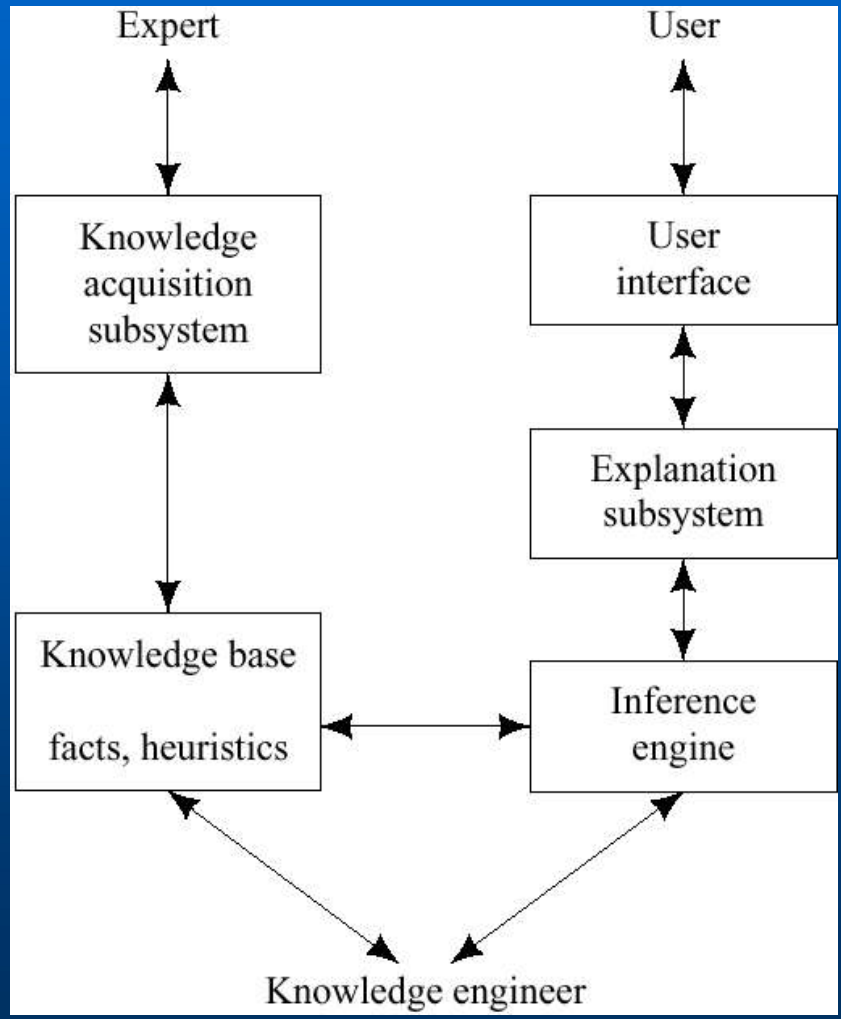
17.4 Rule-Based Expert Systems (1/9)

- *Expert Systems*
  - One of the most successful applications of AI reasoning technique using facts and rules
  - "AI Programs that achieve expert-level competence in solving problems by bringing to bear a body of knowledge [Feigenbaum, McCorduck & Nii 1988]"
- Expert systems vs. knowledge-based systems
- Rule-based expert systems
  - Often based on reasoning with propositional logic Horn clauses.

17.4 Rule-Based Expert Systems (2/9)

- Structure of Expert Systems
  - Knowledge Base
    - Consists of predicate-calculus facts and rules about subject at hand.
  - Inference Engine
    - Consists of all the processes that manipulate the knowledge base to deduce information requested by the user.
  - Explanation subsystem
    - Analyzes the structure of the reasoning performed by the system and explains it to the user.



17.4 Rule-Based Expert Systems (3/9)

- Knowledge acquisition subsystem
  - Checks the growing knowledge base for possible inconsistencies and incomplete information.
- User interface
  - Consists of some kind of natural language processing system or graphical user interfaces with menus.
- "Knowledge engineer"
  - Usually a computer scientist with AI training.
  - Works with an expert in the field of application in order to represent the relevant knowledge of the expert in a forms of that can be entered into the knowledge base.

17.4 Rule-Based Expert Systems (4/9)

Example: loan officer in a bank

"Decide whether or not to grant a personal loan to an individual."

OK (The loan should be approved.)
COLLAT (The collateral for the loan is satisfactory.)
PYMT (The applicant is able to the loan payments.)
REP (The applicant has a good financial reputation.)
APP (The appraisal on the collateral is sufficiently grater than the loan amount.)
RATING (The applicant has a good credit rating.)
INC (The applicant's income exceeds his/her expenses.)
BAL (The applicant has an excellent balance sheet.)

Facts

Rules

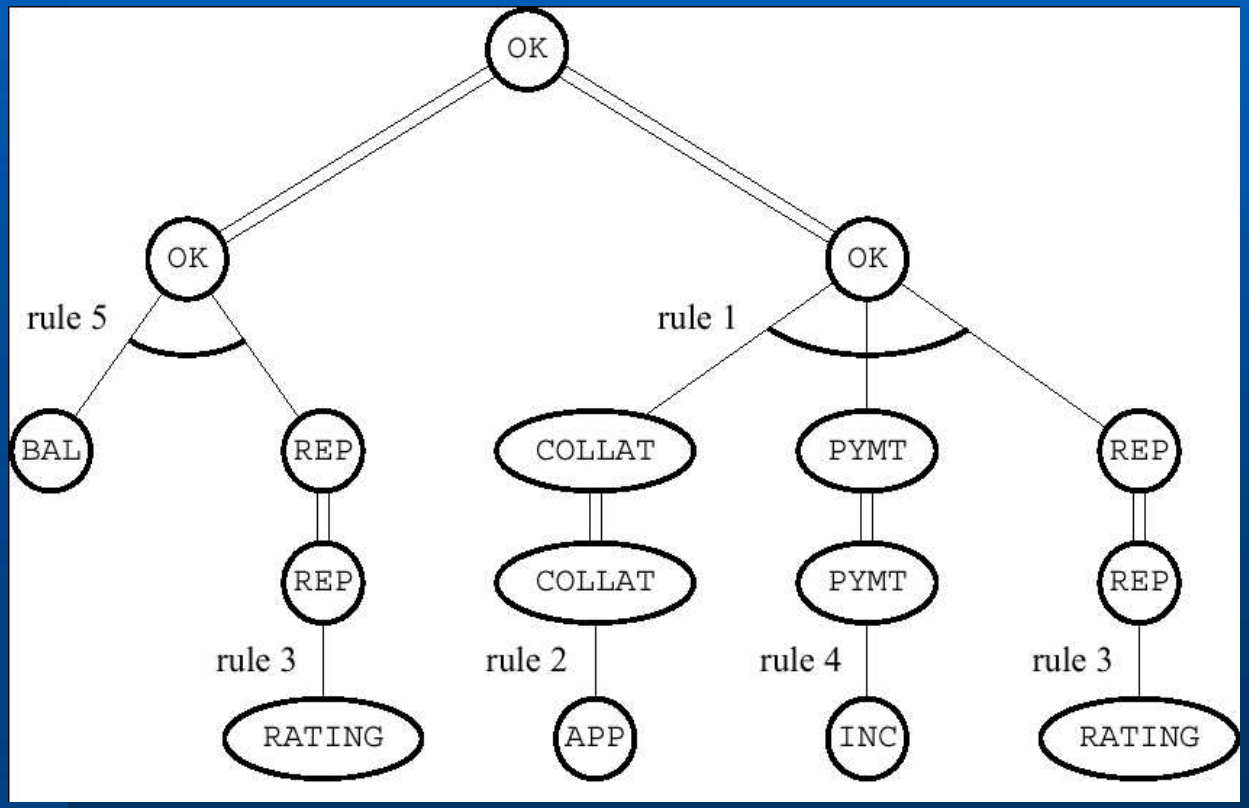
1. $COLLAT \wedge PYMT \wedge REP \supset OK$
2. $APP \supset COLLAT$
3. $RATING \supset REP$
4. $INC \supset PYMT$
5. $BAL \wedge REP \supset OK$

17.4 Rule-Based Expert Systems (5/9)

- To prove OK
  - The inference engine searches fro AND/OR proof tree using either backward or forward chaining.
- AND/OR proof tree
  - Root node: OK
  - Leaf node: facts
  - The root and leaves will be connected through the rules.
- Using the preceding rule in a backward-chaining
  - The user's goal, to establish OK, can be done *either* by proving *both* BAL and REP or by proving *each* of COLLAT, PYMT, and REP.
  - Applying the other rules, as shown, results in other sets of nodes to be proved.

17.4 Rule-Based Expert Systems (6/9)

- By backward-chaining



17.4 Rule-Based Expert Systems (7/9)

- *Consulting system*
  - Attempt to answer a user's query by asking questions about the truth of propositions that they might know about.
  - Backward-chaining through the rule is used to get to askable questions.
  - If a user were to "volunteer" information, bottom-up, forward chaining through the rules could be used in an attempt to connect to the proof tree already built.
  - The ability to give explanations for a conclusion
    - Very important for acceptance of expert system advice.
  - Proof tree
    - Used to guide the explanation-generation process.

17.4 Rule-Based Expert Systems (8/9)

- In many applications, the system has access only to uncertain rules, and the user not be able to answer questions with certainty.
- MYCIN [Shortliffe 1976]: Diagnose bacterial infections.

Rule 300

If : 1) The infection which requires therapy is meningitis, and

2) The patient does have evidence of serious skin or soft tissue infection, and

3) Organisms were not seen on the stain of the culture, and

4) The type of the infection is bacterial

Then : There is evidence that the organism (orther than those seen on cultures or smears) which might be causing the infection is staphylococcus - coag - pos (.75) ; streptococcus - group - a (.5).

17.4 Rule-Based Expert Systems (9/9)

- PROSPECTOR [Duda, Gaschnig & Hart 1979, Campbell, et al. 1982]
  - Reason about ore deposits.

> If there is a pre - intrusive, thorough - going fault system, then there is (5, 0.7) a regional environment favorable for a porphyry copper deposit.

- The numbers (.75 and .5 in MYCIN, and 5, 0.7 in PROSPECTOR) are ways to represent the certainty or strength of a rule.
- The numbers are used by these systems in computing the certainty of conclusions.

17.5 Rule Learning

- Inductive rule learning
  - Creates new rules about a domain, not derivable from any previous rules.
  - Ex) Neural networks
- Deductive rule learning
  - Enhances the efficiency of a system's performance by deducting additional rules from previously known domain rules and facts.
  - Ex) EBG (explanation-based generalization)

17.5.1 Learning Propositional Calculus Rules (1/9)

- Train rules from given training set
  - Seek a set of rules that covers only positive instances
    - Positive instance: OK = 1
    - Negative instance: OK = 0
  - From training set, we desire to induce rules of the form

    $\alpha_1 \wedge \alpha_2 \wedge \cdots \alpha_n \supset OK$ where $\alpha_i \in \{\text{APP}, \text{RATING}, \text{INC}, \text{BAL}\}$
  - We can make some rule more specific by adding an atom to its antecedent to make it cover fewer instances.
    - Cover: If the antecedent of a rule has value *True* for an instance in the training set, we say that the rule *covers* that instance.
  - Adding a rule makes the system using these rules more general.
  - Searching for a set of rules can be computationally difficult.
    - ➔here, we use "greedy" method which is called *separate and conquer*.

17.5.1 Learning Propositional Calculus Rules (2/9)

- Separate and conquer
  - First attempt to find a single rule that covers only positive instances
    - Start with a rule that covers all instances
    - Gradually make it more specific by adding atoms to its antecedent.
  - Gradually add rules until the entire set of rules covers all and only the positive instances.
  - Trained rules can be simplified using *pruning*.
    - Operations and noise-tolerant modifications help minimize the risk of overfitting.

17.5.1 Learning Propositional Calculus Rules (3/9)

- Example: loan officer in a bank
  - Start with the provisional rule $T \supset OK$.
    - Which cover all instances.
  - Add an atom it cover fewer negative instances-working toward covering only positive ones.
  - Decide, which item should we added ?
    - From $\{APP, RATING, INC, BAL\}$ by

$$r_\alpha = n_\alpha^+ / n_\alpha$$

n_α : the totoal number of instance covered by the antecedent of the rule after the addition of α to the antecedent.

n_α^+ : the totoal number of positive instance covered by the antecedent of the rule after the addition of α to the antecedent.

17.5.1 Learning Propositional Calculus Rules (4/9)

- Select that α yielding the largest value of r_α.

$r_{APP} = 3/6 = 0.5$

$r_{RATING} = 4/6 = 0.667$

$r_{INC} = 3/6 = 0.5$

$r_{BAL} = 3/4 = 0.75$

So, we select BAL, yielding the provisional rule.

$\text{BAL} \supset \text{OK}$

Individual	APP	RATING	INC	BAL	OK
1	1	0	0	1	0
2	0	0	1	0	0
3	1	1	0	1	1
4	0	1	1	1	1
5	0	1	1	0	0
6	1	1	1	0	1
7	1	1	1	1	1
8	1	0	1	0	0
9	1	1	0	0	0

Table 17.1

Bank Data

17.5.1 Learning Propositional Calculus Rules (5/9)

- Rule $\text{BAL} \supset \text{OK}$ covers the positive instances 3,4, and 7, but also covers the negative instance 1.
  - So, select another atom to make this rule more specific.
- We have already decided that the first component in the antecedent is BAL, so we have to consider it.

$$r_{APP} = 2/3 = 0.667$$
$$r_{RATING} = 3/3 = 1.0$$
$$r_{INC} = 2/2 = 1.0$$

We select RATING because r_{RATING} is based on a larger sample.

$$\text{BAL} \wedge \text{RATING} \supset \text{OK}$$

17.5.1 Learning Propositional Calculus Rules (6/9)

We need more rules which cover positive instance 6.
To learn the next rule, eliminate from the table all of the positive instances already covered by the first rule.

Individual	APP	RATING	INC	BAL	OK
1	1	0	0	1	0
2	0	0	1	0	0
5	0	1	1	0	0
6	1	1	1	0	1
8	1	0	1	0	0
9	1	1	0	0	0

Table 17.2

Reduced Data

17.5.1 Learning Propositional Calculus Rules (7/9)

- Begin the process all over again with reduced table
  - Start with the rule $T \supset OK$.

$r_{APP} = 1/4 = 0.25$
$r_{RATING} = 0/3 = 0.0$
$r_{INC} = 1/4 = 0.25$
$r_{BAL} = 0/1 = 0.0$

APP=INC=0.25, arbitrarily select APP.

$APP \supset OK$

This rule covers negative instances 1, 8, and 9
➔ we need another atom to the antecedent.

$r_{RATING} = 1/2 = 0.5$
$r_{INC} = 1/2 = 0.5$
$r_{BAL} = 0/1 = 0.0$

Select RATING, and we get

$APP \wedge RATING \supset OK$

This rule covers negative example 9.

  - Finally we get $APP \wedge RATING \wedge INC \supset OK$ which covers only positive instances with first rule, so we are finished.

17.5.1 Learning Propositional Calculus Rules (8/9)

- Pseudocode of this rule learning process.
  - Generic Separate-and-conquer algorithm (GSCA)

Ξ is the initial training set of instances of binary-valued features each labeled by the value of an atom, γ

π is a set of rules to be learned

ρ is one of the rules; it has γ as its consequent and (the conjunction of atoms) Γ as its antecedent

α is an atom drawn from one of the features in Ξ

17.5.1 Learning Propositional Calculus Rules (9/9)

GSCA

1. Initialize $\Xi_{cur} \leftarrow \Xi$.
2. Initialize $\pi \leftarrow$ empty set of rules.
3. **repeat** The outer loop adds rules until π covers all (or most) of the positive instances.
4. Initialize $\Gamma \leftarrow \mathrm{T}$.
5. Initialize $\rho \leftarrow \Gamma \supset \gamma$.
6. **repeat** The inner loop adds atoms to Γ until ρ covers only (or mainly) positive instances.
7. Select an atom α to add to Γ. This is a nondeterministic choice point that can be used for backtracking.
8. $\Gamma \leftarrow \Gamma \wedge \alpha$.
9. **until** ρ covers only (or mainly) positive instances in Ξ_{cur}.
10. $\pi \leftarrow \pi, \rho$. We add the rule ρ to the set of rules.
11. $\Xi_{cur} \leftarrow \Xi_{cur} -$ (the positive instances in Ξ_{cur} covered by π).
12. **until** π covers all (or most) of the positive instance in Ξ.

17.5.2 Learning First-Order Logic Rules (1/10)

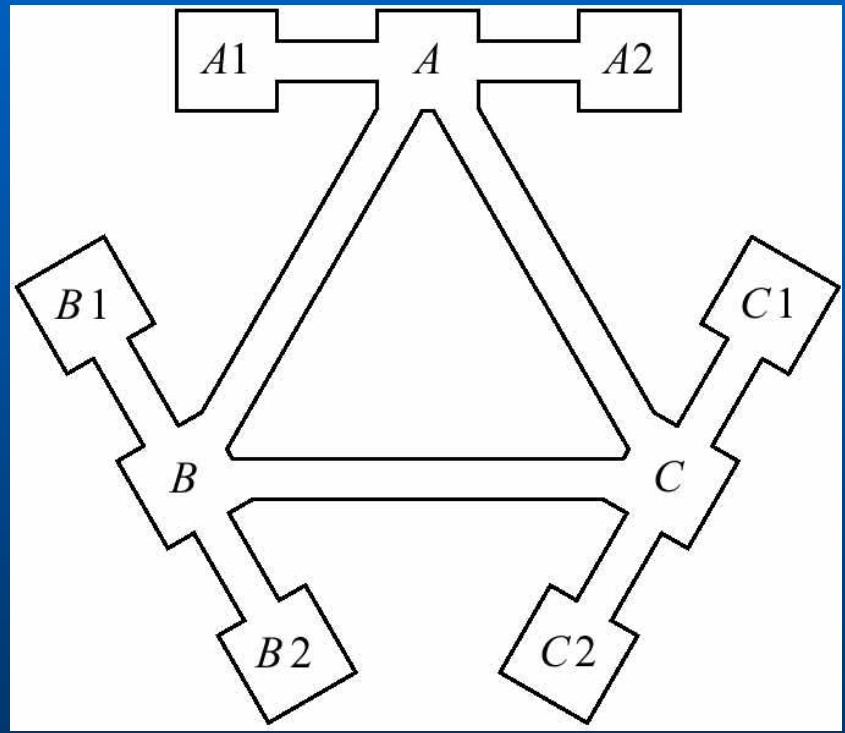
- Inductive logic programming (ILP)
  - Concentrate on methods for inductive learning of Horn clauses in first order predicate calculus (FOPC) and thus PROLOG program.
  - FOIL [Quinlan, 1990]
- The Objective of ILP
  - To learn a program, π,consisting of Horn clauses, ρ,each of which is of the form $\rho :- \alpha_1, \alpha_2, \cdots, \alpha_i$ ere, the α_i are atomic formulas that unify with ground atomic facts.
    - Ξ^+ : π should evaluate to *True* when its variables are bound to some set of values known to be in the relation we are trying to learn (positive instance: *training* set).
    - Ξ^- : π should evaluate to *False* when its variables are bound to some set of values known not to be in the relation (negative instance).

17.5.2 Learning First-Order Logic Rules (2/10)

- We want π to cover the positives instances and not cover negative ones.
- Background knowledge
  - The ground atomic facts with which the α are to unify.
  - They are given-as either subsidiary PROLOG programs, which can be run and evaluated, or explicitly in the form of a list of facts.
- Example: A delivery robot navigating around in a building finds through experience, that it is easy to go between certain pairs of locations and not so easy to go between certain other pairs.

17.5.2 Learning First-Order Logic Rules (3/10)

- A, B, C: *junctions*
- All of the other locations: *shops*
- Junction(x)
  - Whether junction or not.
- Shop(x,y)
  - Whether shop or not which is connected to junction x.



17.5.2 Learning First-Order Logic Rules (4/10)

- We want a learning program to learn a program, Easy(x,y) that covers the positive instances in Ξ but not the negative ones.
- Easy(x,y) can use the background subexpressions Junction(x) and Shop(x,y).
- Training set

$$\text{Positive instances of Easy} : \Xi^{+}$$
$$\{<A,B>,<A,C>,<B,C>,<B,A>,<C,A>,<C,B>,$$
$$<A,A1>,<A,A2>,<A1,A>,<A2,A>,<B,B1>,<B,B2>,$$
$$<B,B1>,<B2,B>,<C,C1>,<C,C2>,<C1,C>,<C2,C>\}$$

$$\text{Negative instances of Easy} : \Xi^{-}$$
$$\{<A,B1>,<A,B2>,<A,C1>,<A,C2>,<B,C1>,<B,C2>,$$
$$<B,A1>,<B,A2>,<C,A1>,<C,A2>,<C,B1>,<C,B2>,$$
$$<B1,A>,<B2,A>,<C1,A>,<C2,A>,<C1,B>,<C2,B>,$$
$$<A1,B>,<A2,B>,<A1,C>,<A2,C>,<B1,C>,<B2,C>\}$$

17.5.2 Learning First-Order Logic Rules (5/10)

- For all of the locations named in Ξ , only the following pairs give a value *True* for Shop:

$$\{< A1, A >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$$

- The following PROLOG program covers all of the positive instances of the training set and none of the negative ones

```
Easy(x, y) :- Junction(x), Junction(y)
        :- Shop(x, y)
        :- Shop(y, x)
```

17.5.2 Learning First-Order Logic Rules (6/10)

- Learning process: generalized separate and conquer algorithm (GSCA)
  - Start with a program having a single rule with no body
  - Add literals to the body until the rule covers only (or mainly) positive instances
  - Add rules in the same way until the program covers all (or most) and only (with few exceptions) positive instances.

17.5.2 Learning First-Order Logic Rules (7/10)

- Practical ILP systems restrict the literals in various ways.
- Typical allowed additions are
  - Literals used in the background knowledge
  - Literals whose arguments are a subset of those in the head of the clause.
  - Literals that introduce a new distinct variable different from those in the head of the clause.
  - A literal that equates a variable in the head of the clause with another such variable or with a term mentioned in the background knowledge.
  - A literal that is the same (except for its arguments) as that in the head of the clause.

17.5.2 Learning First-Order Logic Rules (8/10)

- The literals that we might consider adding to a clause are

  Junction(x), Junction(y), Junction(z)
  Shop(x,y), Shop(y,x), Shop(x,z)
  Shop(z,y), (x=y)

- ILP version of GSCA
  - First, initialize first clause as Easy(x, y) :-
  - Add Junction(x), so Easy(x, y) :- Junction(x) covers the following instances

$$\{<A,B>,<A,C>,<B,C>,<B,A>,<C,A>,<C,B>,$$
$$<A,A1>,<A,A2>,<B,B1>,<B,B2>,<C,C1>,<C,C2>\}$$

← Positive instances

$$\{<A,B1>,<A,B2>,<A,C1>,<A,C2>,<C,A1>,<C,A2>,$$
$$<C,B1>,<C,B2>,<B,A1>,<B,A2>,<B,C1>,<B,C2>\}$$

← Negative instances

  - Include more literal 'Junction(y)' ➔ Easy(x, y) :- Junction(x), Junction(y)

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

17.5.2 Learning First-Order Logic Rules (9/10)

- But program π does not cover the following positive instances.

$$\{<A,A1>,<A,A2>,<A1,A>,<A2,A>,<B,B1>,<B,B2>,$$
$$<B1,B>,<B2,B>,<C,C1>,<C,C2>,<C1,C>,<C2,C>\}$$

- Remove the positive instance covered by Easy(x,y):- Junction(x), Junction(y) from Ξ to form the Ξ_{CUR} to be used in next pass through inner loop.
  - Ξ_{CUR} : all negative instance in Ξ + the positive instance that are not covered yet.
- Inner loop create another initial clause "Easy(x,y) :-"
  - Add literal Shop(x,y) : Easy(x,y) :- Shop(x,y) ➔ cover no negative instances, so we are finished with another pass through inner loop.
  - Covered positive instance by this rule (remove this from Ξ_{CUR})

$$\{<A1,A>,<A2,A>,<B1,B>,<B2,B>,<C1,C>,<C2,C>\}$$

17.5.2 Learning First-Order Logic Rules (10/10)

- Now we have Easy(x,y) :- Junction(x), Junction(y)
  :- Shop(x, y)
- To cover following instance

$$\{< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$$

- Add Shop(y, x)
- Then we have Easy(x,y) :- Junction(x), Junction(y)
  :- Shop(x, y)
  :- Shop(y, x)
- This cover only positive instances.

17.5.3 Explanation-Based Generalization (1/2)

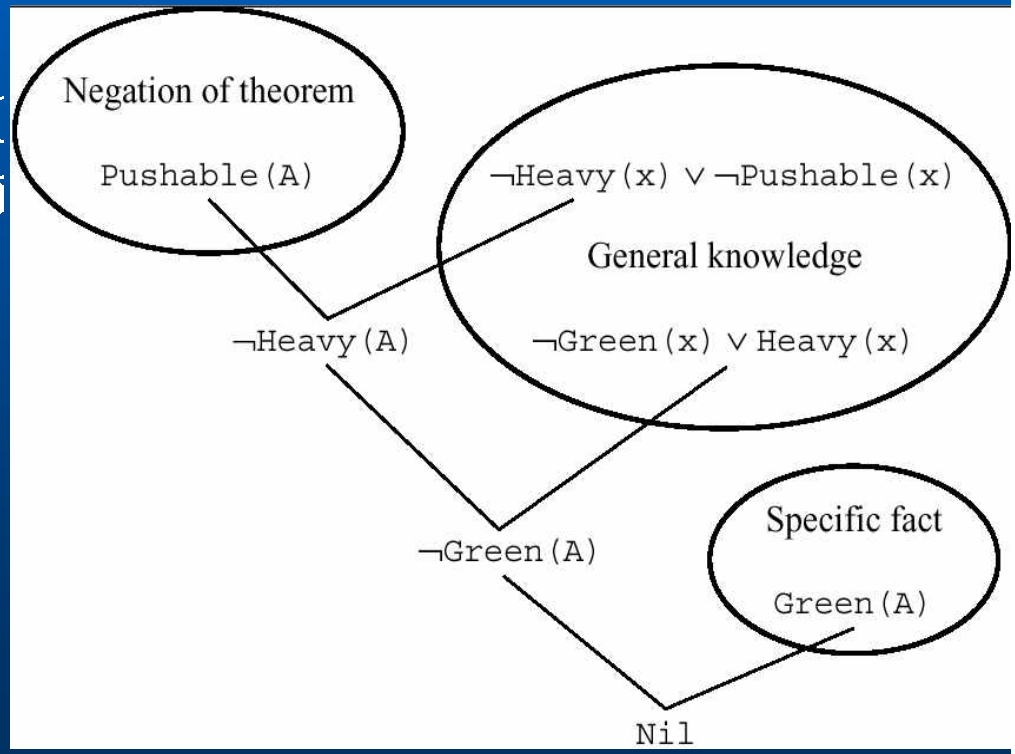
- Example: "Block world"
  - General knowledge of the "Block world".
    - Rules

      Green(x) ∧ Heavy(x) → ¬Pushable(x)

      Heavy(x) → ¬Pushable(x)
    - Fact

      Green(A)
    - We want to proof

      " ¬Pushable(A) "
    - Proof is very simple ➔



17.5.3 Explanation-Based Generalization (2/2)

- Explanation: Set of facts used in the proof
  - Ex) explanation for " ¬Pushable(A) " is " Green(A) "
  - From this explanation, we can make "Green(Rusha)"
    - Replacing constant 'A' by variable 'x', then, we have "Green(x)"
    - Then we can proof " Pushable(A)", as like the case of "¬Pushable(x)"
    - ➔Explanation Based Generalization
- Explanation-based generalization (EBG): Generalizing the explanation by replacing constant by variable
  - More rules might slow down the reasoning process, so EBG must be used with care-possibly by keeping information about the utility of the learned rules.

Additional Readings (1/4)

- [Levesque & Brachman 1987]
  - Balance between logical expression and logical inference
- [Ullman 1989]
  - DATALOG
- [Selman & Kautz 1991]
  - Approximate theory: Horn greatest-lower-bound, Horn least-upper-bound
- [Kautz, Kearns, & Selman 1993]
  - Characteristic model

Additional Readings (2/4)

- [Roussel 1975, Colmerauer 1973]
  - PROLOG interpreter
- [Warren, Pereira, & Pereira 1977]
  - Development of efficient interpreter
- [Davis 1980]
  - AO* algorithm searching AND/OR graphs
- [Selman & Levesque 1990]
  - Determination of minimum ATMS label: NP-complete problem

Additional Readings (3/4)

- [Kautz, Kearns & Selman 1993]
  - TMS calculation based on characteristic model
- [Doyle 1979, de Kleer 1986a, de Kleer 1986b, de Kleer 1986c, Forbus & de Kleer 1993, Shoham 1994]
  - Other results for TMS
- [Bobrow, Mittal & Stefik 1986], [Stefik 1995]
  - Construction of expert system

Additional Readings (4/4)

- [McDermott 19982]
  - Examples of expert systems
- [Leonard-Barton 1987]
  - History and usage of DEC's expert system
- [Kautz & Selman 1992], [Muggleton & Buntine 1988]
  - Predicate finding
- [Muggleton, King & Sternberg 1992]
  - Protein secondary structure prediction by GOLEM