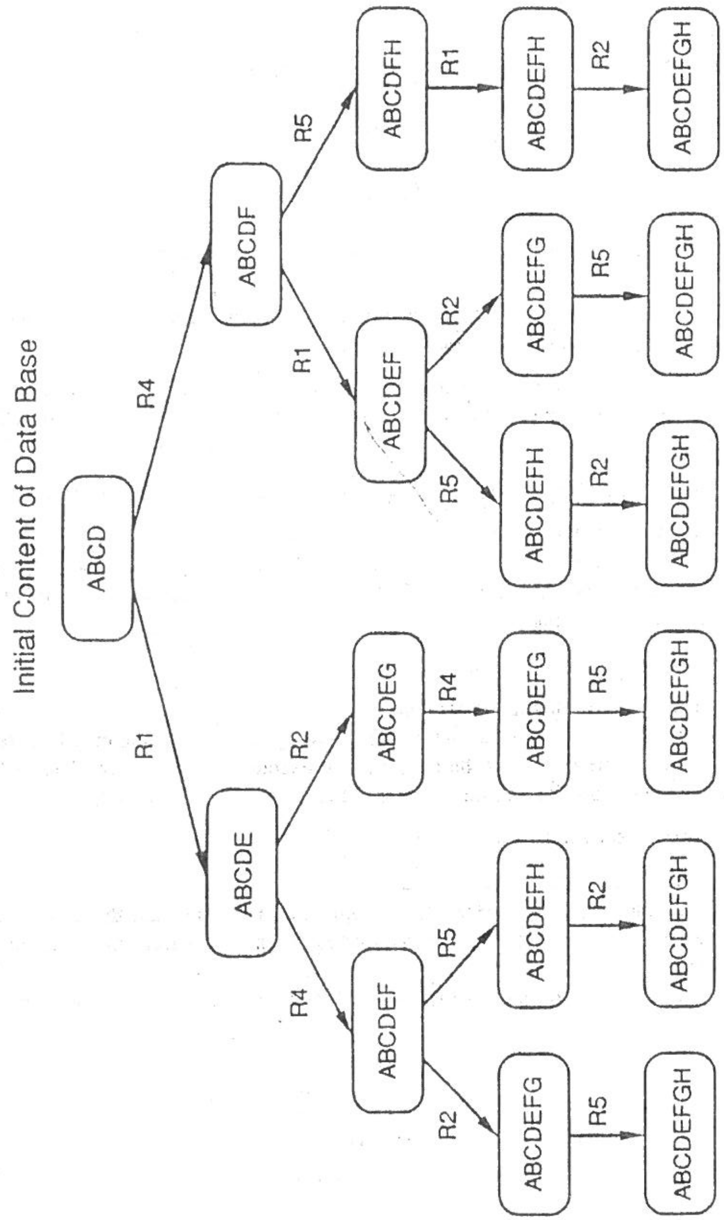

Figure 2.5 Search Space

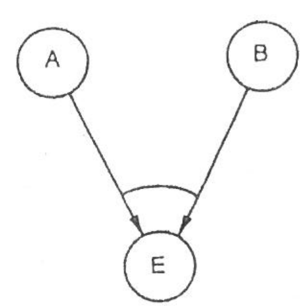

Figure 2.6 An Inference Network for Rule R1

sequences. The *search space* for the problem is presented in Figure 2.5, where nodes represent current contents of the data base and arcs depict applicable rules.

From Figure 2.5 it follows that with initial contents of the data base rules, R1 or R4 may fire. If the fired rule is R4, then rules R1 or R5 may fire, and so on.

The preceding sets of facts and rules may be represented by an *inference network*, a concept discussed in Duda *et al.* (1979). In the inference network, facts are viewed as nodes and rules as arcs. For example, rule R1, which says

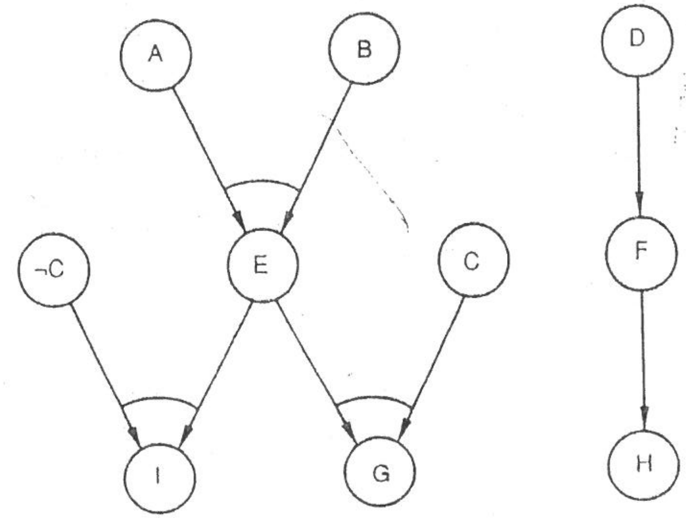

Figure 2.7 An Inference Network

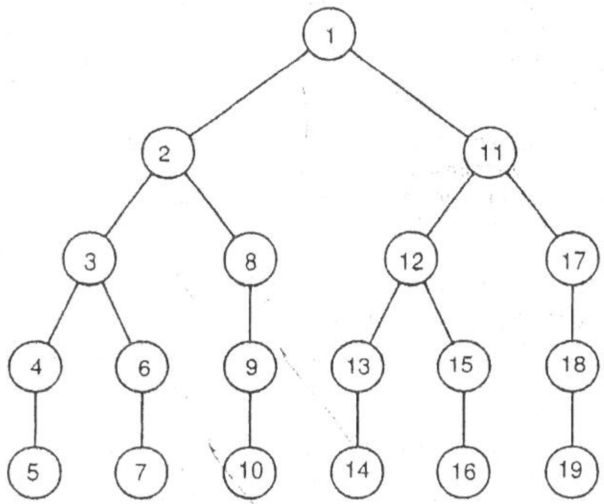

Figure 2.8 Depth-First Search

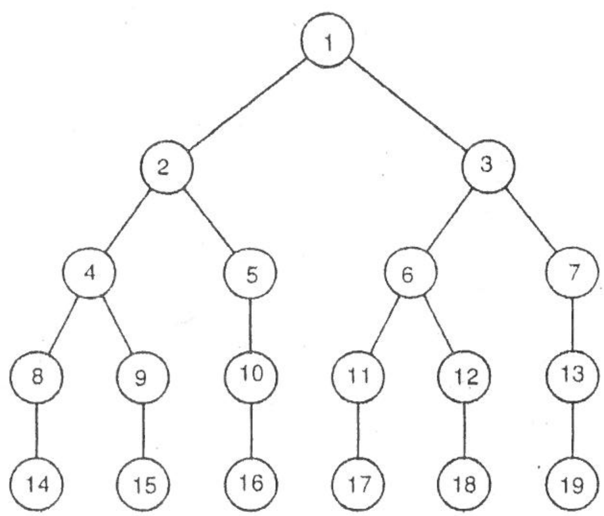

Figure 2.9 Breadth-First Search

If A and B then E

is presented in Figure 2.6.

The minor arc, connecting arrows from A to E and from B to E, indicates the **and** connective, which appears in rule R1 between facts A and B. The inference network illustrating the preceding example is presented in Figure 2.7, where I denotes the following fact:

The battery is dead.

Inference networks are helpful in solving problems associated with production systems. In some systems handling uncertainty, inference networks are used as useful models of knowledge propagation (see Section 4.2).

2.2.5 Depth-First and Breadth-First Search

The tree in Figure 2.5 may be traversed starting at the root and then following the leftmost path to a leaf, then starting from the last choice-point, again following the leftmost path to a leaf, and so on. This is depicted in Figure 2.8.

Another way of traversing the same tree is presented in Figure 2.9. Now the earliest choice-point is picked first, so that all nodes on a given level are visited from left to right before going on to the next level.

The main disadvantage of the depth-first search is that before the identification of the shortest path, many other long paths may be traced. The advantage is the simplicity of the implementation. Another advantage of the depth-first search is at the same time the disadvantage of the breadth-first search, namely, the breadth-first search needs more memory, since all nodes at a given level to the left and all nodes at the preceding level must be memorized. The advantage of the breadth-first method is that the first-found solution is always the shortest path.

2.2.6 Backward Chaining

In *backward chaining*, also called *goal-driven*, *top-down*, or *consequent chaining*, the production system establishes whether a goal is supported by the data base. For example, the goal is fact F. First, it should be checked whether F is in the data base. If so, F is supported by the data base. If not, but $\neg F$ is in the data base, the goal should be rejected. If neither F nor $\neg F$ is in the data base, applicable rules are determined. In **backward** chaining, applicable rules are recognized by matching action parts of rules with fact F. Let R be an applicable rule, selected by the interpreter. The condition part $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ of rule R is now checked against the data base. If all $C_1, C_2, \ldots, C_m$, after the substitutions determined by matching, are in the data base, the solution is reached and F is true. Let's say that C is any of $C_1, C_2, \ldots, C_m$ (again, after corresponding substitutions). If $\neg C$ is present in the data base, then R cannot be used and another rule should be selected. If neither C nor $\neg C$ can be found in the data base, then C is a *subgoal* and the preceding procedure should start again the same way it is described for F. If no applicable rule exists and the truth of F is not established, the system may ask the user to provide additional facts or rules.

As an example, the same set of rules R1, R2, R3, R4, and R5 and initial content of the data base as in Section 2.2.4 will be considered. The action of rule R3

The battery is dead

will be denoted by I.

The goal is $H \wedge I$. First, H will be considered. H is not in the data base. The only rule whose action part matches H is R5.

The condition part of R5 is F. F is not in the data base, so it is a subgoal. The only rule with the action part matching F is R4. The condition part of R4 is D, and it is in the data base, so F is supported, and hence H is supported. Next I must be checked. I is not in the data base, so applicable rules are sought. The only such rule is R3. The condition part of R3 is $\neg C \wedge E$. C is in the data base, hence R3 can not be used. Because this is the only applicable rule to match I, I is not supported by the data base and the entire goal should be rejected.

Usually backward chaining is executed as depth-first search. Backward chaining is used in applications with a large amount of data (e.g. in medical diagnosis). If the goal is to infer a specific fact, forward chaining may waste time, inferring a lot of unnecessary facts. On the other hand, during backward chaining a very large tree of possibilities may be constructed.

Forward and backward chainings are two basic forms of inference in production systems. However, mixed strategies are frequently used in practice. For example, given facts and rules, forward chaining may be used initially and then backward chaining is applied to find other facts that support the same goal.

2.2.7 Metarules

Metarules are rules about rules. They may be *domain-specific*, like

If the car does not start,
  then first check the set of rules about the fuel system,

or *domain-free*, i.e., not related to the specific domain, like

If the rules given by the owner's manual apply
  and the rules given by the textbook apply,
  then check first the rules given by the owner's manual.

Metarules reduce computation time by eliminating futile searching for a solution. The reduction may be achieved by *pruning* unnecessary paths in the search space, i.e., by ignoring such paths.

2.2.8 Forward and Backward Reasoning Versus Chaining

Forward and backward chaining were described previously. Both concepts are related to the way rules are activated by the interpreter.

On the other hand, the way *reasoning* is done depends on how the entire program is organized or what the problem-solving strategy is. If that strategy is bottom-up, the reasoning is forward. If it is top-down, the reasoning is backward (see Jackson 1986).

For example, XCON, an expert system designed to configure VAX computers, uses forward chaining and backward reasoning. The main goal, to configure a system, is divided into subgoals, to configure its components, and so on. Thus, the reasoning is backward.

2.2.9 Advantages and Disadvantages of Production Systems

The most obvious advantage of production systems is their modularity. Production rules are the independent pieces of knowledge, so that they may be easily added to or deleted from the knowledge base. The knowledge of experts is expressed in a natural way using rules, and then the rules may be easily understood by people not involved in expert system building.

The disadvantages follow from the inefficiency of big production systems with rules that are not organized in any kind of structure and from the fact that algorithms are difficult to follow. Still, rule-based expert systems are the most popular among all such systems.

2.3 Semantic Nets

Semantic nets and frames belong to a class of knowledge representations called "slot and filler" or "structured object". Semantic nets were first used by M. R. Quillian in 1968. At the same time, B. I. Raphael independently used the concept as a model of human memory. Semantic nets proved their usefulness in representation of natural language sentences.

A *semantic net* is a directed labeled graph, in which *nodes* represent entities, such as objects and situations, and *arcs* represent binary relations between enti-

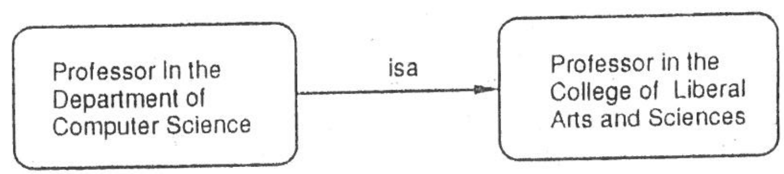

Figure 2.10 An *isa* Arc

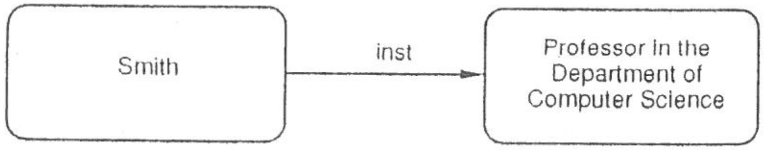

Figure 2.11 An *inst* Arc

ties. *Labels* on nodes and arcs represent their names.

2.3.1 Basic Properties

One of the key ideas in artificial intelligence is the *isa hierarchy*. The concept may be expressed using two predicates, *isa* ("is a") and *inst* ("is an instance of"). The former indicates that a class is a specific case of another class, while the latter says that a specific element belongs to a class. In both cases, objects of a specific nature *inherit properties* of objects of a more general nature.

For example, a class represented by a professor in the Department of Computer Science is a subclass of the class represented by a professor in the College of Liberal Arts and Sciences. On the other hand, Smith is an instance of a class represented by a professor in the Department of Computer Science. This is illustrated by Figures 2.10 and 2.11.

Semantic nets are natural representations for domains where reasoning is based on property inheritance. A simple example is presented in Figure 2.12. As shown, Smith uses CSNET, has a Ph.D. degree, and teaches CS 747.

A problem arises when we try to represent n-ary relations using semantic nets. For example, one wishes to express not only that Smith teaches CS 747, but also that he uses the textbook "AI" and that the location of his course is room 111 in Green Hall. To do that, an additional node is necessary, to represent the 4-ary relation "teaching", as illustrated by Figure 2.13.

2.3.2. Extended Semantic Nets

A version of semantic nets, *extended semantic nets*, was introduced in Deliyanni and Kowalski (1979). The main idea is based on "clausal form", introduced by R. A. Kowalski, also known as the "Kowalski clausal form" (Frost, 1986).

A *clause in the Kowalski form* is an expression of the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow K_1 \vee K_2 \vee \ldots \vee K_n,$$

where $C_1, C_2, \ldots, C_m, K_1, K_2, \ldots, K_n$ are atomic formulas, $m \geq 0$ and $n \geq 0$. The atomic formulas $C_1, C_2, \ldots, C_m$ are called *conditions*, and atomic formulas $K_1, K_2, \ldots, K_n$ are called *conclusions*.

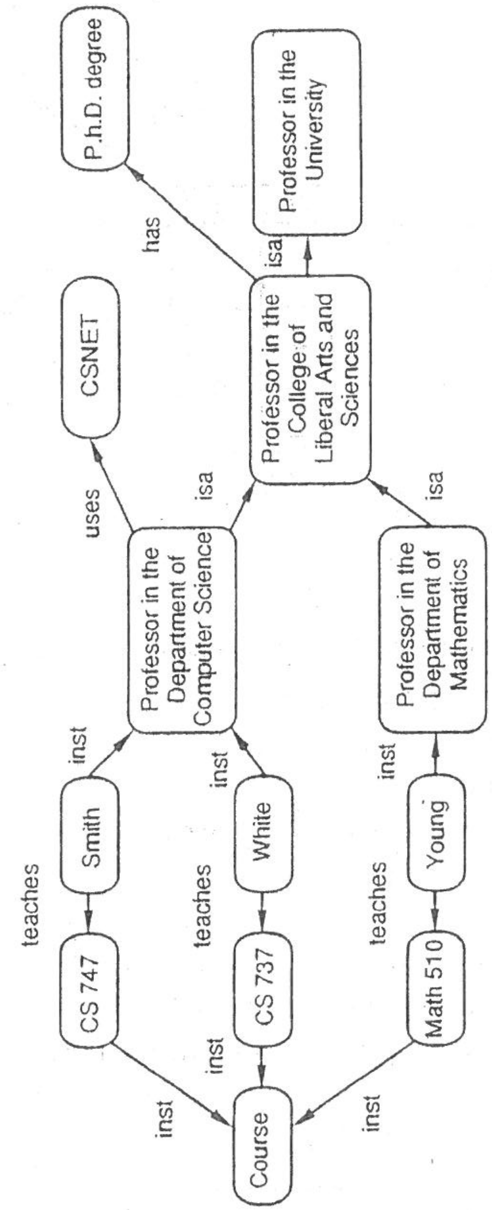

Figure 2.12 A Property Inheritance

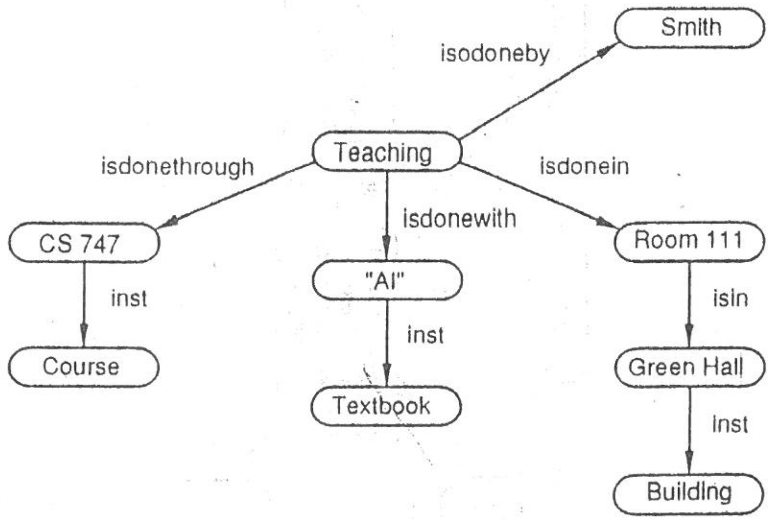

Figure 2.13 Representation of 4-ary Relation

For $m = 0$ the form is reduced to

$$\rightarrow K_1 \vee K_2 \vee \ldots \vee K_n$$

and is interpreted as

$$K_1 \vee K_2 \vee \ldots \vee K_n.$$

When $n = 0$, then the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m \rightarrow$$

is interpreted as the denial of $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ and written as

$$\neg(C_1 \wedge C_2 \wedge \ldots \wedge C_m).$$

If $m = n = 0$, then the form should be interpreted as a fallacy.

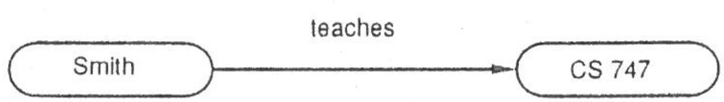

Figure 2.14 Conclusion Arc

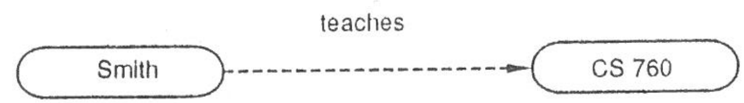

Figure 2.15 Condition Arc

In extended semantic nets, nodes represent terms, and arcs can represent binary relations. Two different types of arcs are used. Components of a condition are connected by the broken arrow, while components of a conclusion are connected by the ordinary arrow. Thus, the clause

$\rightarrow$ Smith teaches CS 747

is represented by Figure 2.14, while

Smith teaches CS 760 $\rightarrow$

meaning that Smith does not teach CS 760, is represented by Figure 2.15. Figure 2.16 illustrates the clause

Smith inst professor $\rightarrow$ (Smith inst male $\vee$ Smith inst female).

An example of the extended semantic net is presented in Figure 2.17. This net represents the following set of clauses:

$\rightarrow$ Martin likes Robin,

$\rightarrow$ Robin likes corn,

$\rightarrow$ corn isa food,

Robin inst bird $\rightarrow$ Robin hasa beak,

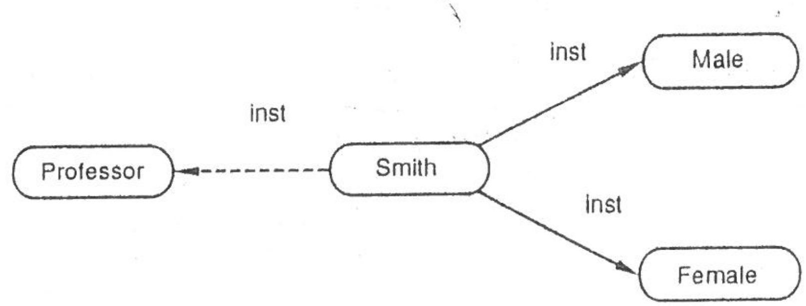

Figure 2.16 A Clause

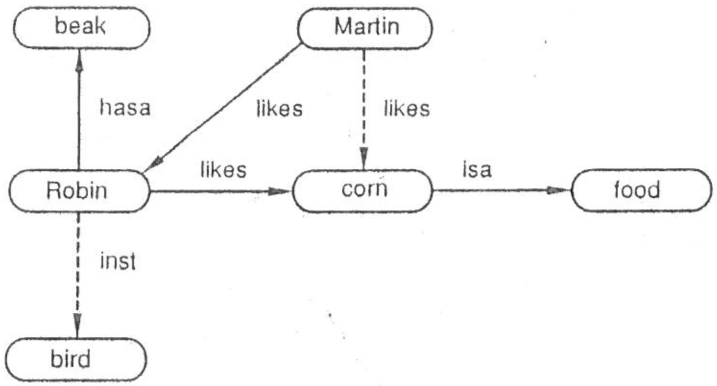

Figure 2.17 An Extended Semantic Net

Martin likes corn →.

2.3.3 Concluding Remarks

An example of the application of semantic nets to expert systems is PROSPECTOR, developed by Stanford Research Institute between 1974 and 1983. PROSPECTOR, a rule-based expert system (like KAS, a shell derived from it) uses a semantic net to organize production rules.

Inference in semantic nets is based on inheritance, but some other mechanisms are used as well, e.g., matching a fragment of the net with the entire net.

2.4 Frames

The concept of a frame as used here was introduced by M. Minsky in 1975. A frame system is a generalization of a semantic net. Frames are designed for holding clusters of knowledge. They are similar to semantic nets because frames are also linked together in a net.

Originally, frame representations were used as part of pattern-recognition systems, especially for understanding of natural language.

2.4.1 Basic Concepts

A *frame* is a data structure to represent an entity or an entity class. It contains a collection of memory areas called *slots*. Slots may have different sizes and may be filled by pieces of different kinds of knowledge.

Contents of slots may be categorized into *declarative* and *procedural*. Declarative content may be represented by *attributes* and their *values*, *descriptions*, or *graphical explanations*, *pointers* to other frames, *collections of rules*, and other frames.

Name Professor in the Department of Computer Science		
	Value:	*Procedure*:
Slot: Age *Condition*	18 ≤ Age ≤70	If-wrong
Slot: Ph.D. in		
Slot: Tenure		
Slot: Promotion rules		
Slot: Languages known	English	

Name Full Professor		
	Value:	*Procedure*:
Slot: Age *Condition*		
Slot: Ph.D. in		
Slot: Tenure	Yes	
Slot: Promotion rules		
Slot: Languages known	English	

Figure 2.18 Generic Frames

Name John Smith		
	Value:	*Procedure:*
Slot: Age *Condition*	45	
Slot: Ph.D. in	CS	
Slot: Tenure	Yes	
Slot: Promotion rules		
Slot: Languages known	English	

Figure 2.19 A Specific Frame

Some slots may contain, in addition, procedures that are triggered when the value of the attribute of that slot is changed. These procedures may have names like "If-added", "If-removed", "If-needed", and so on. They represent the procedural aspect of the slot content. Such procedures are called *demons*.

Frames are labeled by their names. When a frame represents a general concept, the frame is called *generic*. Frames containing specific information are said to be *specific* or *instantiated*. Examples of generic frames are given in Figure 2.18, and an example of a specific frame is presented in Figure 2.19.

Slot "languages known" in the frame "Professor in the Department of Computer Science" already has one of the possible values filled, i.e., "English". Such a value will occur in any instantiation of that frame and is called a *generic value*. Each frame from Figure 2.20 *inherits* that value.

Slot "tenure" in the frame "Full Professor" is filled by "Yes". It is expected that a full professor has tenure, so such a value is called a *default value*. If during the instantiation of that frame for a specific professor, it turned out that the status of "tenure" is "No", the default value will be changed.

In the frame "Professor in the Department of Computer Science", slot "Age" is furnished with a *condition* $18 \leq \text{Age} \leq 70$. An attempt to fill that slot with a value not satisfying the condition may trigger the attached procedure "If-wrong" and ask the user whether the value really is not an error in filling.

Frames may be linked together in taxonomical structures like semantic nets, using the same arcs "isa" and "inst", as shown in Figure 2.20.

Frame systems may be used to organize production rules, so the resulting knowledge representation is *hybrid*. Thus, rules are no longer unorganized, and

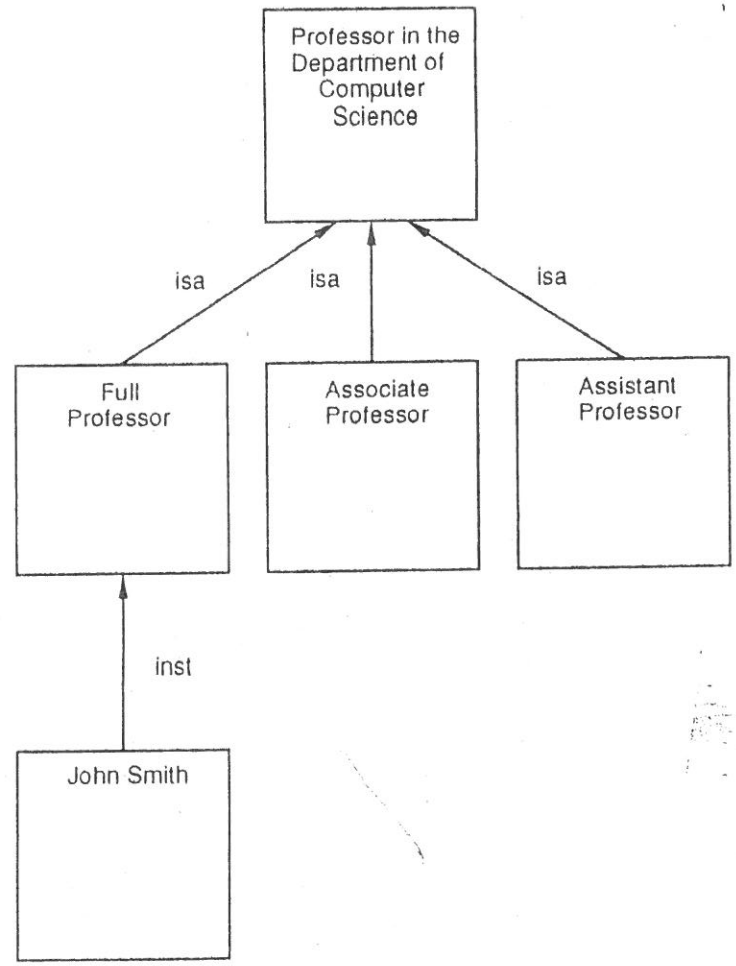

Figure 2.20 A Frame System

in one system advantages of both types of knowledge representation may be balanced.

2.4.2 Inference

Some types of inference can be made using frames. The first type of inference is based on structural properties of frames and taxonomical structures of a system. Some methods use inheritance, as in semantic nets. For example, generic and default values of a frame are inferred as values of corresponding slots in all frames linked to that frame by "isa" or "inst" links.

Another type may be based on recognition of illegal values of a slot. In such a situation, the system may exclude the illegal value and call the user for help.

A third type of inference is done by frame matching, i.e., looking for a frame in the frame system matching a given frame. In practice, a perfect match is difficult to accomplish, so the goal is to have as good a match as possible.

2.4.3 Advantages and Disadvantages of Frame Systems

The main advantage of frame systems follows from the fact that a single frame represents an entity or an entity class by a cluster of knowledge. A frame system represents a number of valuable aspects of the nature of knowledge, such as a taxonomy, generic values, and default values. Moreover, both procedural and declarative knowledge can be represented.

The main disadvantages are lack of a formal theory of frames and lack of a uniform way to deal with uncertainty. Frame representation of knowledge, frequently used in conjunction with production rule representation, is used in many expert systems and shells.

Exercises

1. For forward chaining write an algorithm, in pseudo-Pascal or pseudo-Lisp, or a program, in Pascal or Lisp, to build a tree representing search space, using

a. Depth-first approach,

b. Breadth-first approach.

2. A production system is described as follows:

Rules

R1. $A \wedge B \rightarrow C$,

R2. $A \wedge \neg D \rightarrow E$,

R3. $C \wedge \neg D \rightarrow E$,

R4. $C \wedge D \rightarrow F$,

R5. $E \wedge F \rightarrow L$,

R6. $E \wedge H \rightarrow \neg G$,

R7. $E \wedge \neg H \rightarrow G$,

R8. $I \rightarrow J$,

R9. $J \rightarrow K$.

Conflict Resolution

i. Rules are ordered according to their names.

ii. The first applicable rule is selected.

iii. During each session, each rule may be fired once.

a. Tell what the content of the data base is after forward-chaining session if the initial content of the data base is

$$\{A, B, \neg D, \neg H, I\},$$

b. As (a), but the initial content is

$$\{A, B, D, E, I\},$$

c. Tell whether the goal $\{L\}$ is supported by the following content of the data base

$$\{A, B, \neg D, E\}$$

(use backward chaining),

d. As (c), but the goal is $\{K, L\}$ and the content is

$$\{A, \neg D, \neg H, I\}.$$

3. A production system is described as follows:

Rules

R1. $A \wedge B \rightarrow C$,

R2. $\neg A \rightarrow H$,

R3. $B \wedge D \rightarrow \neg C$,

R4. $D \wedge E \rightarrow G$,

R5. $C \rightarrow I$,

R6. $\neg C \wedge G \rightarrow I$,

R7. $\neg C \wedge G \rightarrow H$,

R8. $\neg B \wedge \neg D \rightarrow J$,

R9. $I \rightarrow J$.

Conflict Resolution

i. Rules are ordered according to their names.

ii. The first applicable rule is selected.

iii. During each session, each rule may be fired once.

a. Tell whether the goal $\{H\}$ is supported by the following content of the data base:

$$\{A, \neg C, D, E\}$$

(use backward chaining),

b. As (a), but the goal is

$$\{J\}.$$

4. A production system is described as follows:

Rules

R1. $A \wedge B \rightarrow D$,
R2. $\neg A \rightarrow F$,
R3. $\neg A \wedge B \rightarrow \neg E$,
R4. $B \wedge C \rightarrow E$,
R5. $D \rightarrow G$,
R6. $D \wedge E \rightarrow I$,
R7. $D \wedge \neg E \rightarrow H$,
R8. $E \wedge F \rightarrow H$,
R9. $\neg E \rightarrow G$.

Conflict Resolution

i. Rules are ordered according to their names.
ii. The first applicable rule is selected.
iii. During each session, each rule may be fired once.

Tell whether the goals

a. $\{G\}$,
b. $\{H\}$,
c. $\{I\}$

are supported by the following content of data base:

$$\{\neg A, B, C\}.$$

Use backward chaining.

5. A production system is described as follows:

Rules

R1. $A \wedge B \rightarrow E$,

R2. $A \wedge C \rightarrow F$,
R3. $\neg A \wedge B \rightarrow D$,
R4. $\neg A \wedge \neg B \rightarrow E$,
R5. $B \wedge C \rightarrow F$,
R6. $\neg B \wedge \neg C \rightarrow \neg F$,
R7. $D \wedge \neg F \rightarrow H$,
R8. $D \wedge F \rightarrow G$,
R9. $E \wedge F \rightarrow H$,
R10. $E \wedge \neg F \rightarrow G$.

Conflict Resolution

Rules are ordered according to their names. During each scanning of the list of rules by the interpreter, applicable rules are pushed into a stack successively. After scanning, a rule popped from the top of the stack is fired. During each session, any rule may be fired once.

a. Tell what the content of the data base is after the forward chaining session if the initial content of the data base is

$$\{A, B, C\},$$

b. Tell whether the goal $\{G\}$ is supported by the following content of the data base

$$\{\neg A, \neg B, \neg C\}$$

(use backward chaining),

c. As (b), but the goal is

$$\{H\}.$$

6. A production system is described as follows

Rules

R1. $A \wedge B \rightarrow \neg D$,
R2. $A \wedge B \rightarrow \neg E$,
R3. $A \wedge \neg B \rightarrow D$,
R4. $\neg A \wedge B \rightarrow D$,
R5. $\neg A \wedge C \rightarrow \neg E$,
R6. $\neg B \wedge \neg C \rightarrow E$,
R7. $E \wedge \neg D \rightarrow G$,
R8. $\neg E \wedge D \rightarrow G$,

a. Tell whether the goal $\{H\}$ is supported by the following content of the data base:

$$\{A, \neg C, D, E\}$$

(use backward chaining),

b. As (a), but the goal is

$$\{J\}.$$

4. A production system is described as follows:

Rules

R1. $A \wedge B \to D$,

R2. $\neg A \to F$,

R3. $\neg A \wedge B \to \neg E$,

R4. $B \wedge C \to E$,

R5. $D \to G$,

R6. $D \wedge E \to I$,

R7. $D \wedge \neg E \to H$,

R8. $E \wedge F \to H$,

R9. $\neg E \to G$.

Conflict Resolution

i. Rules are ordered according to their names.

ii. The first applicable rule is selected.

iii. During each session, each rule may be fired once.

Tell whether the goals

a. $\{G\}$,

b. $\{H\}$,

c. $\{I\}$

are supported by the following content of data base:

$$\{\neg A, B, C\}.$$

Use backward chaining.

5. A production system is described as follows:

Rules

R1. $A \wedge B \to E$,

R2. $A \wedge C \to F$,

R3. $\neg A \wedge B \to D$,

R4. $\neg A \wedge \neg B \to E$,

R5. $B \wedge C \to F$,

R6. $\neg B \wedge \neg C \to \neg F$,

R7. $D \wedge \neg F \to H$,

R8. $D \wedge F \to G$,

R9. $E \wedge F \to H$,

R10. $E \wedge \neg F \to G$.

Conflict Resolution

Rules are ordered according to their names. During each scanning of the list of rules by the interpreter, applicable rules are pushed into a stack successively. After scanning, a rule popped from the top of the stack is fired. During each session, any rule may be fired once.

a. Tell what the content of the data base is after the forward chaining session if the initial content of the data base is

$$\{A, B, C\},$$

b. Tell whether the goal $\{G\}$ is supported by the following content of the data base

$$\{\neg A, \neg B, \neg C\}$$

(use backward chaining),

c. As (b), but the goal is

$$\{H\}.$$

6. A production system is described as follows

Rules

R1. $A \wedge B \to \neg D$,

R2. $A \wedge B \to \neg E$,

R3. $A \wedge \neg B \to D$,

R4. $\neg A \wedge B \to D$,

R5. $\neg A \wedge C \to \neg E$,

R6. $\neg B \wedge \neg C \to E$,

R7. $E \wedge \neg D \to G$,

R8. $\neg E \wedge D \to G$,