Expert Systems & Knowledge Acquisition: Orientation

Chris Thornton -- January 11, 1989

-- Coverage ----------

The course will cover some peripheral topics in expert systems and look at the three main techniques for automatic knowledge acquisition.

-- Assessment ----------

One practical (worth 20%) and one exam (worth 80%).

-- Timetable ----------

Lectures: Tuesdays and Thursdays 2-00, F10.
Tutorials: Thursdays, 1-00, E6

wk	lecture 1	lecture 2	exercise due
1	orientation	popll (1) expressions & commands	
2	popll (2) procedures	mycin-like system in popll	
3	manual knowledge acquisition	information theory and the expert system	
4	intro. to knowledge acquisition	the classification algorithm	
5	implementation of classification, windowing	version spaces, candidate-elimintation algorithm	
6	focussing set practical-1	the aq algorithm	
7	the aq11 family	conceptual clustering	
8	unimem	ebl	submit practical-1
9	lex	the problem of abstract classes	

-- Topics ----------

es01
-- INTRODUCTION, Definitions of an Expert System, Distinctive features

es02
-- POP11 PROGRAMMING 1 (expressions and commands), Pop11 references, Statements, expressions and commands, Numerical expressions and simple commands, Functions, Variables, Assignments, Boolean expressions, Word expressions, List expressions, Single hats, Double hats, List functions, Complex commands, Conditionals, The repeat command, The quitloop command, The while and until commands, The [illegible]mmand, nextloop v. quitloop, Using loops to construct lists

es03
-- POP11 PROGRAMMING 2 (procedures), define, return, Recursion, Local variables, Applist, Maplist, Syssort, Matching, Restriction procedures, Restriction procedures can have side-effects, Integer restrictions, The matcher arrow, The database, Present, Foreach, Tracing, Showtree, Readline

es04
-- IMPLEMENTING A MYCIN-LIKE SYSTEM, Fuzzy set theory, Setting up the knowledge-base, Implementing the inference engine, Implementing certainty-factors, Forwards or backwards, Reading, Exercise

es05
-- MANUAL KNOWLEDGE ACQUISITION, Stages of KA (alternative version), Some rules of thumb in KA

es06
-- INFORMATION THEORY, Uncertainty reduction, Measuring uncertainty, The additive property of information, The link with successive halving, Entropy, Information and content, Information and knowledge

es07
-- INTRO. TO AUTOMATIC KNOWLEDGE ACQUISITION, Why not just use statistics?, Bayes theorem, Problems with Bayesian statistics, Machine Learning, Expertise as classification, Informal v. coupled representations, The neighbourhood representation scheme (NR)

es08
-- INDUCTION OF DECISION TREES, Classical concepts

es09
-- ID3, The information-theoretic heuristic, Minising the entropy of the distibution

es10
-- VERSION SPACES AND CANDIDATE ELIMINATION, The need for a richer description language, Generalised descriptions, Generality orderings, Version spaces, The Candidate-Elimination algorithm, Tree-based description languages

es11
-- FOCUSSING

es12
-- THE AQ ALGORITHM, The disjunctive-concept problem, Multiple-neighbourhood representations, Learning disjunctive concepts using Focussing/CE, Multiple convergence, Least disjunctions, The AQ algorithm

es13
-- DISCRIMINATION RULES

es14
-- CONCEPTUAL CLUSTERING

es15
-- UNIMEM

es16
-- EXPLANATION-BASED LEARNING, EBL in theory, EBL in practice

es17
-- THE LEX SYSTEM, Version space heuristics, Processing cycle in LEX, Problems with EBL

es18
-- THE PROBLEM OF ABSTRACT CLASSES

-- Bibliography ----------

Baierlein, R. (1971). ATOMS AND INFORMATION THEORY. San Francisco: W.H. Freeman.

Bar-Hillel, Y. & Carnap, R. (1953). Semantic information. In W. Jackson (Ed.), COMMUNICATION THEORY. London: Butterworths.

Bar-Hillel, Y. (1965). LANGUAGE AND INFORMATION. Reading, Mass.

Barr, A. & Feigenbaum, E.A. (1982). THE HANDBOOK OF ARTIFICIAL INTELLIGENCE: VOLUME 2. Los Altos: William Kaufmann.

Bratko, I. & Lavrac, N. (Eds.) (1987). PROGRESS IN MACHINE LEARNING. Wilmslow: Sigma Press.

Brillouin, L. (1962). SCIENCE AND INFORMATION THEORY. New York: Academic.

Bundy, A., Silver, B. & Plummer, D. (1985). An analytical comparison of some rule-learning programs. ARTIFICIAL INTELLIGENCE, 27, No. 2, 137-181.

Burton, M. & Shadbolt, N. (1987). POP-11: PROGRAMMING FOR ARTIFICIAL INTELLIGENCE. Addison-Wesley.

Charniak, E. & McDermott, D. (1985). INTRODUCTION TO ARTIFICIAL INTELLIGENCE. Addison-Wesley.

Dietterich, T.G., London, B., Clarkson, K. & Dromey, G. (1982). Learning and inductive inference. In P.R. Cohen & E.A. Feigenbaum (Eds.), THE HANDBOOK OF ARTIFICIAL INTELLIGENCE: VOL III. Los Altos: William Kaufmann.

Dretske, F.I. (1981). KNOWLEDGE AND THE FLOW OF INFORMATION. Oxford: Basil Blackwood.

Feigenbaum, E.A. & McCorduck, P. (1984). THE FIFTH GENERATION. London: Pan Books Ltd.

Forsyth, R. & Rada, R. (1986). MACHINE LEARNING: APPLICATIONS IN EXPERT SYSTEMS AND INFORMATION RETRIEVAL. Chichester: Ellis Horwood.

Gatlin, L.L. (1972). INFORMATION THEORY AND THE LIVING SYSTEM. London: Columbia University Press.

Gluck, M.A. & Corter, J.E. (1985). Information, uncertainty, and the utility of categories. PROCEEDINGS OF THE SEVENTH ANNUAL CONFERENCE OF THE COGNITIVE SCIENCE SOCIETY. Irvine, California: Lawrence Erlbaum Associates.

Hilgard, E.R. & Bower, G.H. (1975). THEORIES OF LEARNING. Englewood Cliffs, N.J.: Prentice-Hall.

Hunt, E.B., Marin, J. & Stone, P.J. (1966). EXPERIMENTS IN INDUCTION. New York: Academic Press.

Jackson, W. (Ed.) (1953). COMMUNICATION THEORY. London: Butterworths.

Jones, D.S. (1979). ELEMENTARY INFORMATION THEORY. Oxford: Clarendon Press.

Laventhol, J. (1987). PROGRAMMING IN POP-11. Oxford: Blackwell Scientific Publications.

Lebowitz, M. (1986). Concept learning in a rich input domain: generalization-based memory. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH: VOLUME II. Los Altos: Morgan Kaufmann.

Lebowitz, M. (1986). UNIMEM, a general learning system: an overview. PROCEEDINGS OF THE EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. MACHINE LEARNING, 2, No. 2, 103-138, Kluwer Academic Publishers.

MacKay, D.M. (1953). Generators of information. In W. Jackson (Ed.), COMMUNICATION THEORY. London: Butterworths.

[redacted]ay, D.M. (1969). INFORMATION, MECHANISM AND MEANING. London: MIT Press.

Michalski, R.S. & Stepp, R.E. (1983). Learning from observation: conceptual clustering. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH. Palo Alto: Tioga.

Michalski, R.S. (1986). Understanding the nature of learning: issues and research directions. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH: VOL II. Los Altos: Morgan Kaufmann.

Minsky, M. (1988). THE SOCIETY OF MIND. London: Pan Books.

Mitchell, T.M. (1977). Version spaces: a candidate elimination approach to rule learning. PROCEEDINGS OF THE FIFTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE. Cambridge, Mass.

Mitchell, T.M. (1982). Generalization as search. ARTIFICIAL INTELLIGENCE, 18, 203-226.

Mitchell, T.M., Utgoff, P.E. & Banerji, R. (1983). Learning by experimentation: acquiring and modifying problem-solving [redacted]istics. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell [redacted]s.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH. Palo Alto: Tioga.

Mitchell, T.M., Keller, R.M. & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: a unifying view. MACHINE LEARNING, 1, No. 1, 47-80, Boston: Kluwer Academic.

Murray, K.S. (1987). Multiple convergence: an approach to disjunctive concept acquisition. PROCEEDINGS OF THE TENTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE. Los Altos, California: Morgan Kaufmann.

Pountain, D. (1988). POP goes the macintosh. BYTE, 285-292.

Quinlan, J.R. (1979). Discovering rules by induction from collections of examples. In D. Michie (Ed.), EXPERT SYSTEMS IN THE MICRO-ELECTRONIC AGE. Edinburgh: Edinburgh University Press.

Quinlan, J.R. (1983). Learning efficient classification procedures and their application to chess end games. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH. Palo Alto: Tioga.

Ramsay, A. & Barrett, R. (1987). AI IN PRACTICE: EXAMPLES IN POP-11. Chichester: Ellis Horwood.

Shannon, C.E. & Weaver, W. (1949). THE MATHEMATICAL THEORY OF INFORMATION. Urbana: University of Illinois Press.

Shortliffe, E.H. (1976). COMPUTER-BASED MEDICAL CONSULTATIONS: MYCIN. New York: American Elsevier.

Simon, H.A. (1983). Why should machines learn?. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH. Palo Alto: Tioga.

Smith, E.E. & Medin, D. (1981). CATEGORIES AND CONCEPTS. Cambridge, Massachusetts: Harvard University Press.

Thornton, C.J. (1988). Links between content and information-content. PROCEEDINGS OF THE EIGHTH EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE (Munich, 1-5 August). Pitman (also available as CSRP 109, School of Cognitive Sciences, University of Sussex, 1988).

Utgoff, P.E. (1986). Shift of bias for inductive concept learning. In R.S. Michalski, J.G. Carbonnell & T.M. Mitchell (Eds.), MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH: VOL II. Los Altos: Morgan Kaufmann.

Watanabe, S. (1969). KNOWING AND GUESSING: A QUANTITATIVE STUDY OF INFERENCE AND INFORMATION. New York: John Wiley & Sons.

Winston, P.H. (1975). Learning structural descriptions from examples. In P.H. Winston (Ed.), THE PSYCHOLOGY OF COMPUTER VISION. Mcgraw-Hill.

Winston, P.H. (1984). ARTIFICIAL INTELLIGENCE (2nd Edition). Reading, Massachusetts: Addison-Wesley.

Yazdani, M. & Narayanan, A. (1984). ARTIFICIAL INTELLIGENCE: HUMAN EFFECTS. Chichester: Ellis Horwood.

Young, R.M., Plotkin, G.D. & Linz, R.F. (1977). Analysis of an extended concept-learning task. In R. Eddy (Ed.), PROCEEDINGS OF THE FIFTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE. Cambridge, MA.

Zadeh, L. (1965). Fuzzy sets. INFORMATION AND CONTROL, 8, 338-353.

Expert Systems & Knowledge Acquisition

Chris Thornton -- January 16, 1989

-- IMPLEMENTING A MYCIN-LIKE SYSTEM ----------

Mycin-like expert systems
  ordinary backwards-reasoning
  + query-the-user
  + explanations
  + canned-text diagnoses
  + certainty factors

- Mycin is a backwards-reasoning, goal-driven expert system which generates diagnoses (and advice on treatment) for some infectious diseases. Rules have certainty factors associated with them.

The lecture will look at a simple, Pop11 implementation.

-- Certainty-factors: some background ----------

A major problem in implementing a Mycin-like system is working out how to cope with certainty factors. Fuzzy set theory (Zadeh 1965) states that, for ordinary conjunctive inference rules the certainty of a given conclusion is just the *minimum* of the certainties of the "premises"

- This seems to make reasonably good intuitive sense. If our certainty that the pressure is high is 0.6, and our certainty that the clouds are alto-stratus is 0.5, then it seems quite acceptable to say that our certainty that the pressure is high *and* that the clouds are alto-stratus is the lesser of the two figures, namely 0.5.

How can we compute overall certainty of a goal given N different ways of satisfying

- Shortliffe, in his work on the MYCIN system (Shortliffe, 1976), proposed the following method. If H is a goal which has previously been shown to have certainty C1, then, if we find a way of showing that it also has a certainty C2 (using a different proof tree), then the overall certainty of H is

C1 + C2 - (C1 * C2) if C1 and C2 are both positive

C1 + C2 + (C1 * C2) if C1 and C2 are both negative

(C1 + C2) / (1 - min(abs(C1), abs(C2))) if C1 and C2 have different signs

- These rules seem to provide the sort of effect we are looking for. For example in the case where we have derived two different levels of (positive) certainty for a given conclusion we would like to somehow add them together so as to get an increased level overall. But we cannot just add them together. In general this will give us a probability that exceeds 1. Shortliffe's first rule above allows them to be added together in a way which will increase the level of certainty but not cause it to exceed 1. The other rules take care of the other cases in a similar way.

- Unfortunately, it is not clear whether Shortliffe's rules can be given any principled foundation. As Forsyth notes (Forsyth, The Architecture of Expert Systems 1984, p. 56) "Shortliffe does attempt a theoretical justification for these methods, but in my view the rationale is somewhat shakey. The important point is that this set of techniques has served well in a significant program, MYCIN, and its successors."

Setting up the knowledge-base ----------

Use Pop11 database. Each entry is a rule associated with a certainty value. Rules taken from (Ramsay Barrett 1987, chapter 3).

```
[ [[season winter] 1 [month december]]
  [[season winter] 1 [month january]]
  [[season winter] 0.9 [month february]]
  [[season spring] 0.7 [month march]]
  [[season spring] 0.9 [month april]]
  [[season spring] 0.6 [month may]]
  [[season summer] 0.8 [month june]]
  [[season summer] 1 [month july]]
  [[season summer] 1 [month august]]
  [[season autumn] 0.8 [month september]]
  [[season autumn] 0.7 [month october]]
  [[season autumn] 0.6 [month november]]
  [[pressure high] 0.6 [weather_yesterday good]
      [stability_of_the_weather stable]]
  [[pressure high] 0.9 [clouds high]]
  [[pressure high] 0.9 [clouds none]]
  [[temp cold] 0.9 [season winter] [pressure high]]
  [[temp cold] 0.6 [season summer] [pressure low]]
  [[wind none] 0.3 [pressure high]]
  [[wind east] 0.3 [pressure high]]
  [[wind west] 0.6 [pressure low]]
  [[temp warm] 0.8 [wind south]]
  [[temp cold] 0.9 [wind east] [clouds none]
      [season winter]]
  [[temp warm]
   0.9 [wind none] [pressure high] [season summer]]
  [[rain yes] 0.4 [whereabouts west]]
  [[temp cold] 0.4 [whereabouts north]]
  [[rain no] 0.7 [whereabouts east]]
  [[rain yes] 0.3 [season spring] [clouds low]]
  [[rain yes] 0.3 [season spring] [clouds high]]
  [[temp warm] 0.7 [season summer]]
  [[rain yes] 0.2 [season summer] [temp warm]]
  [[rain yes] 0.6 [pressure low]]
  [[rain yes] 0.6 [wind west]]
  [[rain yes] 0.8 [clouds low]]
  [[temp cold] 0.8 [season autumn] [clouds none]]
  [[temp cold] 0.7 [season winter]]
] -> database;
```

- The second-to-last rule here captures the proposition that the probability of it being cold, given the fact that it is autumn and the fact that there are no clouds is quite high (0.8).

- Many other propositions are captured by this rulebase; e.g. the fact that it rains more in the west, the fact that it is colder in the north and the fact that high pressure and no wind are usually associated with warm weather.

-- Implementing the inference engine ----------

NB. rules are of the form

  [<goal> <certainty> <subgoal1> <subgoal2> .]

rather than

  [<goal> <subgoal1> <subgoal2> .]

- must implement backwards-reasoning procedure so as to take account of the fact that the rules are written left to right.

First, implement an inferencing procedure which ignores certainty values.

- Given some goal this procedure should check the rulebase to see if there are any rules (or facts) which might satisfy that goal, and if not, asks the user whether the goal can be assumed to be "given by assumption". This involves calling the database function "present" to check whether the desired entry is present and then, if it is not, calling the interactive function "yesno".

```
define list_to_string(l);
    l >< ''
enddefine;

define satisfy(goals);
    vars goal other_goals subgoals tree other_tree;
    if not(goals matches [?goal ??other_goals]) then /* goals = [] */
        return([])
    elseif present([^goal ==]) then
        foreach [^goal = ??subgoals] do
            if (satisfy(subgoals) ->> tree)
            and (satisfy(other_goals) ->> other_tree) then
                return([[^(list_to_string(goal)) ^^tree] ^^other_tree])
            endif;
        endforeach;
    elseif yesno([is ^(goal(2)) the value of ^(goal(1))])
    and (satisfy(other_goals) ->> other_tree) then
        return([[^(list_to_string(goal)) USER_RESPONSE] ^^other_tree])
    endif;
    return(false);
enddefine;
```

- Note that we have introduced a call on the function "list_to_string" into the list expressions which are returned by the function. We have to do this for the same reason we had to do it in Chapter 2: goals in this scenario are lists not words and this means that if we want to get "showtree" to print out our search spaces properly, we have to convert all goals (i.e. all nodes) to words inside the inference engine.

- Note also that in the case where the rulebase contains no rule which will enable the satisfaction of the given goal, the function "yesno" is called. In effect, the function asks the user whether the goal can be assumed to be satisfied. If the user responds with a "yes", the "yesno" function returns <true> and the overall result is a list with the word "USER_RESPONSE" substituted in for the solution tree.

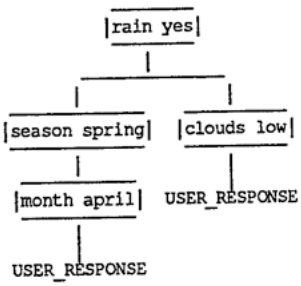
- Testing the inference engine on the goal [rain yes] produces the following behaviour. (Words typed after the "?" prompt are user input.)

```
vars tree;
satisfy([[rain yes]]) -> tree;

>satisfy [[rain yes]]
!>satisfy [[whereabouts west]]
** [is west the value of whereabouts]
? no
```

```
!<satisfy <false>
!>satisfy [[season spring] [clouds low]]
!!>satisfy [[month march]]
** [is march the value of month]
? no
!!<satisfy <false>
!!>satisfy [[month april]]
** [is april the value of month]
? yes
!!!>satisfy []
!!!<satisfy []
!!<satisfy [[month april USER_RESPONSE]]
!!>satisfy [[clouds low]]
** [is low the value of clouds]
? yes
!!!>satisfy []
!!!<satisfy []
!!<satisfy [[clouds low USER_RESPONSE]]
!<satisfy [[season spring [month april
   USER_RESPONSE]] [clouds low USER_RESPONSE]]
!>satisfy []
!<satisfy []
<satisfy [[rain yes [season spring
   [month april USER_RESPONSE]]
   [clouds low USER_RESPONSE]]]
```

- We assigned the result of this call into the variable "tree". Thus we can get the tree printed out using the "showtree" command,

```
showtree(hd(tree));
```



- In a liberal reading of this solution tree we might say that the inference engine showed that it is raining by showing that it is springtime and the clouds are low. It showed that it is springtime by showing that it is april. And it showed that it is april and the clouds are low by asking the user.

-- Implementing certainty-factors ----------

- We can now concentrate on the taking care of the certainty factors. This involves arranging things such that the inference engine can combine certainty values together in a manner which reflects the dictates of Fuzzy set theory (and the implementation of the MYCIN program).

- The approach here involves changing the function such that instead of returning the solution tree (proof tree) for the input goals (conclusions), it returns a certainty value for them. In the case where the function obtains one certainty value *c1* for the subgoals of the current goal, and another *c2* for all the remaining goals, it should simply multiply c1 by the certainty factor for the search rule in question and then return the minimum of the result and *c2*. This will lead to a correct application of the Fuzzy set theory formulae.

- Obviously, in the case where a goal is satisfied by querying the user, the goal can be assumed to have been satisfied with perfect certainty; i.e. with a certainty value of 1. The overall certainty value to be returned is therefore just the *c2* value, i.e. the certainty for the remaining goals.

```
define certainty_of(goals);
   vars goal subgoals other_goals tree other_treef c c1 c2 certainties;
   if not(goals matches [?goal ??other_goals]) then /* gaols = [] */
      return(1)
   elseif present([^goal ==]) then
      foreach [^goal ?c ??subgoals] do
         if (certainty_of(subgoals) ->> c1)
         and (certainty_of(other_goals) ->> c2) then
            return(min(c1 * c, c2))
         endif
      endforeach;
   elseif yesno([is ^(goal(2)) the value of ^(goal(1))])
   and (certainty_of(other_goals) ->> c1) then
      return(c1)
   endif;
   return(false);
enddefine;
```

Testing this function on the same goal leads to the following behaviour. Note that the user responses are identical.

```
certainty_of([[rain yes]]) ==>

>certainty_of [[rain yes]]
!>certainty_of [[whereabouts west]]
** [is west the value of whereabouts]
? no
!<certainty_of <false>
!>certainty_of [[season spring] [clouds low]]
!!>certainty_of [[month march]]
** [is march the value of month]
? no
!!<certainty_of <false>
!!>certainty_of [[month april]]
** [is april the value of month]
? yes
!!!>certainty_of []
!!!<certainty_of 1
!!<certainty_of 1
!!>certainty_of [[clouds low]]
** [is low the value of clouds]
? yes
!!!>certainty_of []
!!!<certainty_of 1
!!<certainty_of 1
!<certainty_of 0.9
!>certainty_of []
!<certainty_of 1
<certainty_of 0.27
** 0.27
```

- Thus, the [rain yes] conclusion is derived with a certainty of 0.27, i.e. with a fairly low degree of certainty given the evidence provided.

-- Forwards or backwards ----------

- The search strategy employed by MYCIN (and the function constructed above) employs backwards-reasoning; i.e. reasoning backwards from goals to known facts. This is inevitable given the assumption that all facts are provided directly by the user in query responses. If there are no facts in the rulebase, then there is no basis on which a forwards-chaining search can be initiated. However, although the interactive behaviour of MYCIN is incompatible with forwards-chaining search, the probabilistic aspect of its behaviour is not. It is quite possible for a forwards-chaining inference engine to implement the Fuzzy set rules in exactly the same way as the backwards-reasoning function.

There is an interesting control strategy which combines forwards-chaining and backwards-reasoning. A system using this strategy might initially accept some basic facts (e.g. symptoms). It would then work bottom-up until some hypotheses have been developed. From these hypotheses it could work top-down so as to generate a new set of primitive goals which can be presented as queries to the patient. The patient's answers constitute satisfaction of the primitive goals; thus another phase of bottom-up search can begin. Eventually, this *bi-directional* strategy will tend to isolate a single high-level conclusion representing a diagnosis.

This strategy is adopted by Intemist.

-- Reading ----------

The main reference for MYCIN is (Shortliffe, 1976). Chapter 3 or Ramsay and Barrett (1987) provides a discussion of, and a POP-11 implementation for a MYCIN-like system. Charniak and McDermott (1985, chapter 8) provide an excellent discussion of ways of dealing with uncertainty in expert systems. Volume 2 of the AI handbook (Barr Feigenbaum 1982) covers most of the more important expert systems. Forsyth (Forsyth, The Architecture of Expert Systems 1984, pp. 51-62) discusses the processing of certainty factors.

Expert Systems & Knowledge Acquisition: 5

Chris Thornton – January 20, 1989

-- MANUAL KNOWLEDGE ACQUISITION ----------

Knowledge acquisition is difficult

- seldom described in any detail (cf. Feigenbaum & McCorduck 1984).

This is the "bottleneck" in expert systems development

== Definition ----------

Knowledge acquisition is the *transfer* and *transformation* of problem-solving expertise from some "knowledge source" to a program.

Knowledge acquisition is not the same thing as asking an expert to explain what he/she knows.

Most expert knowledge is

- unconscious
- heuristic
- imprecise

Knowledge acquisition is not the same as "finding the rules"

== Typical MYCIN rule ----------

IF 1) the stain of the organism is gramneg, and
2) the morphology of the organism is rod, and
3) the aerobicity of the organism is aerobic

THEN: there is strongly suggestive evidence (0.8) that the class of the organism is enterobacteriaceae

1980 version of MYCIN has:

13 context types each with 13 properties
450 rules classified into types
200 clinical parameters (attributes) classified into categories, each with 13 properties

== The 3 stages of knowledge acquisition (M. Bramer) ------

1. Determine structure of the domain

2. Determine first working system

3. testing, debugging and iterative refinement of system

NB. stages are not necessarily separate

== Stage 1 ----------

1. Find relevant entities
(e.g. patients, cultures, organisms, drugs)
& relations amongst them
(e.g. each culture has 1 or more associated organisms)

2. Find the attributes of each entity
(e.g. name, sex, age, allergies, diagnosis - for patient
site - for cultures
identity, aerobicity - for organisms)
& their possible values

3. Break the reasoning down into stages, if possible
(e.g. find whether infection is "significant", identify infecting organism, select appropriate treatments, select best treatment for patient)

== Problems ----------

- unenthusiastic expert

- inaccessible expert

- lack of communication between knowledge engineer and expert

- lack of understanding of field by knowledge engineer

- the "inarticulate" expert

- expert can be over-influenced by initial choice of representation

== Stage 2 - determine domain rules ----------

Rules need a structure to fit into

- the more accurate the better

BUT NB. stage 2 may reveal that the domain structure needs refining

Eliciting knowledge

- interview

- observational technique

== Using interviews ----------

Experts are usually bad at explaining, rather than doing

Ask specific questions to elicit specific information

Avoid
- general theorising by expert
- vague, unstructured questions

But also avoid
- "overstructured" interview, "imposing" the initial view of the domain structure on the expert

== Psychological evidence ----------

Psychological studies of interviewing show

- "uncued recall" is particularly hard - giving cues helps, but

- "recognition" is much more accurate and complete; therefore ask "recognition-style" questions if possible

- giving action required when specified conditions hold is much easier than vice versa

== Observation ----------

- recording expert at work (may take considerable time)

- recording of protocols

- "critical incident analysis" (specific previous incidents)

- setting artificial problems (working through typical cases, questioning about reasoning and proposed actions)

Other techniques

- Repertory grid analysis
- Comparative case analysis

== Estimating probabilities ----------

People are very bad at estimating probabilities

Indirect methods are better (e.g. classifying on 5-point scale annotated with comments)

"Delusion of accuracy" problem (e.g. 100-point scale)

Using probabilities at all is suspect

== Stage 3 - testing, debuggging & iterative refinement ---

Involve expert in criticism/debugging, as aid to further knowledge acquisition and motivation

Importance of

- building and testing initial system *early*

- good user interface in *early* versions

== Automatic aids to refinement ----------

- explanation facility

- consistency checking

- automatic checking of pre-stored cases

- inductive inference

etc.

-- Stages of KA (altemative version) ----------

(Feigenbaum & McCorduck 1984)

1. persuade human expert to agree to devote the considerable time it will take to "have his mind mined"

2. Immerse self in field of expertise (read college textbooks, articles and other background material); pick up jargon etc.

3. At first interview ask expert to describe what he thinks he does; also - ask him how he thinks he solves problems. (Urge expert to choose a fairly difficult problem to examine. Should take a few hours to solve.)

4. Bring information back to other members of the team. Allow knowledge engineer to decide which problem solving framework to use.

5. Get initial system up and running. Use it to get expert interested in project.

6. Record expert's reactions to behaviours of prototype system.

7. Focus on mistakes. Get expert to walk through problem verbalising each step. Check whether steps are justified by data which is actually attended to

8. Construct second version of system.

"One of the difficulties of writing knowledge-based programs is that at least two parties are constantly shifting their points of view: the domain expert and the knowledge engineer. As the knowledge in the program accumulates ahd the problem becomes better clearer, the knowledge engineer may find better ways to represent and process the knowledge. teh resulting behaviour of the program may inspire the expert to tshift his view of the problem, creating for the knowledge engineer further problems to be solved. Development of expert programs involves a process of finding a workable relationship between experts and programmers and slowly evolving a program structure that will work." (H. Penny Nii)

-- Some rules of thumb in KA ----------

(Feigenbaum McCorduck 1984, pp. 111-112)

You can't be your own expert.

From the beginning the knowledge engineer must count on throwing efforts away.

The problem must be well-chosen. AI ... isn't ready to take on every problem the world has to offer. Expert systems work best when the problem is well bounded

You want to do any serious application you need to meet the expert more than halfway; if he's had no exposure to computing your job will be that much harder.

If none of the tools you normally use works, build a new one.

Dealing with anything by facts implies uncertainty. Heuristic knowledge is not hard and fast and cannot be treated as factuual. A weighting procedure has to be built into the expert system to allow for expressions such as "I strongly believe that . . ." or "The evidence suggests that

A high-performance program, or a program that will eventually be taken over by the expert for his own use, must have very easy ways of allowing the knowledge to be modified so that new information can be added and out-of-date information deleted.

== KA - the state of the art ----------

1. Building expert systems is an experimental activity

2. Knowledge acquisition is a craft, not an exact science

- no established or general methodology

- little detailed information in the literature

3. Need to elicit "domain structure" as well as heuristic rules

4. No clear division between stages - don't compartmentalise

5. Don't trust what experts say - observe in action if possible

6. Start building system early (to put initial ideas of structure and of rules to the test and to aid motivation of expert)

7. Crucial importance of iteration (involving expert)

- be prepared to rewrite original system several times

== KA - issues ----------

How can a knowledge acquisition method be chosen?

How can knowledge from different experts be combined?

Do shells solve the problem?

== KA - where next? ----------

Need for more research into knowledge acquisition as an activity (including comparative studies). Insights into selection of methods may be gained from:

- systems analysis

- cognitive psychology

- questionaire design

- etc.

Need for a wide repertoire of shells & selection criteria

Automatic induction of rules from examples?

== Tutorial exercise for Thursday, 26 of January ----------

The tutorial will involve a knowledge acquisition "role-play".

One person in the group will take the role of expert. The others will attempt to encourage the expert to articular his/her knowledge and then to re-express it in the form of a rulebase; i.e. they will act as knowledge engineers. The main aim will to see how robust the guide-lines suggested above actually are.

The "expert" should have detailed knowledge of some fairly complex mechanism (e.g. a bicycle, computer, sewing machine etc.). *Everyone* should come to the tutorial prepared to take on this role. This may involve some preparatory research.

Expert Systems & Knowledge Acquisition: 6

Chris Thornton – January 20, 1989 IN THE SYLLABUS.

-- INFORMATION THEORY ----------

Expert systems work by exchanging *information* with the user. During the interaction information is provided by the user. At the end of the interaction, specialised information (e.g. a diagnosis) is provided by the system.

Is there an information theoretic story for this exchange?

-- Uncertainty reduction ----------

Data originating from some "source" can provide differing amounts of information to some "receiver".

How can this information be quantified?

A basic notion in information theory is that information can be defined as *uncertainty reduction*.

- Seems to make sense. Assume the receiver is some mechanism capable of experiencing uncertainty with respect to some particular outcome.

The more the data (henceforth "the message") reduces the receiver's uncertainty w.r.t. the outcome, the more information it must contain.

Or must it ...?

There is a huge, ongoing debate about this.

-- Measuring uncertainty ----------

If information is to be measured in terms of uncertainty reduction, we need some way of *quantifying* uncertainty reduction.

Assume the source is some process or mechanism capable of producing any one of N different messages (i.e. outcomes) and that the a priori probability of it producing a particular message is always 1/N.

Eg. a rolled dice is capable of coming to rest on any one of its six faces and the a priori probability of it coming to rest on any one is just 1/6 (0.166).

- the a priori probability of randomly picking a specific card from a full pack is 1/52.

Assume the receiver is some mechanism capable of seeing which card is picked randomly from a selection of N playing cards.

What is the receiver's uncertainty with respect to the outcome?

It is obviously related to the size of N. But how?

Information theory says the convenient way of measuring the uncertainty (and therefore the information) is

$I(s) = -\log_2 1/N$ Inverse of event ($\log_2$)

The information in the message is related to minus the log of the a priori probability of the message.

```
-log2(0.1) ==>
** 3.32193

-log2(0.25) ==>
** 2.0

-log2(0.8) ==>
** 0.321928

-log2(0.99999)==>
** 0.000014
```

-- The additive property of information ----------

Two main arguments for using a logarithmic relationship. method of combining 2 events

It gives information an additive property.

Consider two messages s1 and s2.

P(s1) = 0.2
P(s2) = 0.3

P(s1 & s2) = 0.2 * 0.3 = 0.06

quantative not qualitative.
how much info received.
receiver dependent

How much work will it take to predict what event will take place before hand ≡ no. step of binary chop

```
-log2(0.2) + -log2(0.3) ==>
** 4.05889

-log2(0.06) ==>
** 4.05889
```

See also (Shannon & Weaver 1949, p.32)

-- The link with successive halving ---------- ✳

There are N equiprobable possibilities for a particular outcome.

How much work does it take to *discover* which of the possibilities is the real one?

Using successive halving (binary chop) it takes

log2(N) rounded up to the nearest whole number.

[redacted] $\log_2(N) = -\log_2(1/N)$

3 way chop = log3. must stick with this when chosen

Information(s)
= Uncertainty with respect to s
= The amount of "work" it takes to find out if s is the outcome

-- Entropy ---------- Generalises when events are not equi-probable

- Note that the intuitive interpretation of the information equation involves a transformation of perspective. The receiver who assumes an *a priori* probability of Pi for an event Ei, is re-construed as a receiver confronted with a range of 1/Pi equiprobable events. The receiver's uncertainty with respect to Ei is, then, simply the amount of work it would take to discover which in the range of possibilities is actually the case. In effect, this interpretation blurs the distinction between assuming an *a priori* probability of Pi for an event and being confronted with 1/Pi equiprobable events. This is all to the good, since it makes it possible to apply the information equation as an uncertainty measure in the case where the receiver is *really* confronted with a range of equiprobable events. But of course the question is raised

How should we measure uncertainty in the case where the receiver is confronted with a range of possible events which are not equiprobable?

- To cope with this situation we need to generalise the basic equation. The basic measure states that the receiver's uncertainty with an event Ei which has an *a priori* probability of Pi is simply

$I(s_i) - \log_2 P_i$ $\sum_{i=1}^{n} P_i = 1$

- so to calculate the overall uncertainty when there are range of possible events all having different *a priori* probabilities, we simply work out the average or "expected" uncertainty [Watanabe, 1969].

The probability of the uncertainty being exactly

$-\log_2 P_i$

is simply Pi. So to get the overall (expected) uncertainty we simply multiply all the possible uncertainty levels by their corresponding probabilities. Thus we take the sum of

$I(s_i) = P_i * (-\log_2 P_i)$

for all i. This expression makes up the right hand size of Shannon's generalised "entropy of information" equation, referred to as the negentropy equation by Brillouin [Brillouin, 1962]:

$I = -\Sigma\ P_i \log_2 P_i$ Low entropy :- hot water not mixed with cold

-- Information and content ----------

Shannon (and countless other writers) have cautioned people not to confuse "information" with "content" or "meaning".

However, lots of people have attempted to confuse/link the two

There are at least three ways of reading Shannon's caveat.

1. Information is a quantitative measure of content but quantitative measures cannot tell us anything about qualitative properties. Meaning/content is a qualitative property therefore information tells us nothing about meaning.

2. Information is a quantitative measure of something (some statistical property, perhaps to do with signal processing) but NOT content.

3. Content is a purely qualitative property which cannot be quantified.

if event occur which you expect then it tells you nothing. If the event doesn't occur it tells you something.
more uncertain you are the more info you learn when the event doesn't happen.

If a message has content, then it must tell the receiver something.

If it tells the receiver something then it must, in some sense, reduce the receiver's uncertainty w.r.t. some range of possibilities.

Therefore 1 is right...

But...

A receiver may not be able to estimate the a priori probability of a given message. (What is the a priori probability of a specific English sentence?).

A specific message may be given M different (possibly overlapping) interpretations by the receiver. If each interpretation forms one point in a space of possible (mutually exclusive) interpretations then the message has M different information contents.

– Information and knowledge ----------

Information theory is interested in how much information is obtained by *the receiver* of a message.

- It suggests that the amount of information can be calculated by looking at the receiver's estimates for the a priori probabilities for the various possible outcomes. Thus information theory is (fairly) neutral with respect to the status of the message itself. If I see a dice land on a particular face then the amount of information received is 2.584 bits even if, without my knowing it, the dice has been fixed so that it always lands on the same face.

But some philosophers, notably Fred Dretske (1981), interested in knowledge, and how we can be said to have knowledge of the world, have tried to adapt information theory so as to make it say something about the relationship between the source of the message (e.g. the world) and the receiver of the message.

[redacted] approach here involves talking about (i) an actual realisation of just one of a set of possible outcomes and (ii) the perception of same by the receiver.

When the perception perfectly matches the actual situation, information is said to have flowed from the source to the receiver. "Noise" (information perceived but not produced) and "Equivocation" (information not produced) can interfere with this process.

Dretske also persues the line that quantitative measures can tell us something about qualitative properties. The idea seems to be to try to specify a particular meaning in terms of its particular information content.

== Information and Expert systems ----------

So what about Expert Systems?

There are various questions we can ask.

What is the information content of the diagnosis produced by an expert system?

This is just minus the log probability of the diagnosis.

But what is the log probability of the diagnosis for the receiver?

It all depends on the receiver's "knowledge". Also, the probability may vary during the interaction as a result of the receiver making inferences about where the questions are leading...

[redacted]

What is the information content of a single response to a query?

This appears to be a slightly more stable quantity. Provided that the user doesn't tell lies, the log probability of a single response is the log probability of the state of affairs referred to in the response.

How many questions should an expert system need in order to be able to produce a diagnosis?

In the ideal case it should be able to reduce the possibilities by half with each question (cf. 20 questions). Therefore the number of questions should not greatly exceed the number of bits of information contained in the final diagnosis.

== Reading ----------

Dretske 1981
Baierlein 1971
Bar-Hillel & Carnap 1953
Bar-Hillel 1965
Brillouin 1962
Gatlin 1972
Gluck & Carter 1985
Jackson 1953
Jones 1979
MacKay 1953
Mackay 1969

Shannon & Weaver 1949
Watanabe 1969
Thornton, Links Between, 1988

Expert Systems & Knowledge Acquisition: 7

Chris Thornton – January 31, 1989

-- INTRO. TO AUTOMATIC KNOWLEDGE ACQUISITION --

Manual knowledge acquisition is hard, for lots of reasons. In some cases, it may be impossible.

People feel that the solution may lie in the development of programs which *automatically* acquire expert knowledge; i.e. learning programs.

Quite a lot of work is now going on (cf. Bratko & Lavrac 1987). This focusses primarily on two families of algorithms. The ID3 family (derived from the CLS system) and the AQ family (derived originally from the Candidate-Elimination algorithm and Winston's ARCH program). The remaining lectures will concentrate on these two families of algorithms.

-- Why not just use statistics? ----------

Isn't there an easier way to automatically acquire expert knowledge?

Why not just use Bayesian statistics? very combinatorial.

In the classic medical diagnosis example a certain patient P shows a symptom S and we want to know to what extent this evidence justifies the conclusion (i.e. diagnosis) that they have disease D. We know what the probability is of a patient showing symptom S given that they have disease D but not the probability of them having D given that they are showing S. The probability information is back-to-front.

-- Bayes theorem ----------

Bayes theorem provides a way of transforming information relating to the probabilities of premises (bits of evidence) given certain conclusions into the probabilities of conclusions given certain premises.

Take the case where we have the disease D and the symptom S. Assume that we know the number of people who suffer from D and we know how many people there are overall. This means that we can work out the proportion of all people who have the disease D and therefore the *a priori* probability of having D. It is just

P(D) = <number of D sufferers> / <number of people>

Assume that we also know the proportion of D sufferers who show symptom S. That is to say, we know the *conditional probability* of someone showing S given that they have D.

P(S|D)

Now, the probability of being someone who shows S because of having D is just the probability of being someone with D *multiplied* by the probability of showing S because of having D.

P(D) * P(S|D)

The people who show S through having D are a subset of the set of all people who show S. The probability of being in this subset, given the fact that you show S, is just the ratio between the number of people in the subset and the number of people in the superset. Thus the probability of showing S through having D must be

P(D) * P(S|D) / P(S)

- This is Bayes theorem.

-- Problems with Bayesian statistics ----------

- In principle, Bayes theorem provides a way of deriving probabilities for conclusions from probabilities for premises. Thus, it provides a way of constructing the sort of rulebase we need in order to be able to implement expert reasoning in domains where we only have access to data giving probabilities of premises. Unfortunately, a simple-minded application of Bayes theorem in this sort of situation leads directly into difficulties.

- Let us continue to think about conclusions and premises in terms of diseases and symptoms; and let us assume that we have data which show, for any arbitrary symptom, what the probability is of showing that symptom given that the patient has some given disease D. We would like to derive from these data a rulebase in which given sets of symptoms justify given conclusions with given certainties. In effect, we need to work out what the probability of having some given disease is, given that the patient has a certain set of symptoms.

Note that for each possible combination of one symptom and one disease there is one version of the formula. If there are 3000 possible symptoms and 500 possible diseases, that means over a million formulae.

Normally we will be interested in the conditional probability of someone having a disease given that they are showing *several* symptoms. If we start trying to construct all the relevant rules we will need to evaluate one version of the formula for all the different possible combinations of symptoms. In this scenario, the number of computations needed to deal with a certain case rises very fast indeed (see Charniak and McDermott, 1985, pp. 461-462)

The general conclusion is that simple-minded exploitation of Bayesian statistics for the purposes of automatically generating expert systems is not a practical proposition.

- In fact the situation is not quite as bad as it seems. If it is known that in some cases there is no link between a given symptom and a given disease (i.e. if it can be assumed that the variable representing the symptom and the variable representing the disease are *statistically independent*), then all the formulae which try to deal with that link can be dispensed with. Exploiting this idea can make Bayesian statistics a realistic proposition again.

- Statistical independence plays a very important role in many basic theorems in probability theory; e.g. the fundamental rule that the probability of two events A and B both occurring together is just the probability of A occurring multiplied by the probability of B occurring is only valid if A and B are statistically independent. If they are not statistically independent -- if for example the occurrence of B marginally increases the *a priori* probability of A -- then the formula is invalid.

-- Machine Learning ----------

So, back to machine learning.

First some definitions of learning:

"Learning denotes the changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more effectively the next time." (Simon, 1983, ML1, chapter 2)

"When a computer system improves its performance at a given task over time, without re-programming, it can be said to have learned something." (Forsyth & Rada, 1986, chapter 1)

"Learning refers to the change in a subject's behaviour to a given situation brought about by his repeated experiences in that situation, provided that the behaviour change cannot be explained on the basis of native response tendencies, maturation, or temporary states of the subject (e.g., fatigue, drugs, etc.). (Hilgard & Bower, 1975, p. 17).

"Learning is making useful changes in our minds." (Minsky, The Society of Mind, 1988)

"[Learning is] ... theory formation, hypothesis formation, and inductive inference." (Dietterich et al, 1982, HB3, p. 327).

"Learning is constructing or modifying representations of what is being experienced." (Michalski, Understanding 1986, ML2, chapter 1) Most preferred (cf PDP).

-- Expertise as classification ----------

We are only interested in learning processes which might be useful for the automatic construction of expert systems.

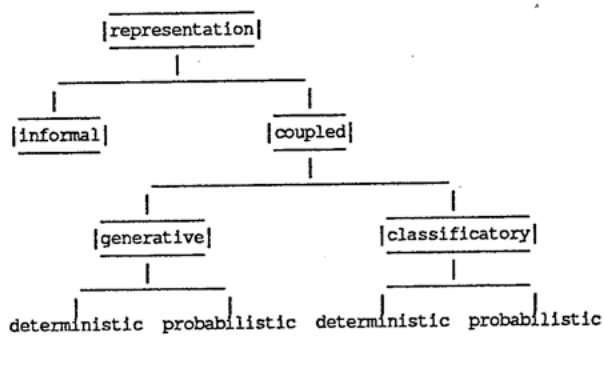
ESs typically do some kind of diagnosis.

Diagnosis can be viewed as classification.

- Can view the problem of learning expertise as the problem of learning to classify data in the correct way.

- Look at strategies for learning representations of classes.

-- Informal v. coupled representations ----------



ML: machine learning.

CLS = Concept learning System

31/1/89.

D = some data
R = representation of D

- If there is no computational relationship between R and D, R is an "informal" representation.
- If there is a computational relationship between R and D, R is a "coupled" representation.

- If R is a computer program which generates D, R is a "generative" representation.
- If R is a program which correctly classifies elements of D, R is a "classificatory" representation.

- A generative (or classificatory) representation which generates (classifies) D perfectly is "deterministic".
- A generative (or classificatory) representation which generates (classifies) D to some given (satisfactory) level of accuracy is "probabilistic".

-- The neighbourhood representation scheme (NR) -----

The behaviour of many learning mechanisms can be thought about in terms of the construction of coupled representations of a certain type -- namely "neighbourhood representations" (NRs).

Basic assumptions -

L is the description space; i.e. the set of all possible data elements. The "data language" is a formal specification of L (e.g. a grammar).

D is the subset of L which we are interested in representing.

A coupled representation R is a program which generates/classifies D to a given level of accuracy.

[redacted] will have a declarative part (RD) and a procedural part (RP). D is just the result of running RP on RD.

R counts as a *neighbourhood representation* if

- it treats the set of possible data elements as an N-dimensional space L

- defines D as the set of points enclosed in M neighbourhoods of L.

== Example ----------

Imagine that a data element is a 2-tuple with the first element drawn from the list [huge vast big tiny med micro] and the second element drawn from the list [moped bike jet car prop rocket]. Possible data elements are

[huge car]
[big bike]

D is a list of 9 data elements.

vars D = [[big prop] [big car] [big moped] [med prop] [med car] [med moped] [tiny prop] [tiny car] [tiny moped]];

== A representation for D ----------

If L is configured as a 2-dimensional space, a representation for D can take the form of a [redacted]r of 2-tuples:

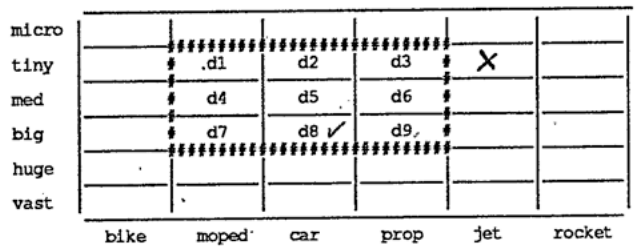
```
vars RD = [[[tiny moped][big prop]]];

define interpret(RD);
   vars x = [micro tiny med big huge vast],
        y = [bike moped car prop jet rocket],
        s v s1 v1 s2 v2 result = [];
   RD --> [[?s1 ?v1][?s2 ?v2]];
   for s in x do
   nextunless(s = s1 or s = s2 or x matches [== ^s1 == ^s == ^s2 ==]);
      for v in y do
      nextunless(v = v1 or v = v2
         or y matches [== ^v1 == ^v == ^v2 ==]);
         [[^s ^v] ^^result] -> result;
      endfor
   endfor;
   return(result);
enddefine;

define RPgenerate(RD);
   return(maplist(RD, interpret <> explode))
enddefine;
```

eg generative grammar.

```
RPgenerate(RD) =>
** [[big prop] [big car] [big moped] [med prop]
   [med car] [med moped] [tiny prop] [tiny car] [tiny moped]]

RPgenerate(RD) = D ==>
** <true>

define RPclassify(input, RD);
   return(member(input, RPgenerate(RD)))
enddefine;

RPclassify([big car], RD) ==>
** <true>

RPclassify([tiny jet], RD) ==>
** <false>
```

== Neighbourhoods identified by R ----------

- The function RPgenerate and its input list RD form a neighbourhood representation. It "works" by arranging the data elements into sequences and combining them together to form a 2-dimensional space L. In this context, D can be represented in terms of a set of 4 "boundaries" in L. Put together these form a rectangle which encloses all the points corresponding to elements of D. D can therefore be reconstructed just by interpolating between the boundaries (see the two "for" loops in "interpret").



- RD is just an identification of the 4 boundaries -- "big" is the lower horizontal boundary, "tiny" is the upper horizontal boundary, "moped" is the leftmost vertical boundary and "prop" is the rightmost vertical boundary. Put together, these boundaries form a rectangle which encloses all the points (i.e. 2-tuples) in D.

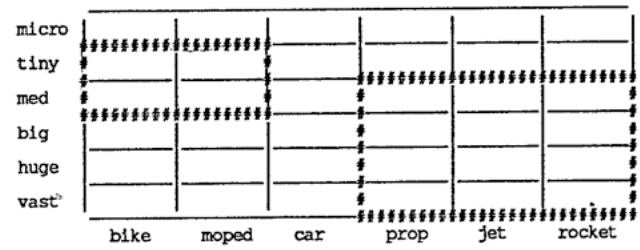
== Multiple-neighbourhood representations ----------

Different RDs represent different bodies of data.

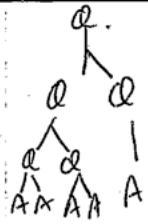
```
RPgenerate([[[tiny bike][vast moped]]]) =>
** [[vast moped] [vast bike] [huge moped] [huge bike] [big moped]
  [big bike] [med moped] [med bike] [tiny moped] [tiny bike]]
```

RDs can identify multiple neighbourhoods; in the present case, this means that they contain multiple sublists.

```
RPgenerate([[[tiny bike] [med moped]][[med prop][huge rocket]]]) =>
** [[med moped] [med bike] [tiny moped] [tiny bike] [huge rocket]
  [huge jet] [huge prop] [big rocket] [big jet] [big prop] [med rocket]
  [med jet] [med prop]]
```



Expert Systems & Knowledge Acquisition: 8



Chris Thornton -- February 1, 1989

-- INDUCTION OF DECISION TREES ----------

Different forms of NR are constructed by different types of learning mechanism. A very simple case involves a type of NR which is closely related to what psychologists call the "classical concept definition". This type of NR can be constructed by a fairly simple mechanism. A slightly more sophisticated variant is constructed by a well-known and frequently used SP learning mechanism -- the Classification algorithm, which is derived from Hunt, Marin and Stone's Concept Learning System (CLS). This algorithm forms the main component in Quinlan's ID3 program.

-- Classical concepts ----------

- A concept identifies a class (or category) of entities. The entities may be either concrete or abstract; they are said to be "covered" by the concept. Technically, a concept is a predicate which returns true only for elements which are covered; i.e. it is a classficatory, coupled representation.

Aristotle on concepts:

- The representation of a concept (in the mind) is a summary description of the entire class rather than an enumeration
- The representation is made up from the set of features which are shared between all members of the class they all have X ...
- The features that represent a concept are singly necessary and jointly sufficient to define that concept.

- So, a classical concept definition is just a set of shared features. Is this a NR?

-- Learning classical concepts ----------

D = data (or description) language
D = D viewed as a space
C = data elements (or class) to be represented
g = generative representation of C

vars C = [[medium red brick] [small green brick] [large blue brick]];

```
define g(C);
  vars f, element, result = [], features = maplist(C, explode);
  for f in features do
    if length([^(foreach [== ^f ==] in C do it endforeach)])
      = length(C)
    and not(member(f, result)) do
      [^^result ^f] -> result;
    endif;
  endfor;
  return(result);
enddefine;

g(C) ==>
** [brick]

g([[medium red brick] [red small brick] [red large brick]]) ==>
** [red brick]
```

-- Classical concepts as NRs ----------

Imagine any feature belongs to a set of mutually exclusive features
colours: red, green, blue
sizes: large, small, medium,

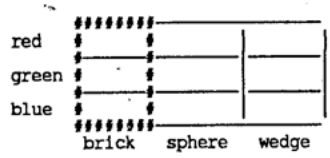
Each set of mutually-exclusive features forms a dimension of a n-dimensional space D.

If a concept definition does not specify a feature from a given dimension then members of the concept can have any feature from that dimension (according to the definition...). The concept definition therefore identifies a single rectangle (in general, a hyper-rectangle) in D. The boundaries of this hyper-rectangle in any "unspecified" dimension enclose the entire dimension. The boundaries in a "specified" dimension just enclose the specified feature

== Example ----------

Concept definition: [brick]
Covered elements: [[red brick] [green brick] [blue brick]]

This forms a NR -- it identifies a single region of D.



-- Disjunctive concepts ----------

- Recently, the classical view of concepts has been under attack. People have pointed out that there are many concepts which cannot be represented using a classical definition.

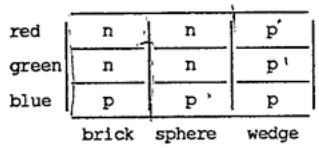
Imagine a game involving little, coloured objects. Objects have different values according to their shape and colour. 5 different objects are all worth 10 points:

[[blue brick][blue sphere][green wedge][red wedge][blue wedge]]

Any object from this set is a "tenner".

Can the concept "tenner" be represented as a classical concept?

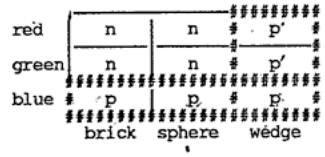
- No -- there is no feature which is shared between all the examples. In terms of D, there is no rectangle which encloses all the points corresponding to tenners but excludes all the points corresponding to non-tenners.



n = not in concept
p = in concept.

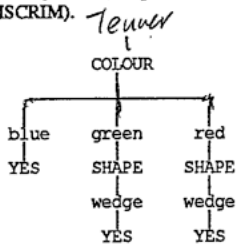
== Multiple NRs ----------

The tenner concept could be defined in terms of a disjunction: [blue] OR [wedge]. This would correspond to an identification of two rectangles.



-- The Classification algorithm ----------

The main algorithm for deriving this sort of disjunctive representation is called the Classification algorithm. It was originally presented by Hunt, Marin & Stone as the Concept Learning System (hence the acronym CLS). It produces a representation in the form of a decision tree (cf. TEACH DISCRIM).



Equivalent to disjunction: [blue] OR [green wedge] OR [red wedge] -- i.e. 3 rectangles in D. NB. this is not the optimal NR.

== Main steps in the algorithm ----------

Initialise C to be the set of elements

- If all elements in C are positive then create YES node and halt

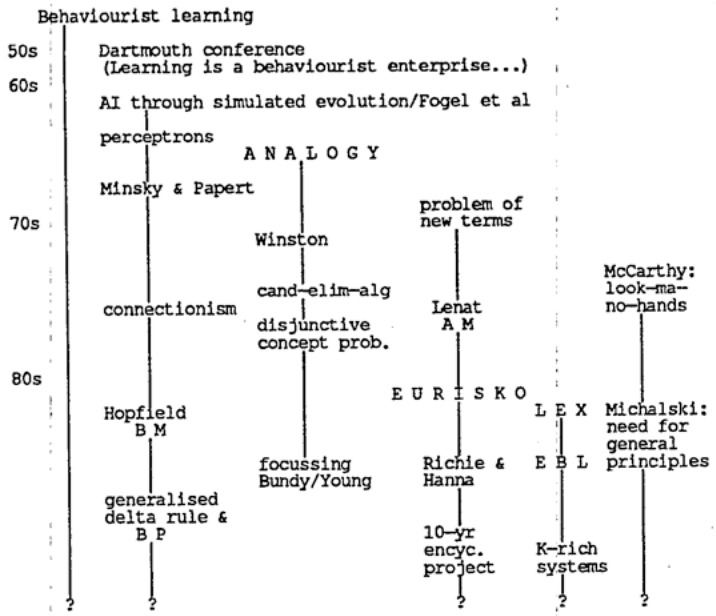
- If all elements in C are negative then create NO node and halt

default: choose first one.

- Otherwise select (perhaps using a heuristic) a feature F with values v1, v2, v3... vN. Partition C into subsets C1, C2, C3 ... CN, according to their values on F. Create a branch with F as parent and C1 etc. as child nodes.

- Apply algorithm recursively to each child

-1- input is ~~test~~ set of instances with each instance noted as being in the class (p) or not in the class (n).


Behaviourist learning
50s
Dartmouth conference
(Learning is a behaviourist enterprise...)
60s
AI through simulated evolution/Fogel et al
perceptrons
A N A L O G Y
Minsky & Papert
problem of
new terms
70s
Winston
McCarthy:
look-ma-
no-hands
cand-elim-alg
connectionism
Lenat
A M
disjunctive
concept prob.
80s
E U R I S K O
L E X
Michalski:
need for
general
principles
Hopfield
B M
focussing
Bundy/Young
Richie &
Hanna
E B L
generalised
delta rule &
B P
10-yr
encyc.
project
K-rich
systems
?
?
?
?
?

Expert Systems & Knowledge Acquisition: 9

Chris Thornton -- February 6, 1989

-- ID3 ----------

ID3 is an extension of CLS.

CLS accepts a set of positive and negative instances of some class and builds a decision tree (classificatory representation) for the class. It does this by repeatedly splitting the instances on a selected attribute.

From a decision tree it's easy to derive a decision rule. The rule is just the disjunction of conjunctions constructed by tracing paths from the root node to "yes" nodes.

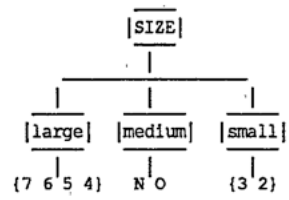
== CLS implementation ----------

```
/* CLS.p - implementation of Classification algorithm */

vars data_elements decision_tree;


define setup;
  vars i;
  [  [size large medium small]
     [shape sphere brick wedge pillar]
     [colour red blue green yellow] ] -> database;
  [  [medium blue brick ^false] [small red wedge ^false]
     [small red sphere ^true] [large red wedge ^true]
     [large green pillar ^true] [large red pillar ^false]
     [large green sphere ^false] ] -> data_elements;
  [(for i from 1 to length(data_elements) do i endfor)] -> decision_tree;
enddefine;

define elements_with(value, elements);
  vars element result = [];
  for element in elements do
    if member(value, data_elements(element)) then
      [^element ^^result] -> result
    endif;
  endfor;
  return(result);
enddefine;

define ispositive(element);
  return(data_elements(element) matches [== ^true])
enddefine;

define isnegative(element);
  return(data_elements(element) matches [== ^false])
enddefine;

define make_node(elements);
  vars x;
  if not(elements matches [== ?x:isnegative ==]) then
    return("YES")
  elseif not(elements matches [== ?x:ispositive ==]) then
    return("NO")
  else
    return([^(explode(elements))])
  endif;
enddefine;

/*

define extend_tree(tree) -> tree;
  vars attribute values attribute_name elements database numbers;
  if isvector(tree) then
    [^(explode(tree))] -> numbers;
    database --> [[?attribute_name ??values] ??database];
    [^(attribute_name;
      for value in values do
        if (elements_with(value, numbers) ->> elements) /= [] then
          [^value ^(make_node(elements))]
        endif
      endfor)] -> tree;
  elseif islist(tree) then
    flush([^(tree(1)) ==]);
    maplist(tree, extend_tree) -> tree;
  endif;
enddefine;

define go;
  extend_tree(decision_tree) -> decision_tree;
  showtree(decision_tree);
enddefine;
```

```
define run;
  until database = [] do go(); enduntil;
enddefine;
```

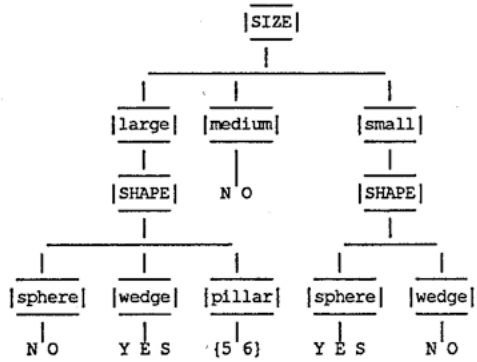
== Interaction ----------

```
setup();
go();
```



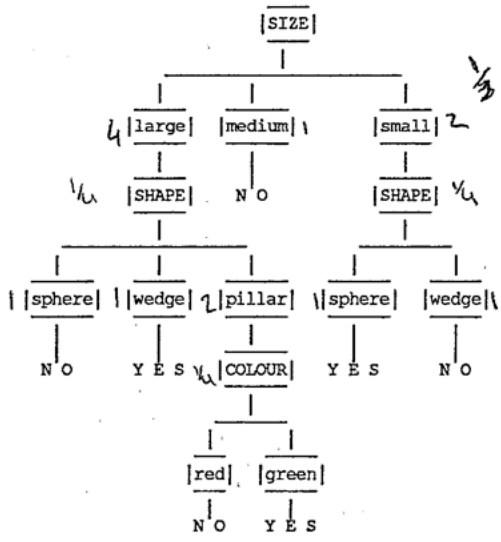
== Splitting on "SHAPE" ----------

```
go();
```



-- Splitting on "COLOUR" ----------

```
go();
```



Disjunction: [large wedge] OR [large green pillar] OR [small sphere]

-- ID3 main loop ----------

Quinlan's ID3 algorithm is an extension of CLS which is suitable for applications in which C is very large. The main steps are as follows.

- Select a random subset ("window") of size W from C
- Use CLS to form a rule for W
- Scan through rest of C to find exceptions to current rule
- Form new window with some of new exceptions plus old elements
- Repeat whole routine until no more exceptions

SYLLABUS?!?!

-- The information-theoretic heuristic ----------

- ID3 uses an information theoretic heuristic for deciding which attribute to split instances on in any given case. The heuristic allows ID3 to choose a feature that best discriminates between positive and negative elements (i.e. sorts them out into even-size bundles)

Imagine that we have some partial decision tree (leaf nodes are not homogenous) and are given some arbitrary instance to classify.

We can estimate the probability of it being positive or negative by finding the leaf node which contains it and working out the ratio of positive to negative instances at that node.

If P+ is the probability that it is positive and P- is the probability that it is negative, then the information content of a message (from an oracle) which tells us which it is

$-(P+ \log_2 P+) - (P- \log_2 P-)$

cf. entropy equation: $-\Sigma Pi \log_2 Pi$

-- Maximising information gain (& minimising entropy)

We want to reach a stage where the message from the oracle has no information content (because the decision tree tells us what the class of the instance is). So choose which attribute to split the instances on next so as to minimise the expected information content of such a message. This is the same as *maximising* the information gained.

- We can talk about how much information we still need for a particular leaf node of the decision tree (i.e. what the information content of a message about an instance at a particular leaf node will be). And we can work out the overall expected information content by multiplying information contents for particular nodes by the probabilities that an arbitrary [redacted]ance will be *at* that node (estimating probablities using relative frequencies as usual).

So, at each stage, chose the next attribute so as to
- minimise the information content of a message from the oracle, or
- maximise our information gain

Shortcut -

At each stage choose next attribute so as to minimise the entropy of the distribution of positive instances across leaf nodes (cf. LIB CLASSIFICATION).

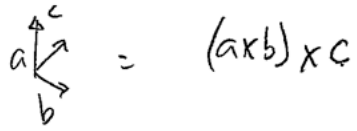
= Reading ----------

Thompson & Thompson, 1986; (nice BYTE article on ID3); - psychologists.
Smith & Medin 1981, chapter 3;
Bundy Silver & Plummer 1985; AI Journal
Hunt Marin & Stone 1966; original
Quinlan 1979; Quinlan 1983; m/c learning Info-theoretic Heurist.
Bratko & Lavrac 1987 (reports some current work using ID3);
Section on CLS and ID3 in HB3, pp. 406-410;
TEACH CLASSIFICATION; TEACH DISCRIM;

AI handbook vol 3

Good Decision tree
bushy trees.
Poor Decision trees. Deep ones.

relative frequencies of +ve, -ve nodes.



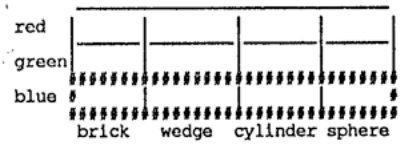
Chris Thornton -- February 13, 1989

-- VERSION SPACES AND CANDIDATE ELIMINATION ---------

In the last lecture we looked at a quite simple form of NR where

- D is a space with dimensions ranging over sets of mutually-exclusive features
- the representation identifies M neighbourhoods
- every neighbourhood is of the same form -- a boundary in a given dimension either covers the entire dimension or just a single value.

This form of neighbourhood can be easily defined as a list of all the covered single values; e.g. [blue] defines the neighbourhood shown below.

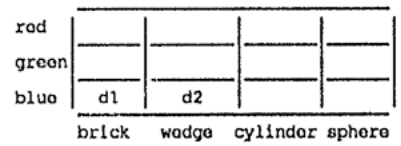


- The classical concept learning algorithm can form NRs identifying a single neighbourhood of this form. The Classification algorithm (CLS) can form NRs identifying multiple neighbourhoods. It actually constructs a decision tree but this corresponds to a set of classical concepts.

-- The need for a richer description language ----------

Imagine

vars C = [[blue brick] [blue wedge]]



How can we construct a NR which includes d1 and d2 but not [blue cylinder], [blue sphere] etc.? In terms of concept learning -- how can we form the concept [blue block]?

-- Generalised descriptions ----------

Imagine that the data language includes elements like

[blue block]

and that there is a covers (or "generalises") function

```
covers([blue block]) ==>
** [[blue brick] [blue wedge]]
```

Then the element [blue block] forms a representation for the set {d1,d2}

But how could this sort of representation be implemented or learned?

-- Generality orderings ----------

We can define a generality ordering

```
[red block] more_general_than [red wedge] ==>
** <true>
[red wedge] more_general_than [red brick] ==>
** <false>
```

in terms of the covers function

```
define 7 d1 more_general_than d2;
  return(subset(covers(d2), covers(d1)))
enddefine;
```

-- Version spaces ----------

Given the generality ordering it is possible to represent a set of descriptions in terms of its most general member(s) and its most specific member(s).

This is essentially a classificatory representation.

- Given some set C of (generalised) descriptions represented in terms of its most-general member and its least general member, we can decide whether some new description d is in C simply by testing whether d is less general than (or as general as) the most general description and more general than (or as general as) the least general description.

Mitchell has called this representation of a set a "version space" (Mitchell 1977; Mitchell 1982).

The set of most general elements is called the "general boundary" -- written "G".

The set of most specific elemetnts is called the "specific boundary" -- written "S".

Normally G and S are singleton sets.

NB.

- The version space is a set of descriptions.
- Generalised descriptions cover other descriptions
- Primitive descriptions cover just themselves

-- The Candidate-Elimination algorithm ----------

Find some description which covers all of P and excludes all of N, where P is a set of positive instances (elements describing positive instances) and N is a set of negative instances (elements describing negative instances)

Candidate-Elimintation algorithm:

Initialise the version space. Set G to be the set of most general descriptions. Set S to be the set of most specific descriptions.

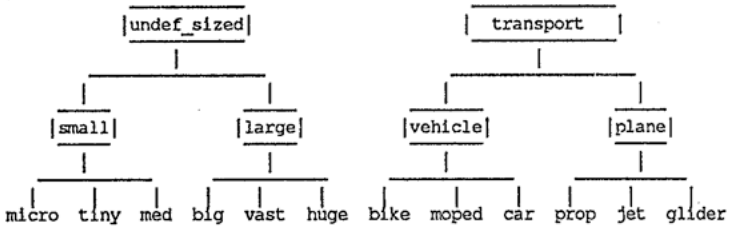
For each new instance

- if it's positive then update S so that all descriptions cover the new instance
- it it's negative then update G so that none of the descriptions cover the new instance.

If ever G = S then exit. Any further refinement of the version space will result in a maximally specific element becoming more general than a maximally general element (or vice versa).

-- Tree-based description languages ----------

In the simplest case, the data language is defined in terms of N generalisation hierarchies; e.g.



- A description (data element) is a sequence containing one node label from each tree. The elements covered by a given description d1 are just those elements which can be "grammatically derived" from d1 (cf. Utgoff, Shift of Bias 1986); i.e. all the elements which can be constructed by pairing up the leaf nodes which are in the subtrees belonging to terms in d1.

Elements made up purely of leaf nodes only cover themselves (these are primitive descriptions).

```
covers([tiny bike]) ==>
** [[tiny bike]]
```

Other descriptions cover multiple descriptions (these are generalised descriptions).

```
covers([small bike]) ==>
** [[tiny bike][micro bike][med bike]]
```

== Tree based descriptions as NRs ----------

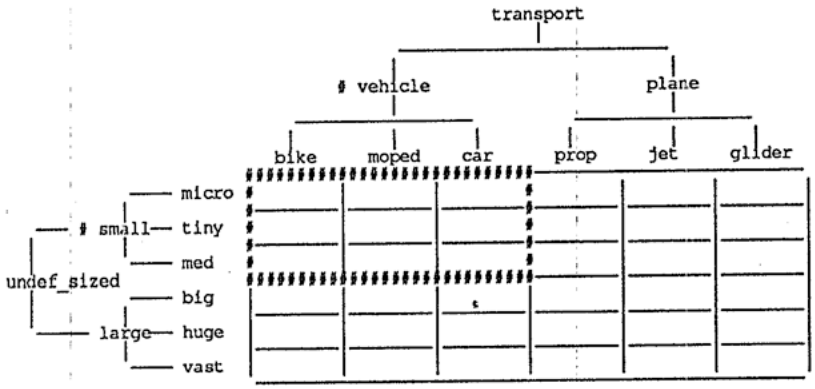
- A description such as [small vehicle] can be seen as a neighbourhood representation for the set of covered instance descriptions.

[small vehicle]

covers

[[micro bike] [micro moped] [micro car]
[tiny bike] [tiny moped] [tiny car]
[med bike] [med moped] [med car]]

- A given description "marks" a node in each generalisation tree. These marks can be seen as defining a region in a 2-dimensional space. The cells in the region correspond to the covered descriptions.

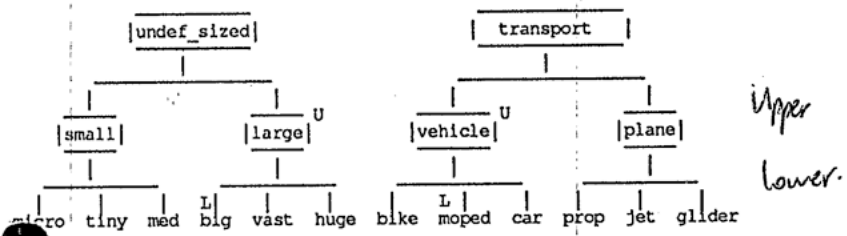


-- Implementing version spaces using tree marks ----------

We can represent a singleton G and a singleton S (making up a specific version space) in terms of two sets of marks.

- The node marked by G will always be above the node maked by S in any tree. So we can talk about G's marks as "upper marks" and S's marks as "lower marks". This way of representing a version space is used in Focussing (see next lecture).

A description is in a given version space if it can be constructed by pairing up nodes which are between the upper and lower marks in the relevant trees.



Upper

lower.

S = [big moped], G = [large vehicle]

VS = [[large vehicle] [large moped] [big vehicle] [big moped]]

== Version spaces as NRs ----------

Assume |G| = 1 and |S| = 1

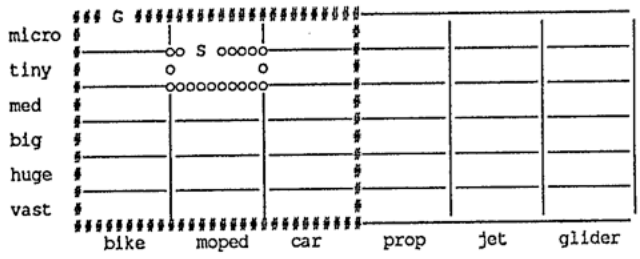
S is then the most specific description which covers all the positive instances. G is the most general description which excludes all the negative instances.

- If we view G and S as NRs in L (see above), then G is just the biggest feasible rectangle which does not enclose any cells corresponding to negative instances. S is the smallest feasible rectangle which encloses all the cells corresponding to positive instances. A feasible rectangle is just a rectangle whose boundary in any dimension encloses all the descendants of a specific internal node.

- The VS consists of all descriptions which are no less general than S but no more general than G. A description always corresponds to a rectangle in S. Thus the VS is effectively the set of all feasible rectangles which can be inscribed between the rectangle for G and the rectangle for S.

S = [tiny moped], G = [undef_sized vehicle].

VS = [[tiny vehicle] [small vehicle]
[undef_sized vehicle] [undef_sized moped]]



Expert Systems & Knowledge Acquisition: 11

Chris Thornton -- February 13, 1989

-- FOCUSSING ----------

There are various ways in which the CE algorithm might be implemented. A common variant involves a process known as Focussing (Young et al. 1977; Bundy Silver & Plummer 1985). This is closely related to Winston's ARCH learning method (Winston 1975, Winston, Artificial Intelligence 1984).

CE algorithm consists of two responses:

- After presentation of a positive instance, generalise S as necessary
- After presentation of a negative instance, specialise G as necessary

If the data language is implemented using generalisation hierarchies and the VS is represented using upper and lower marks then generalising S involves raising lower marks, and specialising G involves lowering upper marks.

- So, after presentation of a positive instance, the algorithm has to raise the lower mark in each tree so as to make sure that every description in the VS covers the new instance. This actually involves raising the lower mark until it is above the leaf which is specified in the instance description.

- After presentation of a negative instance, the algorithm has to lower the upper mark in each tree so as to make sure that no description in the VS covers the new instance. This actually just involves lowering the upper mark in *one* tree so as to ensure that it does not have any leaf specified in the instance description descended from it. If there is more than one way to do then the instance is referred to as a "far miss". If there is just one way, the instance is a "near miss".

Initialisation of the algorithm involves placing upper marks on all the root nodes and lower marks on the leaf nodes identified by the first positive instance.

-- Implementing mark-manipulation ----------

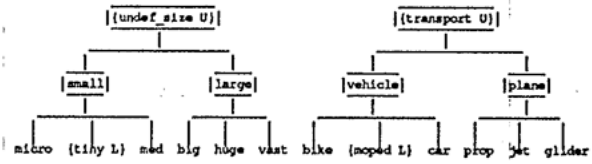
```
/* CE.p - implementation of Candidate-Elimination algorithm */

uses showtree;

vars value element data_elements trees upper_mark lower_mark G S
  myshowtree = erase;

define isnegative(element); return(last(element) = false) enddefine;
define ispositive(element); return(last(element) = true) enddefine;

define parent_of(node, tree);
  vars subtree parent subtrees;
  if tree matches [== ^node ==] or tree matches [== [^node ==] ==] then
    return(tree)
  elseif tree matches [= ??subtrees] then
    for subtree in subtrees do
      if parent_of(node, subtree) ->> parent then return(parent) endif;
    endfor;
  endif;
  return(node = tree);
enddefine;

define subtree_of(n);
  if [^n ==] isin parent_of(n,[0 ^^trees]) do return(it) else return(n) endif;
enddefine;

/*
/* a node which is in a tree must have a parent */
vars contains = parent_of;

define above(n1, n2);
  return(contains(n2, subtree_of(n1)));
enddefine;

define covers(concept, element);
  vars i;
  for i from 1 to length(trees) do
    unless above(concept(i), element(i)) do return(false) endunless;
  endfor;
  return(true);
enddefine;

/*
define raise(tree);
  while parent_of(lower_mark, tree) matches [?parent ==]
  and above(upper_mark, parent)
  and not(above(lower_mark, value)) do
    parent -> lower_mark;
  endwhile;
  return(above(lower_mark, value))
enddefine;

define lower(subtree);
  if contains(lower_mark, subtree)
  and not(contains(value, subtree)) then
    /* reset upper_mark appropriately */
    if not(subtree matches [?upper_mark ==]) do subtree -> upper_mark endif;
    return(true)
  elseif subtree matches [= ==] then
    for node in tl(subtree) do if lower(node) then return(true) endif endfor;
  endif;
  return(false);
enddefine;

/*

define process_new_element(element);
  vars tree update i newG = copylist(G), newS = copylist(S), success;
  if isnegative(element) do lower -> update else raise -> update endif;
  for i from 1 to length(trees) do
    [^(G(i), S(i), element(i))] --> [?upper_mark ?lower_mark ?value];
    if update = lower do subtree_of(upper_mark) else trees(i) endif -> tree;
    update(tree) -> success;
    upper_mark -> newG(i);
    lower_mark -> newS(i);
    quitif(isnegative(element) and success);
    if not(isnegative(element)) and not(success) then return(false) endif;
  endfor;
  if success do newG -> G; newS -> S; return([^G ^S]) else return(false) endif;
enddefine;

/*

define add_marks(tree);
  vars x y mark;
  if upper_mark = lower_mark and tree = upper_mark then
    return([^upper_mark U L])
  elseif tree = upper_mark then
    return([^upper_mark U])
  elseif tree = lower_mark then
    return([^lower_mark L])
  elseif islist(tree) then
    return(maplist(tree, add_marks))
  else
    return(tree)
  endif
enddefine;

define add_marks_in(i);
  [^(G(i), S(i))] --> [?upper_mark ?lower_mark];
  add_marks(trees(i));
enddefine;

define convert_trees;
  vars i;
  return([^(for i from 1 to length(trees) do add_marks_in(i) endfor)])
enddefine;

/*

define set_marks;
  vars element;
  data_elements --> [?element ??data_elements];
  element --> [??S =];
  maplist(trees, hd) -> G;
enddefine;

define set_default_trees;
  [
    [undef_sized
      [small micro tiny med]
      [large big huge vast]
    ]
    [transport
      [vehicle bike moped car]
      [plane prop jet glider]
    ]
  ] -> trees;
enddefine;
```

```
vars set_trees = set_default_trees;

define CE(data_elements);
  set_trees();
  set_marks();
  while data_elements matches [?element ??data_elements] do
    unless process_new_element(element) do return(false) endunless;
  endwhile;
  return([^G ^S]);
enddefine;

/*

define setup;
  set_trees();
  [  [tiny moped ^true] [tiny car ^true] [big jet ^false]
    [med car ^true] [med jet ^false] ] -> data_elements;
  set_marks();
  myshowtree(convert_trees())
enddefine;

define go();
  if data_elements matches [?element ??data_elements] then
    if process_new_element(element) do
      myshowtree(convert_trees())
    else
      [cannot process ^element] ==>
    endif;
  else
    [G = ^G] ==> [S = ^S] ==>
  endif;
enddefine;

define run;
  setup();
  until data_elements = [] do go() enduntil;
enddefine;
```

== Interaction ----------

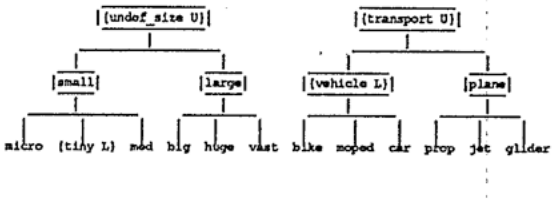
```
setup();
```

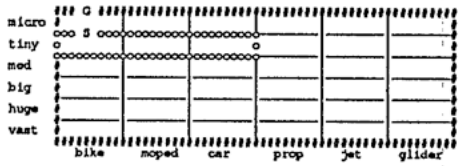
First positive instance: [tiny moped]



== Processing [tiny car ^true] ----------

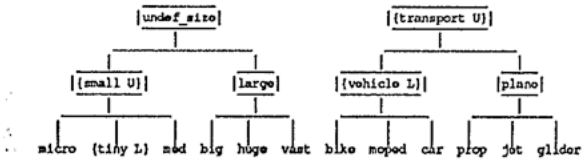
```
go();
>process_new_element [tiny car <true>]
<process_new_element
```

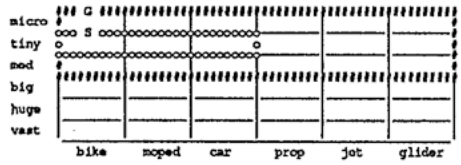




== Processing [big jet ^false] ----------

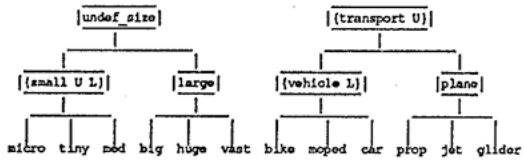
```
go();
>process_new_element [big jet <false>]
<process_new_element
```

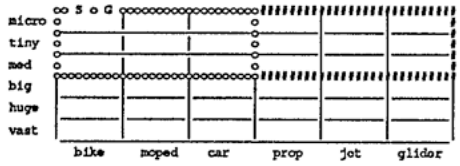




== Processing [med car ^true] ----------

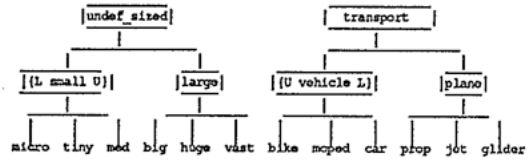
```
go();
>process_new_element [med car <true>]
<process_new_element
```

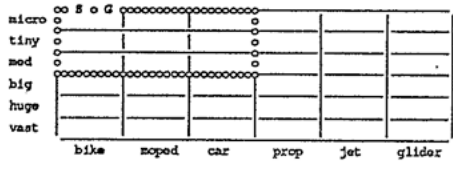




== Processing [med jet ^false] ----------

```
go();
>process_new_element [med jet <false>]
<process_new_element
```






```
go();
** [G = [small vehicle]]
```

== Reading ----------

Mitchell, Utgoff & Banerji, 1983; -- describes one of the most successful applications of CE/Focussing (a system called LEX which learns to do symbolic integration problems). - Also talks about the "inadequate description language" problem (sometimes called the "problem of new terms").

Winston, 1975; Winston Artificial Intelligence 1984; -- describes the forerunner to CE/Focussing: the ARCH program.

Bundy Silver & Plummer, 1985; -- theoretical analysis of Focussing/CE.

Wielemaker & Bundy, DAI Research Paper 262 1985; -- full discussion of tree-hacking technique.

Murray 1987; Utgoff, *Shift of Bias*, 1986; -- other ways of dealing with unlearnable (disjunctive) concepts.

TEACH/HELP/LIB FOCUSSING -- interactive Focussing package

== Questions ----------

Can the first element presented to the Focussing algorithm be either positive or negative? If not, why not?

What is the search strategy implemented by the "lower" function shown above. Is it the ideal strategy. Would a breadth-first strategy be more effective. Why?

Expert Systems & Knowledge Acquisition: 12

Chris Thornton – February 18, 1989

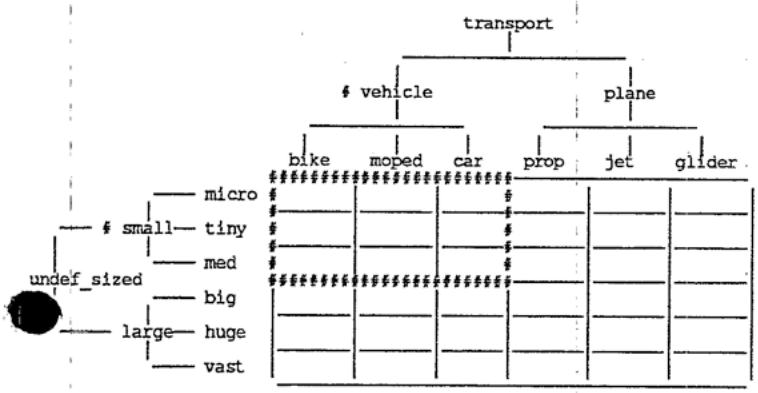
– THE AQ ALGORITHM ----------

Effective disjunctive concepts (i.e. multi-neighbourhood representations) can be learned by the Classification algorithm if data elements are described in terms of simple features.

If the data language allows descriptions to include terms which generalise (cover) other terms; e.g.

"vehicle" covers "lorry" and "car"

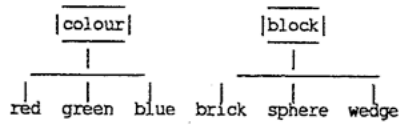
conjunctive concepts (single-neighbourhood representations) can be constructed in the form of descriptions with generalised terms. A description d1 containing generalised terms covers all other data elements which can be grammatically derived from d1. The Candidate-elimination algorithm is the standard mechanism for deriving this sort of representation.

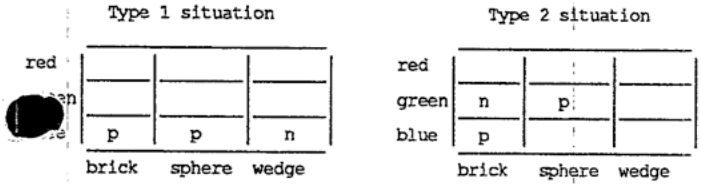


Instance descriptions covered by [small vehicle]

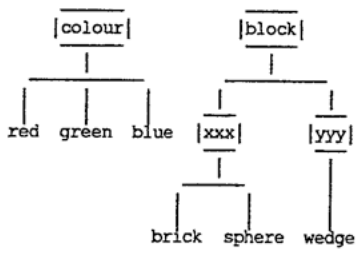
– The disjunctive-concept problem ----------

Unfortunately, it is often impossible to represent a body of data C in terms of a conjunctive concept (single-neighbourhood representation).





- TYPE 1 SITUATION: a set of positive elements P cannot be represented as a conjunctive concept (single-NR) if there is no ancestor node of the leaf-nodes featured in elements of P which is not also an ancestor of a leaf-node featured in an element of N (the negative elements). P can become representable if the structure of the data language is changed (e.g. by tree-hacking).



- TYPE 2 SITUATION: a set of positive elements P cannot be represented as a conjunctive concept if every element of P shares at least one leaf-node (coordinate) with an element of N.

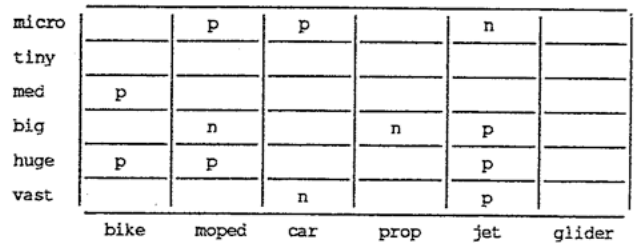
- (NB. There is an even more extreme situation which occurs when positive elements are represented in terms of *multiple* sets of leaf-nodes.)

- These two situations are both examples of what is often called the "disjunctive concept problem".

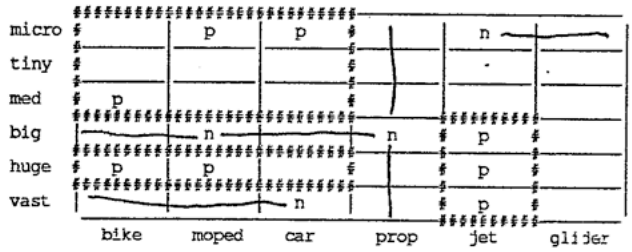
-- Multiple-neighbourhood representations ----------

```
vars data_elements =
  [ [micro moped ^true] [micro car ^true]
    [big moped ^false] [vast jet ^true]
    [big prop ^false] [huge bike ^true]
    [huge moped ^true] [vast car ^false]
    [huge jet ^true] [med glider ^false]
    [big jet ^true] [med bike ^true]
  ];
```

C = <positive instances>



There is no single-neighbourhood representation for C (type-1 DC problem) -- it is not possible to construct a rectangle in D which encloses all the positives but excludes all the negatives. But there *is* a multi-neighbourhood representation for C.



Disjunctive concept definition
[small vehicle] OR [huge vehicle] OR [large jet]

-- Learning disjunctive concepts using Focussing/CE ----------

There are various ways in which Focussing/CE can be adapted for the task of deriving disjunctive definitions:

- Run the algorithm on C several times so as to produce one component of the definition each time... But which subset of C should be processed in each run?

- Run CE just once and whenever it becomes impossible to expand the specific boundary without enclosing a negative element split the version space into two parts (two G/S pairs) and carry on.

This technique, attributed to Mitchell et al. (1983), is called "rule shell creation" The problem with it is that there is no infallible way to decide which version space (rule shell) a new positive element should be assumed to belong to. If a wrong decision is made, the desired definition will not be derived. (cf. Bundy et al. 1985, p. 167).

-- Multiple convergence ----------

A variant of shell creation called "multiple convergence" (Murray 1987) involves adding new positive elements to *all* version spaces. Subsequently, if any version-space is found to include a (new) negative element, it is specialised just enough to exclude the new element. This causes positive elements to be "purged".

-- Least disjunctions ----------

Note that at any given point in the presentation of new elements it is possible to construct a perfect disjunctive concept. This can take the form of a "least disjunction".

"A least disjunction is a disjunction of minimally specific descriptions in the language such that each disjunct covers as many positive elements as possible without covering any negative elements, and every positive element is covered by at least one of the disjuncts." (Utgoff, Shift of Bias, 1986, p. 124).

Geometrically, a least disjunction is just the set of neighbourhoods constructed when, wherever possible, we put a ~~minimal~~-size rectangle around a group of p's.

To construct a "maximal disjunction" we make the rectangles as large as possible; i.e. we make the disjuncts as general as possible while ensuring that they do not cover any negative elements.

-- The AQ algorithm ----------

Michalski 1975;

- Assumes all negative elements are available -- positives arrive as a sequence.
- Uses K passes of CE to derive K definitions (version spaces, conjunctive concepts, single-NRs etc.) covering a body of data which cannot be represented as a single-NR
- Uses a "seeding" heuristic -- each time it starts trying to build a new definition, it uses an element which has not been covered by any previous G as a seed. This helps to ensure that disjuncts are far apart in D.

See ../teach/course/eska/AQ.p

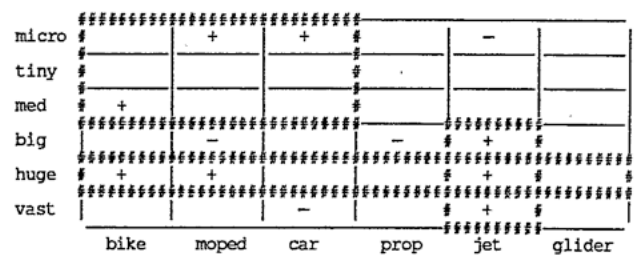
== AQ implementation ----------

```
/* AQ.p - implementation of Aq algorithm */

uses CE;

vars S_list G_list negative_elements positive_elements, myshowtree = erase;

define get_elements(type);
  return([^(foreach [== ^type] in data_elements do it endforeach)])
enddefine;

define setup;
  [  [micro moped ^true] [micro car ^true]
     [big moped ^false] [vast jet ^true]
     [big prop ^false] [huge bike ^true]
     [huge moped ^true] [vast car ^false]
     [micro jet ^false] [huge jet ^true]
     [big jet ^true] [med bike ^true]
  ] -> data_elements;
  get_elements(false) -> negative_elements;
  get_elements(true) -> positive_elements;
enddefine;

define covered_by_oneof(element, descriptions);
  vars description;
  for description in descriptions do
    if covers(description, element) then return(true) endif
  endfor;
  return(false);
enddefine;

define run;
  vars description;
  [] ->> S_list -> G_list;
  setup();
  until positive_elements = [] do
    CE([^(hd(positive_elements)) ^^negative_elements]) --> [?G ?S];
    G :: G_list -> G_list;
    [% for element in positive_elements do
        unless covered_by_oneof(element, G_list) do element endunless
       endfor %] -> positive_elements;
  enduntil;
  G_list ==>
enddefine;
```

== Sample run ----------

```
run();
> CE [[micro moped <true>] [big moped <false>] [big prop
  <false>] [vast car <false>] [micro jet <false>]]
< CE [[small vehicle] [micro moped]]
> covered_by_oneof [micro moped <true>] [[small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [micro car <true>] [[small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [vast jet <true>] [[small vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [huge bike <true>] [[small vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [huge moped <true>] [[small vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [huge jet <true>] [[small vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [big jet <true>] [[small vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [med bike <true>] [[small vehicle]]
< covered_by_oneof <true>
> CE [[vast jet <true>] [big moped <false>] [big prop <false>]
  [vast car <false>] [micro jet <false>]]
< CE [[vast plane] [vast jet]]
> covered_by_oneof [vast jet <true>] [[vast plane] [small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [huge bike <true>] [[vast plane] [small
  vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [huge moped <true>] [[vast plane] [small
  vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [huge jet <true>] [[vast plane] [small
  vehicle]]
< covered_by_oneof <false>
> covered_by_oneof [big jet <true>] [[vast plane] [small
  vehicle]]
< covered_by_oneof <false>
> CE [[huge bike <true>] [big moped <false>] [big prop
  <false>] [vast car <false>] [micro jet <false>]]
< CE [[huge transport] [huge bike]]
> covered_by_oneof [huge bike <true>] [[huge transport]
  [vast plane] [small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [huge moped <true>] [[huge transport]
  [vast plane] [small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [huge jet <true>] [[huge transport]
  [vast plane] [small vehicle]]
< covered_by_oneof <true>
> covered_by_oneof [big jet <true>] [[huge transport] [vast
  plane] [small vehicle]]
< covered_by_oneof <false>
> CE [[big jet <true>] [big moped <false>] [big prop <false>]
  [vast car <false>] [micro jet <false>]]
< CE [[big jet] [big jet]]
> covered_by_oneof [big jet <true>] [[big jet] [huge transport]
  [vast plane] [small vehicle]]
< covered_by_oneof <true>

** [[big jet] [huge transport] [vast plane] [small vehicle]]
```

== Neighbourhood representations ----------



Expert Systems & Knowledge Acquisition: 13

Chris Thornton -- February 18, 1989

-- DISCRIMINATION RULES ----------

Methods for constructing disjunctive definitions can also be used to learn *discrimination rules*.

Imagine

- C decomposes into K classes [redacted] C1, C2, C3 etc.
- we want a mechanism which will construct rules to decide which class an arbitrary data element is a member of.

A simple approach is to take each class in turn and learn a maximally general/specific conjunctive definition (a single-NR) of it by pretending that all the elements of all the other classes are negative elements.

By repeating the process for each class we will derive K definitions. Each one forms a classification rule (a classificatory, coupled NR) for the class in question.

-- Rule learning methods ----------

There are a number of variations on this theme

- learn maximally specific definitions (this produces rules which will provide a unique classification of a small number of elements)
- learn maximally general definitions (this produces rules which will produce a not necessarily unique classification of a large number of elements)
- learn both decide on which one to use at a later date.

- Another possibility is to learn maximally general definitions but to pretend that each learned definition is a negative element (a negative concept?). This will ensure that a unique classification is always generated. However, the order in which definitions are learned will make a big difference (see HB3, p. 424-425).

An extension of AQ called AQ11 constructs "discrimination rules" using all four methods described (cf. HB3, pp. 423-427).

See .../teach/course/cska/AQ11.p

== AQ11 Implementation ----------

```
/* AQ11.p - implementation of AQ11 algorithm */


vars classes rules G_list S_list;
uses CE;

define setup;
  set_trees();
  [
    [class1
      [
        [med bike ^true]
        [micro moped ^true]
        [micro car ^true]
      ]]
    [class2
      [
        [vast jet ^true]
        [huge jet ^true]
        [big jet ^true]
      ]]
    [class3
      [
        [huge bike ^true]
        [huge moped ^true]
      ]]
    ] -> classes;
  [] ->> rules ->> G_list ->> S_list -> rules;
enddefine;



define negative_of(description);
  vars x;
  description matches [??description ?x:isboolean] ->;
    return([^description ^false]);
enddefine;

define form_negative_elements_from_other_classes(class);
  vars other_class elements result;
  [% foreach [?other_class ?elements] in classes do
      if other_class /= class do applist(elements, negative_of) endif
    endforeach %] -> result;
  return(result);
enddefine;

define run;
  vars class, rule elements other_classes negatives;
  setup();
  foreach [?class ?elements] in classes do
    form_negative_elements_from_other_classes(class) -> negatives;
    CE([^elements ^(applist(G_list, negative_of)) ^^negatives]) --> [?G =];
    G :: G_list -> G_list;
    [[^class rule ^G] ^^rules] -> rules;
  endforeach;
  rules ==>
enddefine;
```

== Sample run ----------

Setting up involves stating which positive instances are in which classes.

```
define setup;
  set_trees();
  [
    [class1
      [
        [med bike ^true]
        [micro moped ^true]
        [micro car ^true]
      ]]
    [class2
      [
        [vast jet ^true]
        [huge jet ^true]
        [big jet ^true]
      ]]
    [class3
      [
        [huge bike ^true]
        [huge moped ^true]
      ]]
    ] -> element_lists;
  [] ->> S_list -> rules;
enddefine;
```

== Learning a rule for class1 ----------

```
run();
> form_negative_elements_from_other_classes class1
!> negative_of [vast jet <true>]
!< negative_of [vast jet <false>]
!> negative_of [huge jet <true>]
!< negative_of [huge jet <false>]
!> negative_of [big jet <true>]
!< negative_of [big jet <false>]
!> negative_of [huge bike <true>]
!< negative_of [huge bike <false>]
!> negative_of [huge moped <true>]
!< negative_of [huge moped <false>]
< form_negative_elements_from_other_classes [[vast jet
  <false>] [huge jet <false>] [big jet <false>] [huge
  bike <false>] [huge moped <false>]]
> CE [[med bike <true>] [micro moped <true>] [micro car
  <true>] [vast jet <false>] [huge jet <false>] [big
  jet <false>] [huge bike <false>] [huge moped <false>]]
< CE [[small transport] [small vehicle]]
```

== Learning a rule for class2 ----------

```
> form_negative_elements_from_other_classes class2
!> negative_of [med bike <true>]
!< negative_of [med bike <false>]
!> negative_of [micro moped <true>]
!< negative_of [micro moped <false>]
!> negative_of [micro car <true>]
!< negative_of [micro car <false>]
!> negative_of [huge bike <true>]
!< negative_of [huge bike <false>]
!> negative_of [huge moped <true>]
!< negative_of [huge moped <false>]
< form_negative_elements_from_other_classes [[med bike
```

<false>] [micro moped <false>] [micro car <false>]
[huge bike <false>] [huge moped <false>]]
> negative_of [small transport]
< negative_of [small transport <false>]
> CE [[vast jet <true>] [huge jet <true>] [big jet <true>]
[small transport <false>] [med bike <false>] [micro
moped <false>] [micro car <false>] [huge bike <false>]
[huge moped <false>]]
< CE [[large plane] [large jet]]

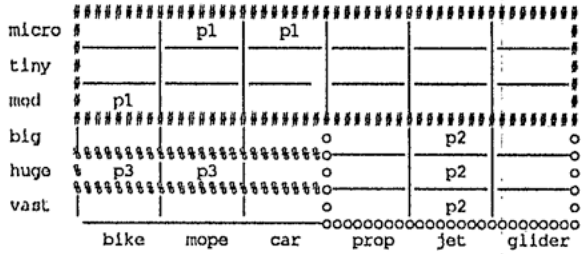
== Learning a rule for class3 ----------

> form_negative_elements_from_other_classes class3
!> negative_of [med bike <true>]
!< negative_of [med bike <false>]
!> negative_of [micro moped <true>]
!< negative_of [micro moped <false>]
!> negative_of [micro car <true>]
!< negative_of [micro car <false>]
!> negative_of [vast jet <true>]
!< negative_of [vast jet <false>]
!> negative_of [huge jet <true>]
!< negative_of [huge jet <false>]
!> negative_of [big jet <true>]
!< negative_of [big jet <false>]
< form_negative_elements_from_other_classes [[med bike
<false>] [micro moped <false>] [micro car <false>]
[vast jet <false>] [huge jet <false>] [big jet <false>]]
> negative_of [large plane]
< negative_of [large plane <false>]
> negative_of [small transport]
< negative_of [small transport <false>]
> CE [[huge bike <true>] [huge moped <true>] [large plane
<false>] [small transport <false>] [med bike <false>]
[micro moped <false>] [micro car <false>] [vast jet
<false>] [huge jet <false>] [big jet <false>]]
< CE [[huge vehicle] [huge vehicle]]

** [[class3 rule [huge vehicle]]
[class2 rule [large plane]]
[class1 rule [small transport]]]

== Neighbourhood representation ----------

AQ11 constructs a coupled, classificatory NR for each subclass of C.



define cN(d, class);
vars rule class;
if [^class = ?rule] isin rules then
return(covers(rule, d));
else
return(false)
endif;
enddefine;

cN([big jet], "class1") ==>
** <false>

cN([big jet], "class2") ==>
** <true>

== Reading ----------

Utgoff, 1986; Murray 1987; Bundy et al. 1985; -- all discuss ways in which disjunctive definitions can be used to guide the introduction of new structure in the data language.

Michalski & Chilausky 1980; - how AQ11 learned rules for diagnosing soybean diseases.

Michalski & Stepp, 1983; Lebowitz UNIMEM 1986; Lebowitz Concept Learning 1986; Lebowitz 1987; -- all discuss conceptual clustering systems.

IIB3, pp. 398-400 & 423-427; -- discussion of AQ and AQ11.

--- Questions ----------

Why is the order of element presentation important?

What difference does it make if data elements correspond to multiple sets of leaf-nodes; i.e. if a data element has the form

[[large bike][small bike]]

Expert Systems & Knowledge Acquisition: 14

Chris Thornton -- February 25, 1989

-- LEARNING BY OBSERVATION ----------

All the learning methods looked at so far perform "learning from examples". They take sequences of positive and negative instances of classes (usually just one class) and produce definitions of the corresponding concepts. They cannot work without the help of a "teacher" or "oracle".

Ideally, we would like to have learning methods which can derive concept definitions without this sort of assistance.

-- Supervised v. unsupervised learning ----------

Learning with the help of a teacher is called "supervised learning". Learning without the help of a teacher is called "unsupervised learning" and and sometimes "learning by observation" (LBO).

How can we get a program to do unsupervised learning?

Note that all the learning-from-examples (LFE) methods we have looked at assume that the instances covered by a given concept will be *similar* to one another. LFE programs produce definitions in various different ways but these are effectively just the different ways of capturing instance-similarities in a general rule.

-- Definitions produced in similarity-based learning -

- Classical concepts are based on the idea that the instances in a class are similar in the sense that they all have certain features.
- Decision trees are based on the idea that the instances in a class can be split up into subgroups whose members are similar in that they all have certain features.
- Rules (generalised definitions) are based on the idea that the instances in a class are similar in the sense that their features are all special cases of the (generalised) features appearing in the rule.

Thus the techniques we have looked at are all based on the same idea -- that the instances covered by a given class are somehow similar to one another.

-- Numerical taxonomy ---------- "clustering"

The idea that concepts capture similarities between instances leads fairly directly to a method for deriving concepts automatically without the help of a teacher.

Given a body of data D1 divide it up into subset of similar elements. Then produce a concept covering each of the subsets.

The process can be recursively applied to the set of concepts produced. The end product is then a hierarchy of concepts; i.e. a taxonomy.

Question: how does a taxonomy differ from a decision tree?

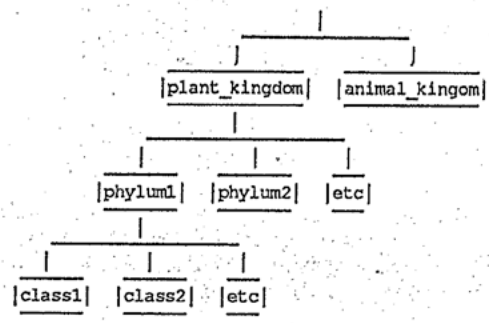
-- Cluster analysis ----------

"The problem of automatically creating such a [class] hierarchy has so far received little attention in AI. Yet creating classifications is a very basic and widely practiced intellectual process.

Past work on this problem was done mostly outside AI under the headings of numerical taxonomy and cluster analysis (Anderberg, 1973). Those methods are based on the application of a mathematical measure of similarity between objects, defined over a finite, a priori given set of object attributes. Classes of objects are taken as collections of objects with high intraclass and low interclass similarity. The methods assume that objects are characterized by sequences of attribute/value pairs and that this information is sufficient for creating a classification.

Numerical taxonomy is used mainly by natural scientists who require classificatory systems for sets of natural objects.

The basic principles guiding biological classification were laid down by Linnaeus but these are now under attack from some quarters.



-- Hierarchical (agglomerative) clustering ----------

Set G to be the set of singleton sets such that each set contains a unique data element. Then

Until |G| = 1 do

- Form a matrix of similarity values for all elements of G (using some given metric -- see below)
- Merge those groups in G which have a maximum similarity value.

The algorithm is very straightforward. However, the question of how to compute the similarity values is a complex one.

There are two issues:

- how should we work out the similarity of two distinct data elements?
- how should we work out the similiarity of two groups of data elements.

-- Euclidean and Manhattan distances ----------

If data elements are just vectors of numbers, then we can measure their similarity by measuring their closeness (inverse distance) in the data space. This can done using, e.g., a euclidean metric or a city-block metric.

Euclidean distance $d1/d2\ \sqrt{\Sigma (d1_i - d2_i)^2}$

City-block (manhattan) distance $\Sigma |d1_i - d2_i|$

If we think about the data elements as points in a geometric space, then their euclidean distance is just the same thing as their "real" distance.

-- Single-linkage and complete-linkage methods --------

The simplest methods for computing the similarity of two groups are the single-linkage method and the complete-linkage method.

In single-linkage, the distance between two groups is defined as the distance between their two closest points.

In complete-linkage, the distance between two groups is defined as the distance between their two most distant points.

In the centroid method, the distance between two groups is defined as the distance between their centroids.

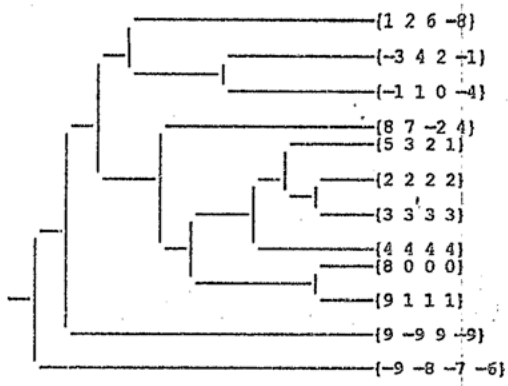
The naive approach of computing group distance in terms of the average of the distances for all pairs in G1 × G2 is computationally expensive and not often used. (It is often called the group-average method.)

There are many other methods (see [Vogt, Nagel & Sator, 1987, pp. 29-36]).

== Example ----------

Using LIB CLUSTER (see SHOWLIB CLUSTER.P)

```
lib cluster;

vars data =
  {{1 2 6 -8}{-3 4 2 -1}{8 7 -2 4}{-1 1 0 -4}
  {5 3 2 1}{-9 -8 -7 -6} {2 2 2 2}{3 3 3 3}
  {4 4 4 4}{9 -9 9 -9}{8 0 0 0}{9 1 1 1}};

showdendro(cluster(data));
```



dendrogram.

-- Conceptual clustering ----------

Continuation of quote from Stepp & Michalski.

"... The [clustering] methods do not take into consideration and *background knowledge* about the semantic relationships among object attributes or global concepts that could be used for characterizing object configurations. Nor do they take into consideration possible goals of classification that might be indicated by background knowledge.

As a result, classifications obtained by traditional methods are often difficult to interpret conceptually." [Stepp and Michalski 1986, p. 472].

These ideas have lead a number of researchers to experiment with an extension of numerical taxonomy called "conceptual clustering".

The general aim in conceptual clustering is the same as in numerical taxonomy: to derive [illegible]archical classification. But in conceptual clustering there is the additional aim that [illegible]node in the hierarchy should form a coherent concept.

Many techniques have been developed. Most of them involve arranging things such that the construction of new groups (clusters) takes into account both their syntactic similarity and the degree to which their union forms a good concept.

== Reading ----------

An excellent survey article on conceptual clustering is [Fisher & Langley 1985] -- this is a paper in IJCAI9.

The classic references for clustering and numerical taxonomy are [Everitt 1974] and [Anderberg 1973].

Another reference of interest is [Fisher 1987].

On the wider epistemological status of clustering see [Good, 1977].

A very accessible article is [Michalski & Stepp 1983] and the followup in ML2 [Michalski & Stepp 1986].

For a mickey-mouse introduction to cluster analysis see [Romesburg 1984].

[Sokal 1977] is an authoritative survey article on clustering and classification.

Expert Systems & Knowledge Acquisition: 15

Chris Thornton -- February 25, 1989

-- UNIMEM ----------

Conceptual clustering is a variant of the hierarchical clustering process in which the formation of new groups takes into account both their syntactic similarity and the degree to which their union will form a "good" concept.

"good" can mean a number of different things, e.g.

- compact
- monothetic (or minimally polythetic) [red, brick] one attribute identified common
- easily represented in terms of a definition or rule which has a natural interpretation for humans
- easily transformed into a representation which can play some kind of role in the achievement of goals

This lecture will look at Lebowitz's conceptual clustering algorithm called Unimem. All unlabelled references are to [Lebowitz 1987].

-- Generalization-based memory ----------

Normally, in conceptual clustering, we (1) initialise a set of singleton groups from the set of data elements and then (2) repeatedly merge similar groups so as to produce good concepts at each stage. Unimem does something like this.

"The task of Unimem is to take a series of examples (or *instances*) that are expressed as collections of features and build up a generalization hierarchy of concepts." (pp.103-104).

However, it does not do any explicit clustering. Instead it attempts to process new instances by updating an existing hierarchy of concepts, which in Unimem is called the generalization-based memory or GBM. Updating the GBM involves adding in the new instance at some point and updating existing nodes as necessary.

"The use of a hierarchy of generalizations as a method of memory organization allows efficient storage of information since it supports inheritance. In addition, GBM allows the generalizations and instances relevant for learning to be found efficiently in memory ..." (p. 109).

-- The instance-processing loop in Unimem ----------

- Search GBM depth-first for the most specific concept node(s) that the instance matches (lowest) (covers)
- Add the new instance to memory at or below this node. This involves comparing the new instance to the ones already stored there and generalizing if appropriate.

(Adapted from pseudo-code algorithm on p. 110)

"As Unimem searches down the concept hierarchy, features are gradually accounted for by various generalizations."

== Creating new concepts ----------

.y point -

- if the sum of the distances between the currently unaccounted-for features and those of the current node is too great, then stop exploring this branch.
- Otherwise, explore relevant children.

The search process returns all the most specific nodes which explain (cover) the new instance.

Unimem then generalises each node so as to account for (cover) the new instance. It then compares the new instance with all the other instances stored at each of the returned nodes. If it finds an instance which has enough features in common with the new instance, it creates a new subnode by generalizing the common features and it stores both instances at the subnode.

Unimem includes lots of kludges for making the whole process work as well as possible (although Lebowitz does a fine job in making them sound like principled features). It also requires 13 parameters to be set by the user in any application.

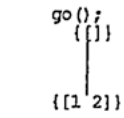
== Flag data ----------

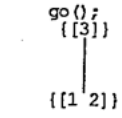
```
[
/*  Flag      Stars Bars Strp Hues Xcrs Icon Hmns Word Num Type */
 [Alabama      0     0    0    3    1    0    0    0    0   C]
 [Arkansas    29     0    0    3    0    0    0    1    0   C]
 [Connecticut  0     0    0    5    0    1    0    4    0   U]
 [Delaware     0     0    0    6    0    1    2    4    2   U]
 [Florida      0     0    0    6    1    1    1   15    0   C]
 [Georgia     13     1    0    3    1    1    0    3    1   C]
 [Illinois     0     0    0    6    0    1    0    6    2   U]
 [Iowa         0     2    0    5    0    1    0   10    0   U]
 [Lousiana     0     0    0    4    0    1    0    4    0   C]
 [Maryland     0    12    0    4    0    1    0    0    0   U]
 [Masachusetts 1     0    0    4    0    1    1    6    0   U]
 [Mississippi 13     0    3    3    1    0    0    0    0   C]
 [New_Hants    9     0    0    5    0    1    0    7    1   U]
 [New_Jersea   0     0    0    5    0    1    2    3    1   U]
 [New_York     0     0    0    6    0    1    2    1    0   U]
 [N_Carolina   1     1    2    4    0    0    0    3    4   C]
 [Ohio        17     0    5    3    0    0    0    0    0   U]
 [Rhode_Is    13     0    0    3    0    1    0    1    0   U]
 [S_Carolina   0     0    0    2    0    1    0    0    0   C]
 [Tennesee     3     2    0    3    0    0    0    0    0   C]
 [Texas        1     1    2    3    0    0    0    0    0   C]
 [Virginia     0     0    0    5    0    1    2    4    0   C]
 [Wisconsin    0     0    0    5    0    1    2    2    1   U]
 [Washington   3     0    5    2    0    0    0    0    0   U]
 ];
```

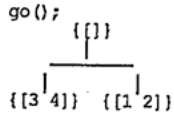
== Toy implementation ----------

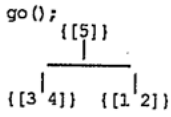
```
/* unimem.p - basic implementation of Lebowitz' Unimem learning algorithm */

/*
Data structures:

<concept_hierarchy> ::= [ <node> <subnode>* ]
<subnode> ::= <node> | <concept_hierarchy>
<node> ::= { [ <instance>* ] }
<instance> ::= <index:integer>
<feature_vector> ::= [ <feature>* ]
<feature> ::= { <index:integer> <minval:number> <maxval:number> }

*/

uses showtree;

vars
  coverage_threshold,
  generalisation_threshold,
  concept_hierarchy,
  input_data,
  input_instances,
  feature_vectors,
  number_of_features,
  feature_maxvals,
  feature_minvals,
  subnodes = tl,
  instance = false,
  ;

define instances_at(node); hd(node)(1) enddefine;
define updaterof instances_at(node); -> hd(node)(1) enddefine;

define addup(numbers);
  if numbers = [] then 0 else hd(numbers) + addup(tl(numbers)) endif
enddefine;

define feature_distance(f, g);
  ((if f(2) >= g(2) then 0 else g(2) - f(2) endif)
    + (if f(3) <= g(3) then 0 else f(3) - g(3) endif))
  / (feature_maxvals(f(1)) - feature_minvals(f(1)))
enddefine;

define appfeatures(unexplained_features, features, pdr);
  vars feature, feature_index, unexplained_feature, distances;
  [ % for feature in features do
      feature(1) -> feature_index;
      for unexplained_feature in unexplained_features do
        if unexplained_feature(1) = feature_index then
          pdr(feature, unexplained_feature)
        endif
      endfor
    endfor % ]
enddefine;

define sum_of_distances(features, unexplained_features);
  addup(appfeatures(unexplained_features, features, feature_distance))
enddefine;

define return_if_explained_by(feature, generalisation);
```

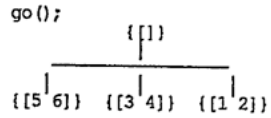
```
    if feature_distance(feature, generalisation) = 0 then feature endif
  enddefine;

  define still_unexplained_features(database, features) -> database;
    applist(
      appfeatures(database, features, return_if_explained_by), flush);
  enddefine;

  define search(node, unexplained_features) -> nodes;
    vars subnode;
    if (sum_of_distances(features(node), unexplained_features)
        / number_of_features) > coverage_threshold then
      [] -> nodes
    else
      still_unexplained_features(unexplained_features, features(node))
        -> unexplained_features;
      [ % for subnode in subnodes(node) do
          explode(search(subnode, unexplained_features))
        endfor % ] -> nodes;
      if nodes = [] then [[^node ^unexplained_features]] -> nodes; endif;
    endif
  enddefine;

  define generalisation(feature_vector1, feature_vector2);
    vars f1, f2;
    [ % for f1 in feature_vector1 do
        for f2 in feature_vector2 do
          if f1(1) = f2(1)
          and feature_distance(f1,f2) < coverage_threshold then
            [ % f1(1), min(f1(2), f2(2)), max(f1(3), f2(3)) % ]
          endif
        endfor;
      endfor % ]
  enddefine;

  define form_generalisation(instance, new_instance) -> node;
    node = [[[^instance ^new_instance]]];
    generalisation(feature_vectors(instance),
      feature_vectors(new_instance))
      -> features(node);
  enddefine;

  define update(node, new_instance, unexplained_features) -> node;
    vars instance, added_subnode = false;
    for instance in instances_at(node) do
      if (sum_of_distances(feature_vectors(instance),
            feature_vectors(new_instance))
          / number_of_features) < generalisation_threshold then
        form_generalisation(instance, new_instance) :: tl(node) -> tl(node);
        delete(instance, instances_at(node)) -> instances_at(node);
        true -> added_subnode;
      endif;
    endfor;
    if not(added_subnode) then /* add instance to current node */
      new_instance :: instances_at(node) -> instances_at(node);
    endif;
  enddefine;

  define process_new_instance(instance);
    vars node, nodes, unexplained_features;
    search(concept_hierarchy, feature_vectors(instance)) -> nodes;
    foreach [?node ?unexplained_features] in nodes do
      update(node, instance, unexplained_features) ->;
    endforeach;
  enddefine;


  define setup; /* flag data example */
    vars values, i, value;
    0.075 -> coverage_threshold;
    0.75 -> generalisation_threshold;
    compile('flag_data.p') --> [??input_data:12 ==];
    input_data(1) --> [= ??values =];
    length(values) -> number_of_features;
    newproperty([],16,[],false) -> features;
    newproperty([],16,999999,false) -> feature_minvals;
    newproperty([],16,-999999,false) -> feature_maxvals;
    [[[]]] -> concept_hierarchy;
    [ % foreach [= ??values =] in input_data do
        [ % for i from 1 to number_of_features do
            values(i) -> value;
            min(feature_minvals(i), value) -> feature_minvals(i);
            max(feature_maxvals(i), value) -> feature_maxvals(i);
            [ % i, value, value % ]
          endfor % ]
      endforeach % ] -> feature_vectors;
  [ % for i from 1 to length(feature_vectors) do i endfor % ] -> input_instances;
  enddefine;

  define go;
    if input_instances matches [?instance ??input_instances] then
      process_new_instance(instance);
      showbaretree(concept_hierarchy);
    else
      [finished] =>
    endif;
  enddefine;
  /*
  - */
  define run;
    setup();
    until input_instances = [] do go() enduntil;
  enddefine;

  == Worked example ----------------------------------

  setup();
  go();
  [[1]]
```

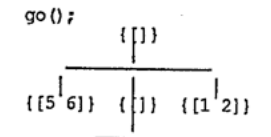




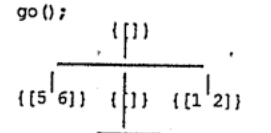


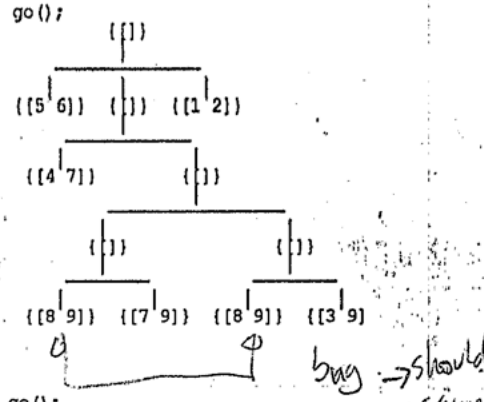




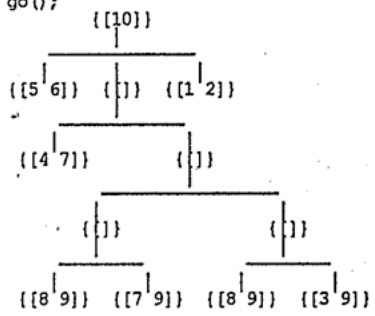


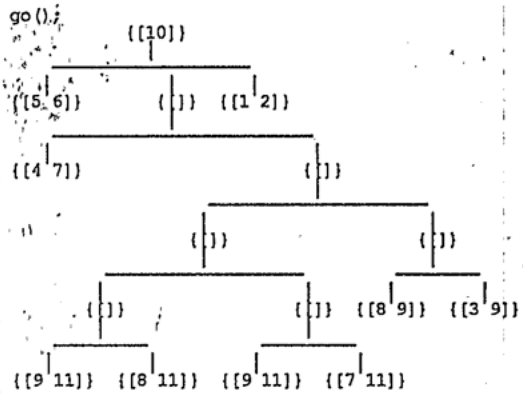
can be one node depending on thresholds

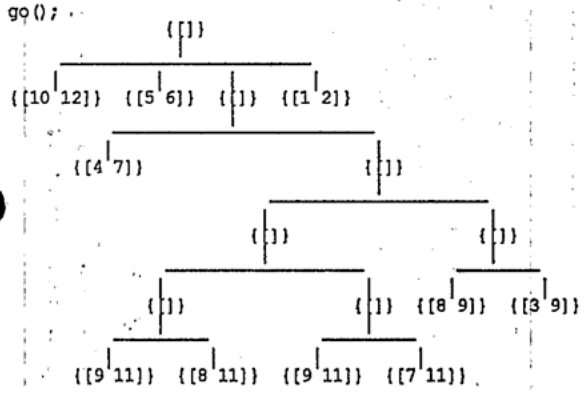




bug → shouldn't create same node twice







go();
** [finished].

--- Reading ----------

The best reference for Unimem is [Lebowitz 1987] but see also [Lebowitz, An Overview, 1986] and [Lebowitz, Experiments 1987].

Chris Thornton -- March 4, 1989

-- EXPLANATION-BASED LEARNING ----------

Symbol Processing

Explanation-based Learning (EBL) is currently one of the most active research areas in SP learning research.

Some people and places
DeJong
University of Illinois at Urbana-Champaign
Carbonell, Minton
Carnegie-Mellon
Hirsh
Stanford

An important early paper is [Mitchell, Keller and Kedar-Cabelli 1986]. The most accessible paper is (currently) the one by DeJong in ML2.

== Background ----------

Most of the SP and PDP learning mechanisms we have looked at so far have used no built-in knowledge. They apply some domain-independent algorithm to input data so as to derive representations of the environment from which the data are drawn. People have noted that there are at least two problems with this approach.

- In the 70s and 80s, AI researchers have tended to place increasing emphasis on the role of *background knowledge* in cognition. Lenat has christened the notion that AI systems cannot hope to perform well without lots of background knowledge the "knowledge principle"; and in Winston's text book [Winston 1984], the idea that the knowledge principle will always have (negative) impact on learning mechanisms takes the form of "Martin's law" - you ca[redacted]n something unless you almost know it already.

- Researchers have noted that human's frequently learn how to solve a problem by observing and understanding how some other human solves it. This seems to suggest that powerful learning (e.g. the sort exhibited by humans) may depend on being able to construct *explanations* for the behaviours of other cognitive mechanisms.

-- EBL in theory ----------

Over the last few years, learning researchers who are either (a) very committed to the knowledge principle, or (b) very interested in mechanisms which learn how to solve problems (or both) have been developing the approach called Explanation-Based Learning (or sometimes Explanation-Based Generalization).

Ideally, an EBL system is a program which learns how to solve some problem by (a) observing strategies by which the problem can be solved and (b) constructing explanations about these strategies, and (c) modifying its own behaviour with reference to these explanations. In practice, the behaviour of EBL systems is a rather specialised version of this ideal.

-- EBL in practice ----------

There is not much point in a learning mechanism which can only learn to solve some problem if it needs to watch a program which can *already* solve it. EBL systems therefore generate and observe their own (flawed) solutions to problems.

There is no robust definition of "explanation" as yet. So the explanation used by an EBL s[redacted]is typically just the search space generated by the problem solving process.

The behaviour modifications which are introduced with reference to the explanations are typically just alterations in the heuristic which guides the problem solving search.

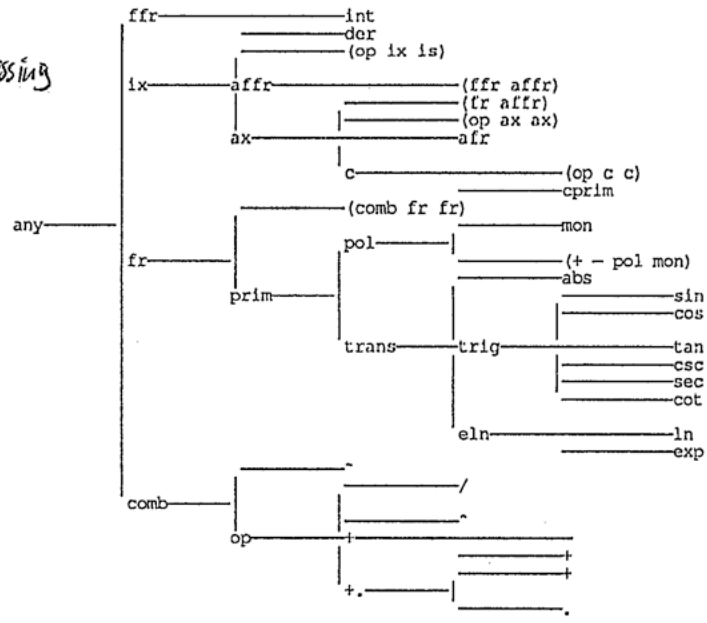
"Almost all EBL problem-solvers learn by analyzing why a solution succeeds in solving a proble" (Minton and Carbonell, 87)

-- The LEX system ----------

The EBL framework was given its first (?) clear articulation in the form of the LEX system. This system is built around a problem solver which solves problems in symbolic integration. Solving a symbolic integration problem involves performing a series of syntactic manipulations on a mathematical expression so as to get rid of an integration sign (cf. performing syntactic manipulations on an equation so as to get all the unknowns on one side).

To understand how LEX works it is important to know that the mathematical expressions in question form a data language which can be defined using a single generalization hierarchy.

== A description language for mathematical expressions



See (Utgoff, Shift of Bias, 1986), p.121 for the complete grammar.

In the data language defined by this generalisation hierarchy, data elements are N-tuples of leaf-nodes - but N is not fixed. Thus a data element might be something like

x + y

or

sin(x + y)

In fact, in the implementation of LEX expressions were written in a standard (LISP) notation so as to make the problems involved in dealing with arbitrary sized tuples less severe (see Utgoff, Shift of Bias, 1986).

== Integration operators ----------

The problem solver component of LEX accepted an integration expression as input and attempted to find a way of manipulating so as to get rid of the integration sign.

To do this it used a set of operators each one of which defined a particular manipulation; i.e. showed how an expression of some paticular form can be simplified.

OP02 (power rule)
convert $\int x^r\,dx$
to $x^{r+1} / (r + 1)$

OP03 (factor out a real constant)
convert $\int rf(x)\,dx$
~~$r\int f(x)\,dx$~~ $r \int f(x)$

OP06
convert $\int \sin x\,dx$
to $-\cos x$

OP08
convert $1 . f(x)$
to $f(x)$

OP10
convert $\int \cos x\,dx$
to $\sin x$

OP12 (integration by parts)
convert $\int u\,dv$
to $uv - \int v\,dv$

OP15
convert $0 . f(x)$
to 0

4 parts:
- solve prob
- look at trace
- Modify way in which tries similar probs start

== The problem space ----------

In any given situation, the problem solver in LEX has to select one of the applicable operators and apply it. The set of applicable operators corresponding to some given expression form the child nodes for the node corresponding to the expression in the search space which is explored by the problem solver.

In LEX each operator is associated with some information which suggests whether the operator is *appropriate* in a particular case. The problem solver just tries the appropriate operators rather than all possible operators (i.e. it executes a huèristic search).

-- Version space heuristics ----------

This information about appropriateness is actually just a version space definition (a G and an S).

Recall that a version space identifies a subset of the possible data elements (i.e. a subspace in D). Thus the appropriate cases in which an operator can be defined as those data elements which are in the version space associated with the operator.

Consider the operator

INT sin(x) dx ==> -cos(x) + C

Typical heuristic (represented in English)

"IF the problem state contains an integrand which is the product of x and any transcendental function of x, THEN try integration by parts, with u and dv bound to the indicated subexpression"

Represented as a version-space

S: {INT 3x cos(x) dx ==> apply OP2}
G: {INT f1(x) f2(x) dx ==> apply OP2}

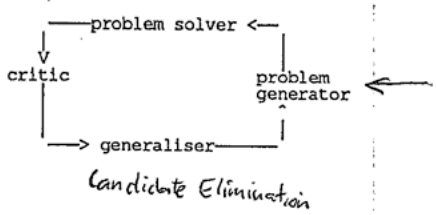
where "f1" and "f2" denote "some function"

== Another example ----------

G = ∫ f(x) g(x) dx ==> OP12, with u = f(x) and dv = g(x) dx

S = ∫ 3x cos x dx ==> OP12, with u = 3x and dv = cos x dx

-- Processing cycle in LEX ----------



Candidate Elimination

Initially, LEX has a store of integration operators. Each one is associated with a "null" heuristic; i.e. a version space which includes just the expression featured in the operator.

It then enters a loop in which it

- generates an integration problem. (Problem generator)

- solves by exploring the space of possible operations using the version spaces associated with each operator to decide whether it is appropriate in any given case. (Problem solver)

- extracts from the trace of the search space generated all the operator applications lying *on* the path to the solution node and all the operator applications lying *off* the path. (Critic)

- updates each version space using the Candidate-Elimination algorithm so as to ensure that each one covers any applications (of the associated operator) which are on the solution path and excludes any applications which lead away from it. (Generalizer)

-- Problems with EBL ----------

EBL is a knowledge-intensive learning mechanism which is capable of producing very impressing learning performance. But the secret of its success is also its achilles heel. It produces impressive performance by exploiting the background knowledge (domain theory) which is pre-programmed in to the system. But if this background knowledge is in some way flawed (if the domain theory used is not perfect) then the learning process may go wildly off-track. This dependence on a perfect domain theory is regarded as one of the major problems with EBL.

== Reading ----------

Dejong, An Approach, 1986;

Expert Systems & Knowledge Acquisition: 17

Chris Thornton -- March 4, 1989

-- THE PROBLEM OF ABSTRACT CLASSES ----------

All the learning mechanisms we have looked at assume that

- there is some description or data language D
- any concept which can be defined in terms of D can be defined in terms of a *subset* of D.

In other words, they make the assumption that any concept which can be defined in terms of some set descriptions covers a subset of those descriptions.

This assumption turns out to be invalid.

-- Abstract v. concrete concepts ----------

D = the set of pairs which describe playing cards

{ [3, hearts] [queen, diamonds] [9, spades] ... }

R is the representation language; i.e. the language in which instances are represented.

R = D

C is the class of all black cards. A concept covering C can be selectively defined; e.g.

[redacted]isblack(card);
matches [== spades] or card matches [== clubs]
enddefine;

Contrast this with the situation where R is the powerset of D and C is the class of all "straights". A concept covering this class has no selective definition on D.

Concepts which have selective definitions on some D are said to be "concrete" (with respect to D). Concepts which can be defined in terms of elements of D but which have no selective definition on D are said to be "abstract" (with respect to D).

-- Very abstract concepts ----------

C is the class of all "close straights". An instance of this class is a set of N hands such that they are all straights with fairly equal overall values. This class can be described in terms of playing cards but clearly not in terms of a subset of playing cards. Thus it is an abstract class with respect to D. The question is: how abstract?

Imagine D' is the set of all descriptions of complete poker hands (5-tuples on D). The class of all straights is abstract with respect to D but concrete with respect to D'. But the class of all close straights is abstract with respect to both. This seems to suggest that, in some sense, the class of close straights is *more* abstract with respect to D than the class of all "straights".

We need a way of defining classes (i.e. of building concepts) which puts all this on a firm footing.

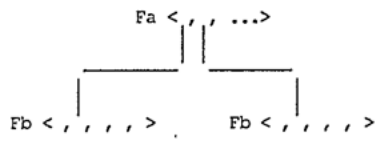
-- Generalised concepts ----------

[redacted]
- *A definition of class C* is a pair containing an n-place function and an n-tuple. The elements of the n-tuple are either all class definitions or all members of D. The inputs to the function are the outputs of the functions (or the elements of D) which appear in the n-tuple. The function produces <false> plus a range of other outputs all of which are interpreted as <true>.

- A class definition whose n-tuple contains only descriptions is of *order* 1. Otherwise, if the n-tuple contains a class definition of order m, the definition is of at least order m+1.

- The arity of a definition is just the arity of its function.

== A definition of STRAIGHT ----------



$$Fb(T) = \begin{cases} \text{value}(T) \text{ if } T \text{ forms a straight} \\ \text{false otherwise} \end{cases}$$

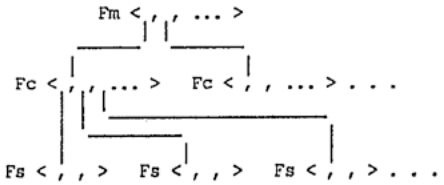
$$Fa(T) = \begin{cases} 1 \text{ if } \max(T) - \text{average}(T) > \text{threshold} \\ \text{false otherwise} \end{cases}$$

-- Higher-order classes? ----------

Class *definitions* always have a specific order and arity. But does the fact that a class has a nth-order, m-place definition say anything about the class itself?

For any extensional characterisation of a class (i.e. any enumeration of its instances) there will be a range of possible definitions of that class amongst which there will, presumably, be a *minimum* observed order and arity. So, define the degree of abstractness of a given class with respect to some given set of basic objects in terms of the minimum order and arity of a definition of that class.

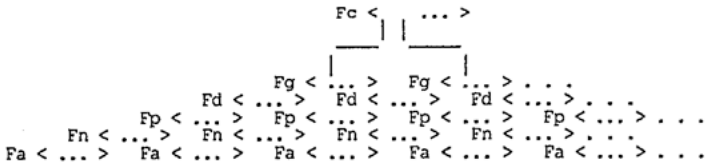
== A definition of SHOPPING-MALL ----------



- D is a set of (descriptions of) blocksworld objects (a set of triples of the form <name colour location>).
- R is the powerset of D.
- C is the class of all "shopping-malls".

- Fs is a function which takes a subset of D and returns a "structure number" (1 for an arch, 2 for a tower, etc.) if the elements form a structure and false otherwise.
- Fc is a function which takes n structure numbers and returns a "configuration number" (1 for a house, 2 for an arcade etc.) if they form a configuration and false otherwise.
- Fm is a function which takes n configuration numbers and returns tue if they are all 2s, false otherwise.

== A definition of CHROMOSOME ----------



- D is the set of elementary particles (the set of descriptions of elementary particles). R is the powerset of D, and c is the concept CHROMOSOME
- Fa is a function which takes a subset of D and returns an atomic number if the corresponding particles form an atom, and false otherwise.
- Fn is a function which takes n atomic numbers and returns a "nucleotide number" (i.e. a label for a nucleotide) if the corresponding atoms form a nucleotide, and false otherwise.
- Fp is a function which takes n nucleotide numbers and returns a "polynucleotide number" if the corresponding nucleotides form a polynucleotide, and false otherwise.
- Fd is a function which takes n polynucleotide numbers and returns a "DNA number" if the corresponding polynucleotides form a valid strand of DNA, and false otherwise.
- Fg is a function which takes n DNA numbers and returns a "gene number" if the corresponding strands of DNA form a valid gene, and false otherwise.
- Fc is a function which takes n gene numbers and returns true if the corresponding genes form a valid chromosome, and false otherwise.