11. Learning

What is Learning?

Learning is an important area in AI, perhaps more so than planning.

- Problems are hard -- harder than planning.
- Recognised Solutions are not as common as planning.
- A goal of AI is to enable computers that can be taught rather than programmed.

Learning is a an area of AI that focusses on processes of self-improvement.

Information processes that improve their performance or enlarge their knowledge bases are said to *learn*.

Why is it hard?

- Intelligence implies that an organism or machine must be able to adapt to new situations.
- It must be able to learn to do new things.
- This requires knowledge acquisition, inference, updating/refinement of knowledge base, acquisition of heuristics, applying faster searches, *etc*.

How can we learn?

Many approaches have been taken to attempt to provide a machine with learning capabilities. This is because learning tasks cover a wide range of phenomena. Listed below are a few examples of how one may learn. We will look at these in detail shortly

- Skill refinement ○ one can learn by practicing, *e.g playing the piano*.
- Knowledge acquisition ○ one can learn by experience and by storing the experience in a knowledge base. One basic example of this type is rote learning.
- Taking advice ○ Similar to rote learning although the knowledge that is input may need to be transformed (or *operationalised*) in order to be used effectively.
- Problem Solving ○ if we solve a problem one may learn from this experience. The next time we see a similar problem we can solve it more efficiently. This does not usually involve gathering new knowledge but may involve reorganisation of data or remembering how to achieve to solution.
- Induction ○ One can learn from *examples*. Humans often classify things in the world without knowing explicit rules. Usually involves a teacher or trainer to aid the classification.
- Discovery ○ Here one learns knowledge without the aid of a teacher.
- Analogy
    - ○ If a system can recognise similarities in information already stored then it may be able to transfer some knowledge to improve to solution of the task in hand.

Rote Learning

Rote Learning is basically *memorisation*.

- Saving knowledge so it can be used again.
- Retrieval is the only problem.
- No repeated computation, inference or query is necessary.

A simple example of rote learning is *caching*

- Store computed values (or large piece of data) Recall this information when required by computation.
- Significant time savings can be achieved.
- Many AI programs (as well as more general ones) have used caching very effectively.

Memorisation is a key necessity for learning:

- It is a basic necessity for any intelligent program -- is it a separate learning process?
- Memorisation can be a complex subject -- how best to store knowledge?

Samuel's Checkers program employed rote learning (it also used parameter adjustment which will be discussed shortly).

- A minimax search was used to explore the game tree.
- Time constraints do not permit complete searches.
- It *records* board positions and scores at search ends.
- Now if the same board position arises later in the game the stored value can be recalled and the end effect is that more deeper searched have occurred.

Rote learning is basically a simple process. However it does illustrate some issues that are relevant to more complex learning issues.

- Organisation ○ access of the stored value must be faster than it would be to recompute it.
  Methods such as hashing, indexing and sorting can be employed to enable this.
  ○ *E.g* Samuel's program indexed board positions by noting the number of pieces.
- Generalisation ○ The number of potentially stored objects can be very large. We may need to generalise some information to make the problem manageable.
  ○ *E.g* Samuel's program stored game positions only for white to move. Also rotations along diagonals are combined.
- Stability of the Environment ○ Rote learning is not very effective in a rapidly changing environment. If the environment does change then we must detect and record exactly what has changed -- *the frame problem*.

Store v Compute

- Rote Learning must not decrease the efficiency of the system.
- We be must able to decide whether it is worth storing the value in the first place.
- Consider the case of multiplication -- it is quicker to recompute the product of two numbers rather than store a large multiplication table.

How can we decide?

- Cost-benefit analysis ○ Decide when the information is first available whether it should be stored. An analysis could weigh up amount of storage required, cost of computation, likelihood of recall.
- Selective forgetting ○ here we allow the information to be stored initially and decide later if we retain it. Clearly the frequency of reuse is a good measure. We could tag an object with its *time of last use*. If the cache memory is full and we wish to add a new item we remove the least recently used object. Variations could include some form of costbenefit analysis to decide if the object should be removed.

Learning by Taking Advice

- The idea of advice taking in AI based learning was proposed as early as 1958 (McCarthy).

  However very few attempts were made in creating such systems until the late 1970s.
- Expert systems providing a major impetus in this area.

There are two basic approaches to advice taking:

- Take high level, abstract advice and convert it into rules that can guide performance elements of the system. *Automate all aspects of advice taking*
- *Develop sophisticated tools* such as knowledge base editors and debugging. These are used to aid an expert to translate his expertise into detailed rules. Here the expert is an *integral* part of the learning system. Such tools are important in *expert systems* area of AI.

Automated Advice Taking

The following steps summarise this method:

- Request ○ This can be simple question asking about general advice or more complicated by identifying shortcomings in the knowledge base and asking for a remedy.
- Interpret ○ Translate the advice into an *internal representation*.
- Operationalise ○ Translated advice may still not be usable so this stage seeks to provide a representation that can be used by the performance element.
- Integrate
  - When knowledge is added to the knowledge base care must be taken so that bad side-effects are avoided.
  - *E.g.* Introduction of redundancy and contradictions.
- Evaluate ○ The system must assess the new knowledge for errors, contradictions *etc.*

The steps can be iterated.

- Knowledge Base Maintenance ○ Instead of automating the five steps above, many researchers have instead assembled tools that aid the development and maintenance of the knowledge base.

Many have concentrated on:

- Providing intelligent editors and flexible representation languages for integrating new knowledge.
- Providing debugging tools for evaluating, finding contradictions and redundancy in the existing knowledge base.

EMYCIN is an example of such a system.

Example Learning System - FOO

Learning the game of hearts

FOO (First Operational Operationaliser) tries to convert high level advice (principles, problems, methods) into effective executable (LISP) procedures.

Hearts:

- Game played as a series of *tricks*.
- One player - who has the *lead* - plays a card. Other players follow in turn and play a card.
    - The player must follow suit.
    - If he cannot he play any of his cards.
- The player who plays the highest value card *wins* the trick and the lead.
- The winning player takes the cards played in the trick.
- The *aim* is to avoid taking points. Each heart counts as one point the queen of spades is worth 13 points.
- The winner is the person that after all tricks have been played has the lowest points score.

Hearts is a game of partial information with no known algorithm for winning.

Although the possible situations are numerous general advice can be given such as:

- Avoid taking points.
- Do not lead a high card in suit in which an opponent is void.
- If an opponent has the queen of spades try to flush it.

In order to receive advice a human must convert into a FOO representation (LISP clause)

(avoid (take-points me) (trick))

FOO *operationalises* the advice by translating it into expressions it can use in the game. It can *UNFOLD* avoid and then trick to give:

```
(achieve (not (during
          (scenario
           (each p1 (players) (play-card p1))
           (take-trick (trick-winner)))
           (take-points me))))
```

However the advice is still not *operational* since it depends on the outcome of trick which is generally not known. Therefore FOO uses *case analysis* (on the during expression) to determine which steps could case one to take points. Step 1 is ruled out and step 2's take-points is UNFOLDED:

(achieve (not (exists c1 (cards-played)

(exists c2 (point-cards)

(during (take (trick-winner) c1)

(take me c2))))))

FOO now has to decide: Under what conditions does (take me c2) occur during (take (trickwinner) c1).

A technique, called *partial matching*, hypothesises that points will be taken if me = trickwinner and c2 = c1. We can reduce our expression to:

(achieve (not (and (have-points(card-played))

(= (trick-winner) me))))

This not quite enough a this means *Do not win trick that has points*. We do not know who the trick-winner is, also we have not said anything about how to play in a trick that has point led in the suit. After a few more steps to achieve this FOO comes up with:

(achieve (>= (and (in-suit-led(card-of me))

(possible (trick-has-points)))

(low(card-of me)))

FOO had an initial knowledge base that was made up of:

- basic domain concepts such as trick, hand, deck suits, avoid, win *etc.*
- Rules and behavioural constraints -- general rules of the game.
- Heuristics as to how to UNFOLD.

FOO has 2 basic shortcomings:

- It lacks a control structure that could apply operationalisation automatically.
- It is specific to hearts and similar tasks.

Learning by Problem Solving

There are three basic methods in which a system can learn from its own experiences.

Learning by Parameter Adjustment

Many programs rely on an evaluation procedure to summarize the state of search *etc.* Game playing programs provide many examples of this.

However, many programs have a static evaluation function.

In learning a slight modification of the formulation of the evaluation of the problem is required.
Here the problem has an evaluation function that is represented as a polynomial of the form such as:

$$c_1t_1 + c_2t_2 + c_3t_3 + \ldots$$

The *t* terms values of features and the *c* terms are weights.

In designing programs it is often difficult to decide on the exact value to give each weight initially.

So the basic idea of idea of *parameter adjustment* is to:

- Start with some estimate of the correct weight settings.
- Modify the weight in the program on the basis of accumulated experiences.

- Features that appear to be good predictors will have their weights increased and bad ones will be decreased.

Samuel's Checkers programs employed 16 such features at any one time chosen from a pool of 38.

Learning by Macro Operators

The basic idea here is similar to Rote Learning:

Avoid expensive recomputation

Macro-operators can be used to group a whole series of actions into one.

For example: Making dinner can be described a lay the table, cook dinner, serve dinner. We could treat laying the table as on action even though it involves a sequence of actions.

The STRIPS problem-solving employed macro-operators in it's learning phase.

Consider a blocks world example in which ON(C,B) and ON(A,TABLE) are true.

STRIPS can achieve ON(A,B) in four steps:

UNSTACK(C,B), PUTDOWN(C), PICKUP(A), STACK(A,B)

STRIPS now builds a macro-operator MACROP with preconditions ON(C,B), ON(A,TABLE), postconditions ON(A,B), ON(C,TABLE) and the four steps as its body.

MACROP can now be used in future operation.

But it is not very general. The above can be easily generalised with variables used in place of the blocks.

However generalisation is not always that easy (See Rich and Knight).

Learning by Chunking

Chunking involves similar ideas to Macro Operators and originates from psychological ideas on memory and problem solving.

The computational basis is in production systems (studied earlier).

SOAR is a system that use production rules to represent its knowledge. It also employs chunking to learn from experience.

Basic Outline of SOAR's Method

- SOAR solves problems it fires productions these are stored in *long term memory*.
- Some firings turn out to be more useful than others.
- When SOAR detects are useful sequence of firings, it creates *chunks*.
- A *chunk* is essentially a large production that does the work of an entire sequence of smaller ones.
- Chunks may be generalised before storing.

Inductive Learning

This involves the process of *learning by example* -- where a system tries to induce a general rule from a set of observed instances.

This involves classification -- assigning, to a particular input, the name of a class to which it belongs. Classification is important to many problem solving tasks.

A learning system has to be capable of evolving its own class descriptions:

- Initial class definitions may not be adequate.
- The world may not be well understood or rapidly changing.

The task of constructing class definitions is called *induction* or *concept learning*

A Blocks World Learning Example -- Winston (1975)

- The goal is to construct representation of the definitions of concepts in this domain.
- Concepts such a house - brick (rectangular block) with a wedge (triangular block) suitably placed on top of it, tent - 2 wedges touching side by side, or an arch - two non-touching bricks supporting a third wedge or brick, were learned.
- The idea of *near miss* objects -- similar to actual instances was introduced.
- Input was a line drawing of a blocks world structure.
- Input processed (see VISION Sections later) to produce a semantic net representation of the structural description of the object (Fig. 27)

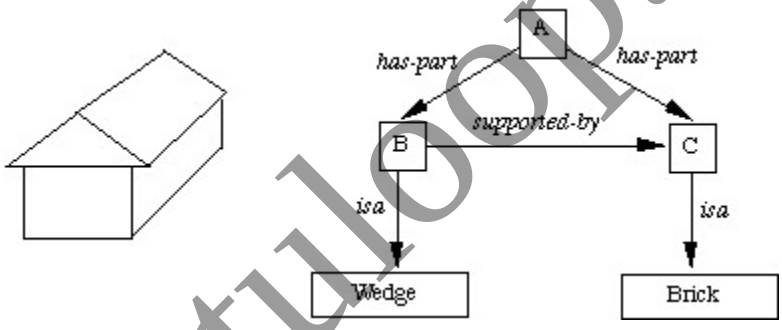


Fig. 27 House object and semantic net

- Links in network include *left-of, right-of, does-not-marry, supported-by, has-part,* and *isa*.
- The *marry* relation is important -- two objects with a common touching edge are said to marry. Marrying is assumed unless *does-not-marry* stated.

There are three basic steps to the problem of concept formulation:

1. Select one know instance of the concept. Call this the *concept definition*.
2. Examine definitions of other known instance of the concept. *Generalise* the definition to include them.
3. Examine descriptions of *near misses*. *Restrict* the definition to *exclude* these.

Both steps 2 and 3 rely on comparison and both similarities and differences need to be identified.

Version Spaces

Structural concept learning systems are not without their problems.

The biggest problem is that the *teacher* must guide the system through carefully chosen sequences of examples.

In Winston's program the order of the process is important since new links are added as and when now knowledge is gathered.

The concept of *version spaces* aims is insensitive to order of the example presented.

To do this instead of evolving a single concept description a set of possible descriptions are maintained. As new examples are presented the set evolves as a process of new instances and near misses.

We will assume that each *slot* in a version space description is made up of a set of predicates that do not negate other predicates in the set -- *positive literals*.

Indeed we can represent a description as a frame bases representation with several slots or indeed use a more general representation. For the sake of simplifying the discussion we will keep to simple representations.

If we keep to the above definition the Mitchell's *candidate elimination algorithm* is the best known algorithm.

Let us look at an example where we are presented with a number of playing cards and we need to learn if the card is *odd and black*.

We already know things like *red, black, spade, club, even card, odd card etc.*

So the is *red* card, an *even* card and a *heart*.

This illustrates on of the keys to the version space method *specificity*:

- Conjunctive concepts in the domain can be partially ordered by specificity.
- In this Cards example the concept *black* is less specific than *odd black* or *spade*.
- *odd black* and *spade* are incomparable since neither is more (or less) specific.
- *Black* is more specific than *any card*, *any 8* or *any odd card*

The training set consist of a collection of cards and for each we are told whether or not it is in the *target set* -- *odd black*

The training set is dealt with *incrementally* and a list of most and least specific concepts consistent with training instances are maintained.

Let us see how can learn from a sample input set:

- Initially the most specific concept consistent with the data is the empty set. The least specific concept is the set of all cards.

- Let the be the first card in the sample set. We are told that this is *odd black*.

- So the most specific concept is alone the least is still all our cards.

- Next card : we need to modify our most specific concept to indicate the generalisation of the set something like ``odd and black cards''. Least remains unchanged.

- Next card : Now we can modify the least specific set to exclude the . As more exclusion are added we will generalise this to all black cards and all odd cards.

- NOTE that negative instances cause least specific concepts to become more specific and positive instances similarly affect the most specific.
- If the two sets become the same set then the result is guaranteed and the target concept is met.

The Candidate Elimination Algorithm

Let us now formally describe the algorithm.

Let *G* be the set of most general concepts. Let *S* be the set of most specific concepts.

Assume: We have a common representation language and we a given a set of negative and positive training examples.

Aim: A concept description that is consistent with all the positive and *none* of the negative examples.

Algorithm:

- Initialise *G* to contain one element -- the *null* description, all features are variables.
- Initialise *S* to contain one element the first positive example.
- Repeat ○ Input the next training example ○ If a *positive example* -- first remove from *G* any descriptions that do not cover the example. Then update *S* to contain the most specific set of descriptions in the version space that cover the example and the current element set of *S*. *I.e. Generalise* the elements of *S* as little as possible so that they cover the new training example.
    - If a *negative example* -- first remove from *S* any descriptions that cover the example. Then update *G* to contain the most general set of descriptions in the version space that do not cover the example. *I.e. Specialise* the elements of *S* as little as possible so that negative examples are no longer covered in *G*'s elements.

  until *S* and *G* are both singleton sets.
- If *S* and *G* are identical output their value.
- *S* and *G* are different then training sets were inconsistent.

Let us now look at the problem of learning the concept of a *flush* in poker where all five cards are of the same suit.

$$(5\clubsuit, 7\clubsuit, 8\clubsuit, J\clubsuit, K\clubsuit)$$

Let the first example be positive:

Then we set

$$G = \{(x_1, x_2, x_3, x_4, x_5)\}$$
$$S = \{(5\clubsuit, 7\clubsuit, 8\clubsuit, J\clubsuit, K\clubsuit)\}$$

Then set

$$(5\clubsuit, 5\heartsuit, 6\diamondsuit, J\heartsuit, A\spadesuit)$$

No the second example is negative:

We must specialise *G* (only to current set):

$$
\begin{aligned}
G \quad = \quad & \{(x_1, x_2 = 7\clubsuit, x_3, x_4, x_5), \\
& (x_1, x_2, x_3 = 8\clubsuit, x_4, x_5), \\
& (x_1, x_2, x_3, x_4 = J\clubsuit, x_5), \\
& (x_1, x_2, x_3, x_4, x_5 = K\clubsuit)\}
\end{aligned}
$$

S is unaffected. $(5\clubsuit, 6\clubsuit, 8\clubsuit, 10\clubsuit, Q\clubsuit)$

Our third example is positive:

Firstly remove inconsistencies from *G* and then generalise *S*:

$$
\begin{aligned}
G \quad = \quad & \{(x_1, x_2 = \clubsuit, x_3, x_4, x_5), \\
& (x_1, x_2, x_3 = \clubsuit, x_4, x_5), \\
& (x_1, x_2, x_3, x_4 = \clubsuit, x_5), \\
& (x_1, x_2, x_3, x_4, x_5 = \clubsuit)\} \\
S \quad = \quad & \{(x_1 = 5\clubsuit, x_2 = \clubsuit, x_3 = \clubsuit, x_4 = \clubsuit, x_5 = \clubsuit)\} \\
& (A\heartsuit, 6\heartsuit, 8\heartsuit, 10\heartsuit, Q\heartsuit)
\end{aligned}
$$

Our fourth example is also positive:

Once more remove inconsistencies from *G* and then generalise *S*:

$$
\begin{aligned}
G \quad = \quad & \{(x_1 = x_2 = x_3 = x_4 = x_5 = samesuit, x_2, x_3, x_4, x_5), \} \\
S \quad = \quad & \{(x_1 = x_2 = x_3 = x_4 = x_5 = samesuit, x_2, x_3, x_4, x_5)\}
\end{aligned}
$$

- We can continue generalising and specialising
- We have taken a few big jumps in the flow of specialising/generalising in this example.
  Many more training steps usually required to reach this conclusion.
  It might be hard to spot trend of same suit *etc.*

Decision Trees

Quinlan in his ID3 system (986) introduced the idea of decision trees.

ID3 is a program that can build trees automatically from given positive and negative instances. Basically each leaf of a *decision tree* asserts a positive or negative concept. To classify a particular input we start at the top and follow assertions down until we reach an answer (Fig 28)

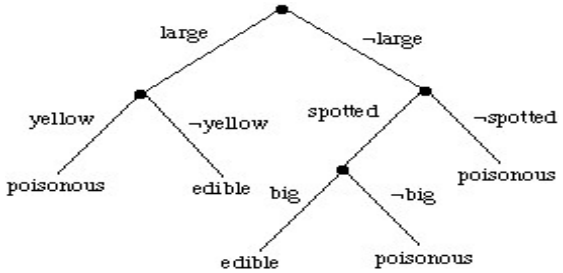


Fig. 28 *Edible Mushroom* decision tree

Building decision trees

- ID3 uses an iterative method.
- Simple trees preferred as more accurate classification is afforded.

- A random choice of samples from training set chosen for initial assembly of tree -- the *window* subset.
- Other training examples used to test tree.
- If all examples classified correctly stop.
- Otherwise add a number of training examples to *window* and start again.

Adding new nodes

When assembling the tree we need to choose when to add a new node:

- Some attributes will yield more information than others.
- Adding a new node might be useless in the overall classification process.
- Sometimes attributes will separate training instances into subsets whose members share a common label. Here branching can be terminates and a leaf node assigned for the whole subset.

Decision tree advantages:

- Quicker than version spaces when concept space is large. Disjunction easier.

Disadvantages:

- Representation not natural to humans -- a decision tree may find it hard to explain its classification.

Explanation Based Learning (EBL)

- Humans appear to learn quite a lot from one example.
- Basic idea: Use results from one examples problem solving effort next time around.
- An EBL accepts 4 kinds of input:
- A training example
    - what the learning *sees* in the world.
    - A goal concept
    - a high level description of what the program is supposed to learn.
    - A operational criterion
    - a description of which concepts are usable.
    - A domain theory
    - a set of rules that describe relationships between objects and actions in a domain.
    - From this EBL computes a generalization of the training example that is sufficient not only to describe the goal concept but also satisfies the operational criterion. This has two steps:
        - Explanation
        - the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.
        - Generalisation
        - the explanation is generalized as far possible while still describing the goal concept.

EBL example

Goal: To get to Brecon -- a picturesque welsh market town famous for its mountains (beacons) and its Jazz festival. The training data is: near(Cardiff, Brecon), airport(Cardiff)

The Domain Knowledge is:

$\forall$ near(x,y) holds(loc(x),s) $\rightarrow$ holds(loc(y), result(drive(x,y),s)) airport(z) loc(z), result(fly(z),s)))

In this case operational criterion is: We must express concept definition in pure description language syntax.

Our goal can expressed as follows:

holds(loc(Brecon),s) -- find some situation *s* for this holds.

We can prove this holds with *s* defined by:

result(drive(Cardiff,Brecon), result(fly(Cardiff), s')))

We can fly to Cardiff and then drive to Brecon.

If we analyse the proof (say with an ATMS). We can learn a few general rules from it.

Since Brecon appears in query and binding we could abstract it to give:

holds(loc(x),drive(Cardiff,x), result(fly(Cardiff), s'))) but this not quite right - we cannot get everywhere by flying to Cardiff.

Since Brecon appears in the database when we abstract things we must explicitly record the use of the fact:

near(Cardiff,x) $\rightarrow$ holds(loc(x),drive(Cardiff,x), result(fly(Cardiff), s')))

This states if x is near Cardiff we can get to it by flying to Cardiff and then driving. We have *learnt* this general rule.

We could also abstract out Cardiff instead of Brecon to get:

near(Brecon,x) $\forall$ airport(x) $\exists$ holds(loc(Brecon), result(drive(x,Brecon), result(fly(x),s')))

This states we can get top Brecon by flying to another nearby airport and driving from there.

We could add airport(Swansea) and get an alternative means of travel plan. Finally we could actually abstract out both Brecon and Cardiff to get a general plan:

near(x,y) $\forall$ airport(y) $\rightarrow$ holds(loc(y), result(drive(x,y),result(fly(x),s')))

Discovery

Discovery is a restricted form of learning in which one entity acquires knowledge without the help of a teacher.

Theory Driven Discovery - AM (1976)

AM is a program that discovers concepts in elementary mathematics and set theory.

AM has 2 inputs:

- A description of some concepts of set theory (in LISP form). *E.g.* set union, intersection, the empty set.
- Information on how to perform mathematics. *E.g.* functions.

Given the above information AM *discovered*:

- Integers ○ it is possible to count the elements of this set and this is an the image of this counting function -- the integers -- interesting set in its own right.
- Addition ○ The union of two disjoint sets and their counting function.
- Multiplication ○ Having discovered addition and multiplication as laborious set-theoretic operations more effective descriptions were supplied by hand.
- Prime Numbers ○ factorisation of numbers and numbers with only one factor were discovered.
- Golbach's Conjecture ○ Even numbers can be written as the sum of 2 primes. *E.g.* 28 = 17 + 11.
- Maximally Divisible Numbers ○ numbers with as many factors as possible. A number *k* is maximally divisible is *k* has more factors than any integer less than *k*. *E.g.* 12 has six divisors 1,2,3,4,6,12.

How does AM work?

AM employs many general-purpose AI techniques:

- A frame based representation of mathematical concepts.
    - AM can create new concepts (slots) and fill in their values.
- Heuristic search employed
    - 250 heuristics represent *hints* about activities that might lead to interesting discoveries.
    - How to employ functions, create new concepts, generalisation *etc.*
- Hypothesis and test based search.
- Agenda control of discovery process.

Data Driven Discovery -- BACON (1981)

Many discoveries are made from observing data obtained from the world and making sense of it -- *E.g.* Astrophysics - discovery of planets, Quantum mechanics - discovery of sub-atomic particles.

BACON is an attempt at provided such an AI system.

BACON system outline:

- Starts with a set of variables for a problem.
    - *E.g.* BACON was able able to derive the *ideal gas law*. It started with four variables *p* - gas pressure, *V* -- gas volume, *n* -- molar mass of gas, *T* -- gas temperature. Recall $pV/nT = k$ where *k* is a constant.
- Values from experimental data from the problem are inputted.
- BACON holds some constant and attempts to notice trends in the data. Inferences made.

BACON has also been applied to Kepler's 3rd law, Ohm's law, conservation of momentum and Joule's law.

Analogy

Analogy involves a complicated mapping between what might appear to be two dissimilar concepts.

Bill is built like a large outdoor brick lavatory.

He was like putty in her hands

Humans quickly recognise the abstractions involved and understand the meaning.
There are two methods of analogical problem methods studied in AI.

Transformational Analogy

Look for a similar solution and *copy* it to the new situation making suitable substitutions where appropriate.

E.g. Geometry.

If you know about lengths of line segments and a proof that certain lines are equal (Fig. 29) then we can make similar assertions about angles.

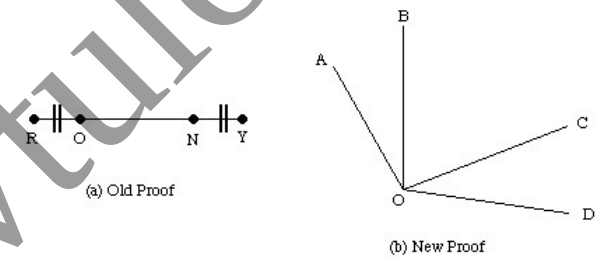


Fig. 29 Transformational Analogy Example

- We know that lines *RO* = *NY* and angles *AOB* = *COD* We have seen that *RO* + *ON* = *ON* + *NY* - additive rule.
- So we can say that angles *AOB* + *BOC* = *BOC* + *COD*
- So by a transitive rule line *RN* = *OY*
- So similarly angle *AOC* = *BOD*

Carbonell (1983) describes a *T-space* method to transform old solutions into new ones.

- Whole solutions are viewed as states in a problem space -- the *T-space*.
- *T-operators* prescribe methods of transforming existing solution states into new ones.

Derivational Analogy

Transformational analogy does not look at how the problem was solved -- it only looks at the final solution.

The *history* of the problem solution - the steps involved - are often relevant.

Carbonell (1986) showed that derivational analogy is a necessary component in the transfer of skills in complex domains:

- In translating Pascal code to LISP -- line by line translation is no use. You will have to *reuse* the major structural and control decisions.
- One way to do this is to *replay* a previous derivation and modify it when necessary.
- If initial steps and assumptions are still valid copy them across.
- Otherwise alternatives need to found -- best first search fashion.
- Reasoning by analogy becomes a search in T-space -- means-end analysis.

12. Expert Systems

Expert Systems

- Expert systems (ES) are one of the prominent research domains of AI. It is introduced by the researchers at Stanford University, Computer Science Department.

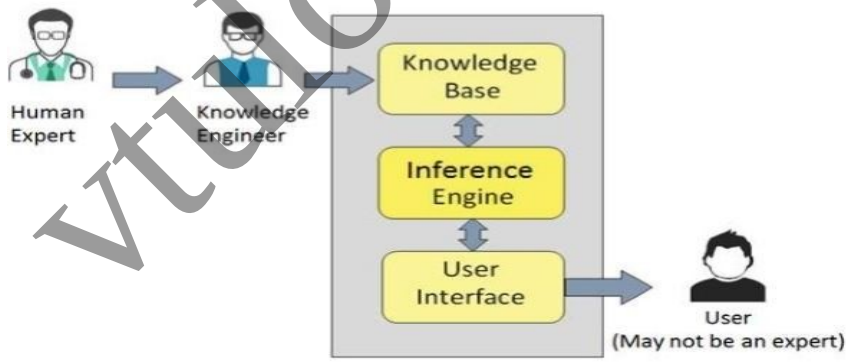
- Expert systems solve problems that are normally solved by human “experts”. To solve expert-level problems, expert systems need access to a substantial domain knowledge base, which must be built as efficiently as possible. They also need to exploit one or more reasoning mechanisms to apply their knowledge to the problems they are given. Then they need a mechanism for explaining what they have done to the users who rely on them.
- The problems that expert systems deal with are highly diverse. There are some general issues that arise across these varying domains. But it also turns out that there are powerful techniques that can be defined for specific classes of problems.
- What are Expert Systems?
    - The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise.

Capabilities of Expert Systems

- The expert systems are capable of − ○ Advising ○ Instructing and assisting human in decision making ○ Demonstrating ○ Deriving a solution ○ Diagnosing ○ Explaining ○ Interpreting input ○ Predicting results ○ Justifying the conclusion ○ Suggesting alternative options to a problem
- They are incapable of − ○ Substituting human decision makers ○ Possessing human capabilities ○ Producing accurate output for inadequate knowledge base ○ Refining their own knowledge

Components of Expert Systems

- The components of ES include − ○ Knowledge Base ○ Inference Engine ○ User Interface



Knowledge Base

- It contains domain-specific and high-quality knowledge. Knowledge is required to exhibit intelligence. The success of any ES majorly depends upon the collection of highly accurate and precise knowledge.
- What is Knowledge?
    - The data is collection of facts. The information is organized as data and facts about the task domain. Data, information, and past experience combined together are termed as knowledge.

- Components of Knowledge Base ○ The knowledge base of an ES is a store of both, factual and heuristic knowledge.
    - Factual Knowledge − It is the information widely accepted by the Knowledge Engineers and scholars in the task domain.
    - Heuristic Knowledge − It is about practice, accurate judgment, one's ability of evaluation, and guessing.
- Knowledge representation ○ It is the method used to organize and formalize the knowledge in the knowledge base. It is in the form of IF-THEN-ELSE rules.
- Knowledge Acquisition ○ The success of any expert system majorly depends on the quality, completeness, and accuracy of the information stored in the knowledge base.
    - The knowledge base is formed by readings from various experts, scholars, and the Knowledge Engineers. The knowledge engineer is a person with the qualities of empathy, quick learning, and case analyzing skills.
    - He acquires information from subject expert by recording, interviewing, and observing him at work, etc.
    - He then categorizes and organizes the information in a meaningful way, in the form of IF-THEN-ELSE rules, to be used by interference machine. The knowledge engineer also monitors the development of the ES.

Inference Engine

- Use of efficient procedures and rules by the Inference Engine is essential in deducting a correct, flawless solution.
- In case of knowledge-based ES, the Inference Engine acquires and manipulates the knowledge from the knowledge base to arrive at a particular solution.
- In case of rule based ES, it − ○ Applies rules repeatedly to the facts, which are obtained from earlier rule application.
    - Adds new knowledge into the knowledge base if required.
    - Resolves rules conflict when multiple rules are applicable to a particular case.
- To recommend a solution, the Inference Engine uses the following strategies − ○ Forward Chaining ○ Backward Chaining

User Interface

- User interface provides interaction between user of the ES and the ES itself.
- It is generally Natural Language Processing so as to be used by the user who is well-versed in the task domain.
- The user of the ES need not be necessarily an expert in Artificial Intelligence.
- It explains how the ES has arrived at a particular recommendation. The explanation may appear in the following forms − ○ Natural language displayed on screen. ○ Verbal narrations in natural language. ○ Listing of rule numbers displayed on the screen.
    - The user interface makes it easy to trace the credibility of the deductions.

Requirements of Efficient ES User Interface

- It should help users to accomplish their goals in shortest possible way.
- It should be designed to work for user's existing or desired work practices.
- Its technology should be adaptable to user's requirements; not the other way round.
- It should make efficient use of user input.

Expert Systems Limitations

No technology can offer easy and complete solution. Large systems are costly; require significant development time, and computer resources. ESs have their limitations which include −

- Limitations of the technology
- Difficult knowledge acquisition
- ES are difficult to maintain
- High development costs

Applications of Expert System

The following table shows where ES can be applied.

Application	Description
Design Domain	Camera lens design, automobile design.
Medical Domain	Diagnosis Systems to deduce cause of disease from observed data, conduction medical operations on humans.
Monitoring Systems	Comparing data continuously with observed system or with prescribed behavior such as leakage monitoring in long petroleum pipeline.
Process Control Systems	Controlling a physical process based on monitoring.
Knowledge Domain	Finding out faults in vehicles, computers.
Finance/Commerce	Detection of possible fraud, suspicious transactions, stock market trading, Airline scheduling, cargo scheduling.

Expert System Technology

- There are several levels of ES technologies available. Expert systems technologies include
  - − ○ Expert System Development Environment − The ES development environment includes hardware and tools. They are − ▪ Workstations, minicomputers, mainframes.
- High level Symbolic Programming Languages such as LISt Programming (LISP) and PROgrammation en LOGique (PROLOG). ▪ Large databases.
- Tools − They reduce the effort and cost involved in developing an expert system to large extent.
  - ○ Powerful editors and debugging tools with multi-windows.

- They provide rapid prototyping ○ Have Inbuilt definitions of model, knowledge representation, and inference design.

- Shells − A shell is nothing but an expert system without knowledge base.
- A shell provides the developers with knowledge acquisition, inference engine, user interface, and explanation facility. For example, few shells are given below − ○ Java Expert System Shell (JESS) that provides fully developed Java API for creating an expert system.
    - *Vidwan*, a shell developed at the National Centre for Software Technology, Mumbai in 1993. It enables knowledge encoding in the form of IF-THEN rules.

Representing and Using Domain Knowledge

Expert systems are complex AI programs. The most widely used way of representing domain knowledge in expert systems is as a set of production rules, which are often coupled with a frame system that defines the objects that occur in the rules.

MYCIN is one example of an expert system rule. All the rules we show are English versions of the actual rules that the systems use.

- RI (sometimes are called XCON) is a program that configures DEC VAX systems. Its rules look like this:

```
If: the most current active context is distributing
        massbus devices, and
    there is a single-port disk drive that has not been
        assigned to a massbus, and
    there are no unassigned dual-port disk drives, and
        the number of devices that each massbus should
        support is known, and
    there is a massbus that has been assigned at least
        one disk drive and that should support additional
        disk drives,
    and the type of cable needed to connect the disk drive
        to the previous device on the massbus is known
then: assign the disk drive to the massbus.
```

Notice that RI's rules, unlike MYCIN's, contain no numeric measures of certainty. In the task domain with which RI deals, it is possible to state exactly the correct thing to be done in each particular set of circumstances. One reason for this is that there exists a good deal of human expertise in this area. Another is that since RI is doing a design task, it is not necessary to consider all possible alternatives; one good one is enough. As a result, probabilistic information is not necessary in RI.

- PROSPECTOR is a program that provides advice on mineral exploration. Its rules look like this:

```
If: magnetite or pyrite in disseminated or veinlet form is present
then: (2, -4) there is favorable mineralization and texture
    for the propylitic stage.
```

In PROSPECTOR, each rule contains two confidence estimates. The first indicates the extent to which the presence of the evidence described in the condition part of the rule suggests the validity of the rule's conclusion. In the PROSPECTOR rule shown above, the number 2 indicates that the presence of the evidence is mildly encouraging. The second confidence estimate measures the extent to which the evidence is necessary to the validity of the conclusion or stated another way, the extent to which the lack of the evidence indicates that the conclusion is not valid.

- DESIGN ADVISOR is a system that critiques chip designs. Its rules look like:

```
If: the sequential level count of ELEMENT is greater than 2,
    UNLESS the signal of ELEMENT is resetable
then: critique for poor resetability
DEFEAT: poor resetability of ELEMENT
due to: sequential level count of ELEMENT greater than 2
by: ELEMENT is directly resetable
```

This gives advice to a chip designer, who can accept or reject the advice. If the advice is rejected,, the system can exploit a justification-based truth maintenance system to revise its model of the circuit. The first rule shown here says that an element should be criticized for poor resetability if the sequential level count is greater than two, unless its signal is currently believed to be resettable.

Reasoning with the Knowledge

Expert systems exploit many of the representation and reasoning mechanisms that we have seen. Because these programs are usually written primarily as rule-based systems, forward chaining, backward chaining, or some combination of the two is usually used. For example, MYCIN used backward chaining to discover what organisms were present; then it used forward chaining to reason from the organisms to a treatment regime. RI, on the other hand, used forward chaining. As the field of expert systems matures, more systems that exploit other kinds of reasoning mechanisms are being developed. The DESIGN ADVISOR is an example of such a system; in addition to exploiting rules, it makes extensive use of a justification-based truth maintenance system.

EXPERT SYSTEM SHELLS

Initially, each expert system that was built was created from scratch, usually in LISP. In particular, since the systems were constructed as a set of declarative representations combined with an interpreter for those representations, it was possible to separate the interpreter from the domainspecific knowledge and thus to create a system that could be used to construct new expert systems by adding new knowledge corresponding to the new problem domain. The resulting interpreters are called shells. One influential example of such a shell is EMYCIN (for empty MYCIN) which was derived from MYCIN.

There are now several commercially available shells that serve as the basis for many of the expert systems currently being built. These shells provide much greater flexibility in representing knowledge and in reasoning with it than MYCIN did. They typically support rules, frames, truth maintenance systems, and a variety of other reasoning mechanisms.

Early expert systems shells provided mechanisms for knowledge representation, reasoning and explanation. But as experience with using these systems to solve real world problem grew, it became clear that expert system shells needed to do something else as well. They needed to make it easy to integrate expert systems with other kinds of programs.

EXPLANATION

In order for an expert system to be an effective tool, people must be able to interact with it easily. To facilitate this interaction, the expert system must have the following two capabilities in addition to the ability to perform its underlying task:

- Explain its reasoning:
    - In many of the domains in which expert systems operate, people will not accept results unless they have been convinced of the accuracy of the reasoning process that produced those results. This is particularly true, for example, in medicine, where a doctor must accept ultimate responsibility for a diagnosis, even if that diagnosis was arrived at with considerable help from a program.
- Acquire new knowledge and modifications of old knowledge:
    - Since expert systems derive their power from the richness of the knowledge bases they exploit, it is extremely important that those knowledge bases be as complete and as accurate as possible. One way to get this knowledge into a program is through interaction with the human expert. Another way is to have the program learn expert behavior from raw data.

KNOWLEDGE ACQUISITION

How are expert systems built? Typically, a knowledge engineer interviews a domain expert to elucidate expert knowledge, when is then translated into rules. After the initial system is built, it must be iteratively refined until it approximates expert-level performance. This process is expensive and time-consuming, so it is worthwhile to look for more automatic ways of constructing expert knowledge bases.

While no totally automatic knowledge acquisition systems yet exist, there are many programs that interact with domain experts to extract expert knowledge efficiently. These programs provide support for the following activities:

- Entering knowledge
- Maintaining knowledge base consistency
- Ensuring knowledge base completeness

The most useful knowledge acquisition programs are those that are restricted to a particular problem-solving paradigm e.g. diagnosis or design. It is important to be able to enumerate the roles that knowledge can play in the problem-solving process. For example, if the paradigm is

diagnosis, then the program can structure its knowledge base around symptoms, hypotheses and causes. It can identify symptoms for which the expert has not yet provided causes.

Since one symptom may have multiple causes, the program can ask for knowledge about how to decide when one hypothesis is better than another. If we move to another type of problemsolving, say profitably interacting with an expert.

MOLE (Knowledge Acquisition System)

It is a system for heuristic classification problems, such as diagnosing diseases. In particular, it is used in conjunction with the cover-and-differentiate problem-solving method. An expert system produced by MOLE accepts input data, comes up with a set of candidate explanations or classifications that cover (or explain) the data, then uses differentiating knowledge to determine which one is best. The process is iterative, since explanations must themselves be justified, until ultimate causes the ascertained.

MOLE interacts with a domain expert to produce a knowledge base that a system called MOLE-p (for MOLE-performance) uses to solve problems. The acquisition proceeds through several steps:

1. Initial Knowledge base construction.

   MOLE asks the expert to list common symptoms or complaints that might require diagnosis. For each symptom, MOLE prompts for a list of possible explanations. MOLE then iteratively seeks out higher-level explanations until it comes up with a set of ultimate causes. During this process, MOLE builds an influence network similar to the belief networks.

   The expert provides covering knowledge, that is, the knowledge that a hypothesized event might be the cause of a certain symptom.

2. Refinement of the knowledge base.

   MOLE now tries to identify the weaknesses of the knowledge base. One approach is to find holes and prompt the expert to fill them. It is difficult, in general, to know whether a knowledge base is complete, so instead MOLE lets the expert watch MOLE-p solving sample problems. Whenever MOLE-p makes an incorrect diagnosis, the expert adds new knowledge. There are several ways in which MOLE-p can reach the wrong conclusion. It may incorrectly reject a hypothesis because it does not feel that the hypothesis is needed to explain any symptom.

MOLE has been used to build systems that diagnose problems with car engines, problems in steelrolling mills, and inefficiencies in coal-burning power plants. For MOLE to be applicable, however, it must be possible to preenumerate solutions or classifications. It must also be practical to encode the knowledge in terms of covering and differentiating.

One problem-solving method useful for design tasks is called propose-and-revise. Propose-andrevise systems build up solutions incrementally. First, the system proposes an extension to the current design. Then it checks whether the extension violates any global or local constraints. Constraints violations are then fixed, and the process repeats.

SALT Program

The SALT program provides mechanisms for elucidating this knowledge from the expert. Like MOLE, SALT builds a dependency network as it converses with an expert. Each node stands for a value of a parameter that must be acquired or generated. There are three kinds of links:

- *Contributes-to:* Associated with the first type of link are procedures that allow SALT to generate a value for one parameter based on the value of another.
- *Constraints:* Rules out certain parameter values.
- *Suggests-revision-of*: points of ways in which a constraint violation can be fixed.

SALT uses the following heuristics to guide the acquisition process:

1. Every non-input node in the network needs at least one contributes-to link coming into it. If links are missing, the expert is prompted to fill them in.
2. No contributes-to loops are allowed in the network. Without a value for at least one parameter in the loop, it is impossible to compute values for any parameter in that loop. If a loop exists, SALT tries to transform one of the contributes-to links into a constraints link.
3. Constraining links should have suggests-revision-of links associated with them. These include constraints links that are created when dependency loops are broken.

Control Knowledge is also important. It is critical that the system propose extensions and revisions that lead toward a design solution. SALT allows the expert to rate revisions in terms of how much trouble they tend to produce.

SALT compiles its dependency network into a set of production rules. As with MOLE, an expert can watch the production system solve problems and can override the system's decision. At the point, the knowledge base can be changes or the override can be logged for future inspection.