1. The Concept of Artificial Intelligence

Artificial Intelligence (AI) is a branch of *Science* which deals with helping machines finding solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behaviour appears.

AI is generally associated with *Computer Science*, but it has many important links with other fields such as *Maths*, *Psychology*, *Cognition*, *Biology* and *Philosophy*, among many others. Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being.

AI is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. However, the study of intelligence is one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done. The advent of usable computers in the early 1950s turned the learned but armchair speculation concerning these mental faculties into a real experimental and theoretical discipline. Many felt that the new ``Electronic Super-Brains" had unlimited potential for intelligence. ``Faster Than Einstein" was a typical headline. But as well as providing a vehicle for creating artificially intelligent entities, the computer provides a tool for testing theories of intelligence, and many theories failed to withstand the test. AI has turned out to be more difficult than many at first imagined, and modern ideas are much richer, more subtle, and more interesting as a result.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavour. In this sense, it is truly a universal field.

Definition of Artificial intelligence

It is often difficult to construct a definition of a discipline that is satisfying to all of its practitioners. AI research encompasses a spectrum of related topics. Broadly, AI is the *computer-based* exploration of methods for solving challenging tasks that have traditionally depended on people for solution. Such tasks include complex logical inference, diagnosis, visual recognition, comprehension of natural language, game playing, explanation, and planning.

We shall begin our study of AI, by considering a number of alternative definitions of this topic

Alternative

AI is the study of how to make computers do things which at the moment people do better. This is ephemeral as it refers to the current state of computer science and it excludes a major area ; problems that cannot be solved well either by computers or by people at the moment.

Alternative

AI is a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans.

Alternative

AI is the branch of computer science that is concerned with the automation of intelligent behaviour. A I is based upon the principles of computer science namely data structures used in knowledge representation, the algorithms needed to apply that knowledge and the languages and programming techniques used in their implementation.

These definitions avoid philosophic discussions as to what is meant by artificial or intelligence.

Alternative

AI is the field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes.

Alternative

AI is about generating representations and procedures that automatically or autonomously solve problems heretofore solved by humans.

Alternative

A I is the part of computer science concerned with designing intelligent computer systems, that is, computer systems that exhibit the characteristics we associate with intelligence in human behaviour such as understanding language, learning, reasoning and solving problems.

Alternative

A I is the study of mental faculties through the use of computational models

Alternative

A I is the study of the computations that make it possible to perceive, reason, and act

Alternative

A I is the exciting new effort to make computers think *machines with minds*, in the full and literal sense

In brief summary, AI is concerned with developing computer systems that can store knowledge and effectively use the knowledge to help solve problems and accomplish tasks. This brief statement sounds a lot like one of the commonly accepted goals in the education of humans. We want students to learn (gain knowledge) and to learn to use this knowledge to help solve problems and accomplish tasks.

The above definitions give us four possible goals to pursue in artificial intelligence:

- Systems that think like humans
- Systems that act like humans
- Systems that think rationally. (A system is rational if it does the right thing.)
- Systems that act rationally

Historically, all four approaches have been followed. As one might expect, a tension exists between approaches centered around humans and approaches centered around rationality. (We should point out that by distinguishing between *human* and *rational* behaviour, we are not suggesting that humans are necessarily ``irrational" in the sense of ``emotionally unstable" or ``insane." One merely need note that we often make mistakes; we are not all chess grandmasters even though we may know all the rules of chess; and unfortunately, not everyone gets an A on the exam. A human-centered approach must be an empirical science, involving hypothesis and experimental confirmation. A rationalist approach involves a combination of mathematics and engineering. People in each group sometimes cast aspersions on work done in the other groups, but the truth is that each direction has yielded valuable insights.

Acting humanly: The Turing Test approach

The **Turing Test**, proposed by Alan Turing (Turing, 1950), was designed to provide a satisfactory operational definition of intelligence. Turing defined intelligent behaviour as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. Roughly speaking, the test he proposed is that the computer should be interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end.
Programming a computer to pass the test provides plenty to work on. The computer would need to possess the following capabilities:

- **natural language processing** to enable it to communicate successfully in English (or some other human language);
- **knowledge representation** to store information provided before or during the interrogation;
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions;

- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because *physical* simulation of a person is unnecessary for intelligence. However, the so-called **total Turing Test** includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects ``through the hatch." To pass the total Turing Test, the computer will need

- **computer vision** to perceive objects, and
- **robotics** to move them about.

Within AI, there has not been a big effort to try to pass the Turing test. The issue of acting like a human comes up primarily when AI programs have to interact with people, as when an expert system explains how it came to its diagnosis, or a natural language processing system has a dialogue with a user. These programs must behave according to certain normal conventions of human interaction in order to make themselves understood. The underlying representation and reasoning in such a system may or may not be based on a human model.

Thinking humanly: The cognitive modelling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get *inside* the actual workings of human minds. There are two ways to do this:
Through introspection (trying to catch our own thoughts as they go by)
Through psychological experiments.
Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input/output and timing behavior matches human behavior, that is evidence that some of the program's mechanisms may also be operating in humans.
For example, Newell and Simon, who developed GPS, the ``General Problem Solver" (Newell and Simon, 1961), were not content to have their program correctly solve problems. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems. This is in contrast to other researchers of the same time (such as Wang (1960)), who were concerned with getting the right answers regardless of how humans might do it. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

Thinking rationally: The laws of thought approach

The Greek philosopher Aristotle was one of the first to attempt to codify ``right thinking," that is, irrefutable reasoning processes. His famous **syllogisms** provided patterns for

argument structures that always gave correct conclusions given correct premises. For example, ``Socrates is a man; all men are mortal; therefore Socrates is mortal." These laws of thought were supposed to govern the operation of the mind, and initiated the field of **logic**.

The development of formal logic in the late nineteenth and early twentieth centuries, provided a precise notation for statements about all kinds of things in the world and the relations between them. (Contrast this with ordinary arithmetic notation, which provides mainly for equality and inequality statements about numbers.) By 1965, programs existed that could, given enough time and memory, take a description of a problem in logical notation and find the solution to the problem, if one exists. (If there is no solution, the program might never stop looking for it.) The so-called **logicist** tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem ``in principle" and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to *any* attempt to build computational reasoning systems, they appeared first in the logicist tradition because the power of the representation and reasoning systems are well-defined and fairly well understood.

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An **agent** is just something that perceives and acts. In this approach, AI is viewed as the study and construction of rational agents.

In the ``laws of thought" approach to AI, the whole emphasis was on correct inferences. Making correct inferences is sometimes *part* of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals, and then to act on that conclusion. On the other hand, correct inference is not *all* of rationality, because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also ways of acting rationally that cannot be reasonably said to involve inference. For example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the ``cognitive skills" needed for the Turing Test are there to allow rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for

dealing with it. We need visual perception not just because seeing is fun, but in order to get a better idea of what an action might achieve

Areas of Artificial Intelligence

- **Perception**

  Machine Vision:
  It is easy to interface a TV camera to a computer and get an image into memory; the problem is *understanding* what the image represents. Vision takes *lots* of computation; in humans, roughly 10% of all calories consumed are burned in vision computation.
  Speech Understanding:
  Speech understanding is available now. Some systems must be trained for the individual user and require pauses between words. Understanding continuous speech with a larger vocabulary is harder.
  Touch(*tactile* or *haptic*) Sensation:
  Important for robot assembly tasks.

- **Robotics**

Although industrial robots have been expensive, robot hardware can be cheap: Radio Shack has sold a working robot arm and hand for $15. The limiting factor in application of robotics is not the cost of the robot hardware itself.

What is needed is perception and intelligence to tell the robot what to do; ``blind" robots are limited to very well-structured tasks (like spray painting car bodies).

- **Planning**

Planning attempts to order actions to achieve goals. Planning applications include logistics, manufacturing scheduling, planning manufacturing steps to construct a desired product. There are huge amounts of money to be saved through better planning.

- **Expert Systems**

Expert Systems attempt to capture the knowledge of a human expert and make it available through a computer program. There have been many successful and economically valuable applications of expert systems. Expert systems provide the following benefits

- Reducing skill level needed to operate complex devices.
- Diagnostic advice for device repair.
- Interpretation of complex data.
- ``Cloning" of scarce expertise.
- Capturing knowledge of expert who is about to retire.
- Combining knowledge of multiple experts.

- Intelligent training.

- **Theorem Proving**

Proving mathematical theorems might seem to be mainly of academic interest. However, many practical problems can be cast in terms of theorems. A general theorem prover can therefore be widely applicable.

Examples:

- Automatic construction of compiler code generators from a description of a CPU's instruction set.
- J Moore and colleagues proved correctness of the floating-point division algorithm on AMD CPU chip.

- **Symbolic Mathematics**

Symbolic mathematics refers to manipulation of *formulas*, rather than arithmetic on numeric values.

- Algebra
- Differential and Integral Calculus

Symbolic manipulation is often used in conjunction with ordinary scientific computation as a generator of programs used to actually do the calculations. Symbolic manipulation programs are an important component of scientific and engineering workstations.

- **Game Playing**

Games are good vehicles for research because they are well formalized, small, and self-contained. They are therefore easily programmed.

Games can be good models of competitive situations, so principles discovered in game-playing programs may be applicable to practical problems.

AI Technique.

Intelligence requires knowledge but knowledge possesses less desirable properties such as

- It is voluminous
- it is difficult to characterise accurately
- it is constantly changing
- it differs from data by being organised in a way that corresponds to its application

An AI technique is a method that exploits knowledge that is represented so that

- The knowledge captures generalisations; situations that share properties, are grouped together, rather than being allowed separate representation.
- It can be understood by people who must provide it; although for many programs the bulk of the data may come automatically, such as from readings. In many AI domains people must supply the knowledge to programs in a form the people understand and in a form that is acceptable to the program.
- It can be easily modified to correct errors and reflect changes in real conditions.
- It can be widely used even if it is incomplete or inaccurate.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must be usually considered.

2. Problem Spaces and Search

Building a system to solve a problem requires the following steps

- Define the problem precisely including detailed specifications and what constitutes an acceptable solution;
- Analyse the problem thoroughly for some features may have a dominant affect on the chosen method of solution;
- Isolate and represent the background knowledge needed in the solution of the problem;
- Choose the best problem solving techniques in the solution.

Defining the Problem as state Search

To understand what exactly artificial intelligence is, we illustrate some common problems. Problems dealt with in artificial intelligence generally use a common term called 'state'. A state represents a status of the solution at a given step of the problem solving procedure. The solution of a problem, thus, is a collection of the problem states. The problem solving procedure applies an operator to a state to get the next state. Then it applies another operator to the resulting state to derive a new state. The process of applying an operator to a state and its subsequent transition to the next state, thus, is continued until the goal (desired) state is derived. Such a method of solving a problem is generally referred to as state space approach

For example, in order to solve the problem play a game, which is restricted to two person table or board games, we require the rules of the game and the targets for winning as well as a means of representing positions in the game. The opening position can be defined as the initial state and a winning position as a goal state, there can be more than one. legal moves allow for transfer from initial state to other states leading to the goal state. However the rules are far too copious in most games especially chess where they exceed the number of particles in the universe 10. Thus the rules cannot in general be supplied accurately and computer programs cannot easily handle them. The storage also presents another problem but searching can be achieved by hashing.

The number of rules that are used must be minimised and the set can be produced by expressing each rule in as general a form as possible. The representation of games in this way leads to a state space representation and it is natural for well organised games with some structure. This representation allows for the formal definition of a problem which necessitates the movement from a set of initial positions to one of a set of target positions. It means that the solution involves using known techniques and a systematic search. This is quite a common method in AI.

Formal description of a problem

- Define a state space that contains all possible configurations of the relevant objects, without enumerating all the states in it. A *state space* represents a problem in terms of *states* and *operators* that change states
- Define some of these states as possible initial states;
- Specify one or more as acceptable solutions, these are goal states;
- Specify a set of rules as the possible actions allowed. This involves thinking about the generality of the rules, the assumptions made in the informal presentation and how much work can be anticipated by inclusion in the rules.

The control strategy is again not fully discussed but the AI program needs a structure to facilitate the search which is a characteristic of this type of program.

Production system

- a set of rules each consisting of a left side the applicability of the rule and the right side the operations to be performed;
- one or more knowledge bases containing the required information for each task;
- a control strategy that specifies the order in which the rules will be compared to the database and ways of resolving conflict;
- a rule applier

Choose an appropriate search technique:

- How large is the search space?
- How well-structured is the domain?
- What knowledge about the domain can be used to guide the search?

Example1: the water jug problem

There are two jugs called **four** and **three** ; four holds a maximum of four gallons and **three** a maximum of three gallons. How can we get 2 gallons in the jug **four**.

The state space is a set of ordered pairs giving the number of gallons in the pair of jugs at any time ie (**four, three**) where **fou**r = 0, 1, 2, 3, 4 and **three** = 0, 1, 2, 3.

The start state is (0,0) and the goal state is (2,n) where n is a don't care but is limited to **three** holding from 0 to 3 gallons.

The major production rules for solving this problem are shown below:

initial condition goal comment

1 (four,three) if four < 4 (4,three) fill four from tap

2 (four,three) if three< 3 (four,3) fill three from tap

3 (four,three) If four > 0 (0,three) empty four into drain

4 (four,three) if three > 0 (four,0) empty three into drain

5 (four,three) if four+three<4 (four+three,0) empty three into four

6 (four,three) if four+three<3 (0,four+three) empty four into three

7 (0,three) If three>0 (three,0) empty three into four

8 (four,0) if four>0 (0,four) empty four into three

9 (0,2) (2,0) empty three into four

10 (2,0) (0,2) empty four into three

11 (four,three) if four<4 (4,three-diff) pour diff, 4-four, into four from three

12 (three,four) if three<3 (four-diff,3) pour diff, 3-three, into three from four and a solution is given below

Jug four, jug three rule applied

0 0

0 3 2

3 0 7

3 3 2

4 2 11

0 2 3

2 0 10

Control strategies.

A good control strategy should have the following requirement:

The first requirement is that it causes motion. In a game playing program the pieces move on the board and in the water jug problem water is used to fill jugs.

The second requirement is that it is systematic, this is a clear requirement for it would not be sensible to fill a jug and empty it repeatedly nor in a game would it be advisable to move a piece round and round the board in a cyclic way. We shall initially consider two systematic approaches to searching

Example2: The missionaries and cannibals problem

The Missionaries and Cannibals problem illustrates the use of state space search for planning under constraints:
Three missionaries and three cannibals wish to cross a river using a two-person boat. If at any time the cannibals outnumber the missionaries on either side of the river, they will eat the missionaries. How can a sequence of boat trips be performed that will get everyone to the other side of the river without any missionaries being eaten?

We decide to represent the various sub-problem states by a cartesian coordinates (m1, c1, m2, c2, boatposition).
m1 and c1 are the number of missionaries and cannibals respectively on side 1 of the river and m2, c2 the number of missionaries and cannibals respectively on side 2 of the river. The variable boatposition indicate the position of the boat. It will be 1 if the boat is at side 1 of the river or 2 if the boat is at side 2 of the river.

The initial and goal state are (3, 3, 0, 0, 1) and (0, 0, 3, 3, 2) respectively.

The major production rules for solving this problem are shown below:

initial **goal**

op1 (m1, c1, m2, c2, 1) (m1-1, c1, m2+1, c2, 2): Condition: boat on side1 and there is at least one missionary on side 1: Comment: 1 missionary leave side 1 to side 2

op2 (m1, c1, m2, c2, 1) (m1-2, c1, m2+2, c2, 2): Condition: boat on side1 and there is at least 2 missionary on side 1: Comment: 2 missionary leave side 1 to side 2

op3 (m1, c1, m2, c2, 1) (m1-1, c1-1, m2+1, c2+1, 2): Condition: boat on side1 and there is at least 1 missionary and 1 cannibal on side 1: Comment: 1 missionary and 1 cannibal leave side 1 to side 2

op4 (m1, c1, m2, c2, 1) (m1, c1-1, m2, c2+1, 2): Condition: boat on side1 and there is at least one cannibal on side 1: Comment: 1 cannibal leave side 1 to side 2

op5 (m1, c1, m2, c2, 1) (m1, c1-2, m2, c2+2, 1): Condition: boat on side1 and there is at least 2 cannibals on side 1: Comment: 2 cannibals leave side 1 to side 2

When the boat is on side 2, the following similar operations can also be applied.

op11 (m1, c1, m2, c2, 2) (m1+1, c1, m2-1, c2, 1): Condition: boat on side2 and there is at least one missionary on side 2: Comment: 1 missionary leave side 2 to side 1

op2 1 (m1, c1, m2, c2, 2) (m1+2, c1, m2-2, c2, 1): Condition: boat on side2 and there is at least 2 missionary on side 2: Comment: 2 missionary leave side 2 to side 1

op31 (m1, c1, m2, c2, 2) (m1+1, c1+1, m2-1, c2-1, 1): Condition: boat on side2 and there is at least 1 missionary and 1 cannibal on side 2: Comment: 1 missionary and 1 cannibal leave side 2 to side 1

op41 (m1, c1, m2, c2, 2) (m1, c1+1, m2, c2-1, 1): Condition: boat on side2 and there is at least one cannibal on side 2: Comment: 1 cannibal leave side 2 to side 1

op51 (m1, c1, m2, c2, 2) (m1, c1+2, m2, c2-2, 1): Condition: boat on side2 and there is at least 2 cannibals on side 2: Comment: 2 cannibals leave side 2 to side 1

The following sequence of operations applied starting from the initial state produce the solution

(3, 3, 0, 0, 1)
(2, 2, 1, 1, 2) op3
(3, 2, 0, 1, 1) op11
(3, 0, 0, 3, 2) op5
(3, 1, 0, 2, 1) op41
(1, 1, 2, 2, 2) op2
(2, 2, 1, 1, 1) op31
(0, 2, 3, 1, 2) op2
(0, 3, 3, 0, 1) op41
(0, 1, 3, 2, 2) op5
(0, 2, 3, 1, 2) op41
(0, 0, 2, 3, 2) op5

Basic Recursive Algorithm

- If the input is a base case, for which the solution is known, return the solution.
- Otherwise,
    - Do part of the problem, or break it into smaller subproblems.
    - Call the problem solver recursively to solve the subproblems.
    - Combine the subproblem solutions to form a total solution.

In writing the recursive program:

- Write a clear specification of the input and output of the program.
- Assume it works already.
- Write the program to use the input form and produce the output form.

Search Order

The excessive time spent in searching is *almost entirely spent on failures* (sequences of operators that do not lead to solutions). If the computer could be made to look at promising sequences first and avoid most of the bad ones, much of the effort of searching could be avoided.

Blind search or exhaustive methods try operators in some fixed order, without knowing which operators may be more likely to lead to a solution. Such methods can succeed only for small search spaces.

Heuristic search methods use knowledge about the problem domain to choose more promising operators first.

Exhaustive search

Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.

- Breadth-First Search

In This technique, the children (i.e the neighbour) of a node are first visited before the grand children (i.e. the neighbour of the neighbour) are visited.

1. Create a variable called NODE-LIST and set it to the initial state.

2. UNTIL a goal state is found OR NODE-LIST is empty DO

(a) Remove the first element from NODE_LIST and call it E.

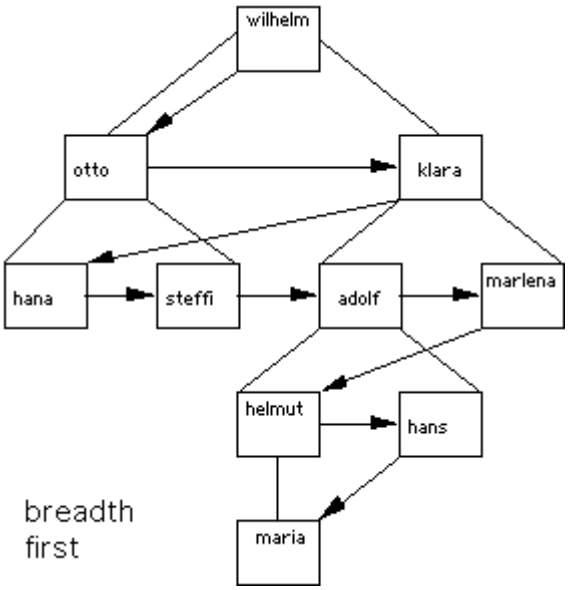
IF NODE-LIST was empty quit.

(b) FOR each way that each rule can match the state described in E DO

(i) Apply the rule to generate a new state

(ii) IF the new state is a goal state quit and return this state.

(iii) Otherwise add the new state to the end of NODE-LIST.



- **Algorithm Depth-First Search**

The depth first search follow a path to its end before stating to explore another path.

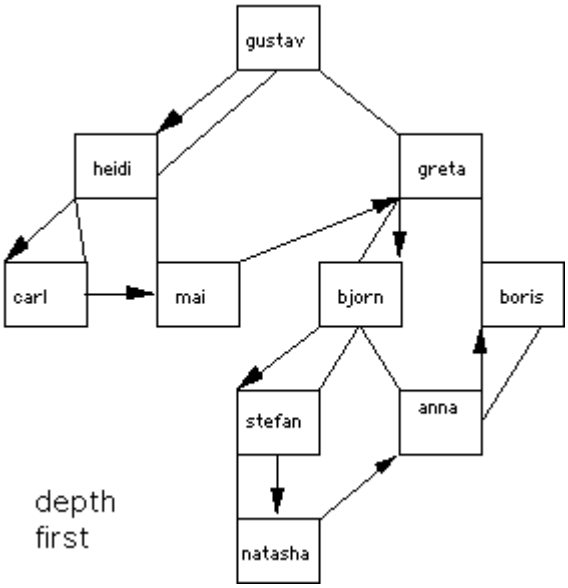
1. IF the initial state is a goal state, quit and return success.

2. Otherwise DO the following until success or failure is signalled

(a) Generate a successor, E, of the initial state. If there are no more successors signal failure.

(b) Call Depth-First Search with E as the initial state.

(c) If success is returned signal success otherwise continue in the loop.



Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.

Advantages:

1. Low storage requirement: *linear* with tree depth.
2. Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages:

1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).
2. Incomplete: without a depth bound, may not find a solution even if one exists.

- Bounded Depth-First Search

Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists.

An easy way to solve this problem is to put a maximum depth bound on the search. Beyond the depth bound , a failure is generated automatically without exploring any deeper.

Problems:

1. It's hard to guess how deep the solution lies.
2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of b^{extra}.

3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

- **Iterative Deepening**

Iterative deepening begins a search with a depth bound of 1, then increases the bound by 1 until a solution is found.

Advantages:

1. Finds an optimal solution (shortest number of steps).
2. Has the low (linear in depth) storage requirement of depth-first search.

Disadvantage:

1. Some computer time is wasted re-exploring the higher parts of the search tree. However, this actually is not a very high cost.
2. **Cost of Iterative Deepening**
3. In general, $(b - 1) / b$ of the nodes of a search tree are on the bottom row. If the branching factor is $b = 2$, half the nodes are on the bottom; with a higher branching factor, the proportion on the bottom row is higher.

Heuristics Search

A heuristic is a method that might not always find the best solution but is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency. It is particularly useful in solving tough problems which could not be solved any other way and if a complete solution was to be required infinite time would be needed i.e. far longer than a lifetime.

To use heuristics to find a solution in acceptable time rather than a complete solution in infinite time. The next example illustrates the requirement for heuristic search as it needs a very large time to find the exact solution.

Example: The travelling salesman problem

A salesperson has a list of cities to visit and she must visit each city only once. There are distinct routes between the cities. The problem is to find the shortest route between the cities so that the salesperson visits all the cities once.

Suppose there are N cities then a solution that would work would be to take all N! possible combinations and to find the shortest distance that being the required route. This is not efficient as with N=10 there are 3 628 800 possible routes. This is an example of combinatorial explosion.

There are better methods for solution, one is called branch and bound.

Generate all the complete paths and find the distance of the first complete path. If the next path is shorter save it and proceed in this way abandoning any path when its length so far exceeds the shortest path length. Although this is better than the previous method it is still exponential.

Heuristic Search applied to the travelling salesman problem

Applying this concept to the travelling salesperson problem.

1 select a city at random as a start point;

2 repeat

3 to select the next city have a list of all the cities to be visited and choose the nearest one to the current city , then go to it;

4 until all cities visited

This produces a significant improvement and reduces the time from order N! to N.

It is also possible to produce a bound on the error in the answer it generates but in general it is not possible to produce such an error bound.

In real problems the value of a particular solution is trickier to establish, this problem is easier as it is measured in miles, other problems have vaguer measures..

Although heuristics can be created for unstructured knowledge producing cogent analysis is another issue and this means that the solution lacks reliability.

Rarely is an optimal solution required good approximations usually suffice.

Although heuristic solutions are bad in the worst case the worst case occurs very infrequently and in the most common cases solutions now exist. Understanding why heuristics appear to work increases our understanding of the problem.

This method of searching is a general method which can be applied to problems of the following type.

Problem Characteristics.

- Is the problem decomposable into a set of nearly independent smaller or easier sub-problems?
- Can the solution steps be ignored or at least undone if they prove unwise?
- Is the problem's universe predictable?
- Is a good solution to the problem obvious without comparison to all other possible solutions?

- Is the desired solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve this problem or is knowledge important only to constrain the search?
- Can a computer that is simply given the problem return the solution or will the solution of the problem require interaction between the computer and a person?

The design of search programs.

Each search process can be considered to be a tree traversal exercise. The object of the search is to find a path from an initial state to a goal state using a tree. The number of nodes generated might be immense and in practice many of the nodes would not be needed. The secret of a good search routine is to generate only those nodes that are likely to be useful. Rather than having an explicit tree the rules are used to represent the tree implicitly and only to create nodes explicitly if they are actually to be of use.

The following issues arise when searching:

- the tree can be searched forwards from the initial node to the goal state or backwards from the goal state to the initial state.
- how to select applicable rules, it is critical to have an efficient procedure for matching rules against states.
- how to represent each node of the search process this is the knowledge representation problem or the frame problem. In games an array suffices in other problems more complex data structures are needed.

The breadth first does take note of all nodes generated but depth first can be modified.

Check duplicate nodes

- 1 examine all nodes already generated to see if new node is present.
- 2 if it does exist add it to the graph.
- 3 if it does already exist then
- a set the node that is being expanded to point to the already existing node corresponding to its successor rather than to the new one.
- The new one can be thrown away.
- b if the best or shortest path is being determined check to see if this path is better or worse than the old one.
- if worse do nothing.
- if better save the new path and work the change in length through the chain of successor nodes if necessary.

3. Knowledge representation

Much intelligent behavior is based on the use of knowledge; humans spend a third of their useful lives becoming educated. There is not yet a clear understanding of how the brain represents knowledge

Knowledge representation (KR) is an area in artificial intelligence that is concerned with how to formally "think", that is, how to use a symbol system to represent "a domain of discourse" that which can be talked about, along with functions that may or may not be within the domain of discourse that allow inference (formalized reasoning) about the objects within the domain of discourse to occur.

Knowledge representation is the study of how knowledge about the world can be represented and what kinds of reasoning can be done with that knowledge.

In order to use knowledge and reason with it, you need what we call a representation and reasoning system (RRS). A representation and reasoning system is composed of a language to communicate with a computer, a way to assign meaning to the language, and procedures to compute answers given input in the language. Intuitively, an RRS lets you tell the computer something in a language where you have some meaning associated with the sentences in the language, you can ask the computer questions, and the computer will produce answers that you can interpret according to the meaning associated with the language

There are several important issues in knowledge representation:

- how knowledge is *stored*;
- how knowledge that is applicable to the current problem can be *retrieved*;
- how *reasoning* can be performed to derive information that is implied by existing knowledge but not stored directly.

The storage and reasoning mechanisms are usually closely coupled.

It is necessary to represent the computer's knowledge of the world by some kind of data structures in the machine's memory. Traditional computer programs deal with large amounts of data that are structured in simple and uniform ways. A.I. programs need to deal with complex relationships, reflecting the complexity of the real world.

Typical problem solving (and hence many AI) tasks can be commonly reduced to:

- *representation* of input and output data as symbols in a physical symbol
- *reasoning* by processing symbol structures, resulting in other symbol structures.

Some problems highlight search whilst others knowledge representation. Several kinds of knowledge might need to be represented in AI systems:

- **Objects**

Facts about objects in our world domain. *e.g.* Guitars have strings, trumpets are brass instruments.

- **Events**

Actions that occur in our world. *e.g.* Steve Vai played the guitar in Frank Zappa's Band.

- **Performance**

A behavior like *playing the guitar* involves knowledge about how to do things.

- **Meta-knowledge**

knowledge about what we know. *e.g.* Bobrow's Robot who plan's a trip. It knows that it can read street signs along the way to find out where it is.

Thus in solving problems in AI we must represent knowledge and there are two entities to deal with:

- **Facts**

truths about the real world and what we represent. This can be regarded as the *knowledge level*

- **Representation of the facts**

which we manipulate. This can be regarded as the *symbol level* since we usually define the representation in terms of symbols that can be manipulated by programs.

We can structure these entities at two levels:

The knowledge level: at which facts are described

The symbol level: at which representations of objects are defined in terms of symbols that can be manipulated in programs

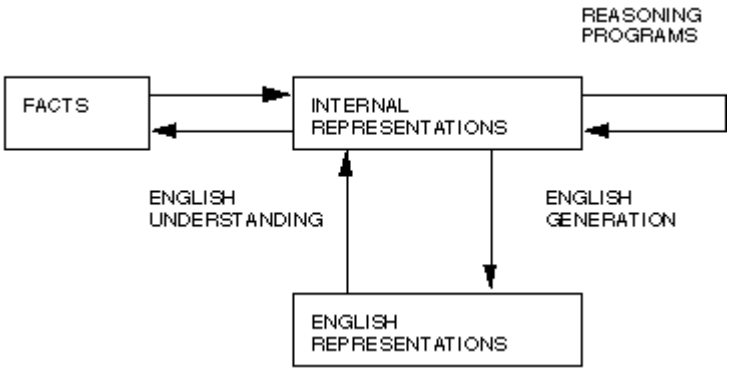


Fig: Two Entities in Knowledge Representation

English or natural language is an obvious way of representing and handling facts. Logic enables us to consider the following fact: *spot is a dog* as *dog(spot)* We could then infer that all dogs have tails with: ∀x: *dog(x)* →*hasatail(x)* We can then deduce:

hasatail(Spot)

Using an appropriate backward mapping function the English sentence *Spot has a tail can be generated.*

The available functions are not always one to one but rather are many to many which is a characteristic of English representations. The sentences *All dogs have tails* and *every dog has a tail* both say that each dog has a tail but the first could say that each dog has more than one tail try substituting teeth for tails. When an AI program manipulates the internal representation of facts these new representations should also be interpretable as new representations of facts.

Using Knowledge

We have briefly mentioned where knowledge is used in AI systems. Let us consider a little further to what applications and how knowledge may be used.

- **Learning**
It refers to acquiring knowledge. This is more than simply adding new facts to a knowledge base. New data may have to be *classified* prior to storage for easy *retrieval, etc.*. *Interaction* and *inference* with existing facts to avoid redundancy and replication in the knowledge and also so that facts can be updated.
- **Retrieval**
The representation scheme used can have a critical effect on the *efficiency* of the method. Humans are very good at it.
- **Reasoning**
Infer facts from existing data.

Properties for Knowledge Representation Systems

The following properties should be possessed by a knowledge representation system.

- **Representational Adequacy**
the ability to represent the required knowledge;
- **Inferential Adequacy**
the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;
- **Inferential Efficiency**
the ability to direct the inferential mechanisms into the most productive directions by storing appropriate guides;
- **Acquisitional Efficiency**

the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

To date no single system optimizes all of the above

Approaches to Knowledge Representation

- **Simple relational knowledge**

The simplest way of storing facts is to use a relational method where each fact about a set of objects is set out systematically in columns. This representation gives little opportunity for inference, but it can be used as the knowledge basis for inference engines.

- Simple way to store facts.
- Each fact about a set of objects is set out systematically in columns (Fig below)
- Little opportunity for inference.
- Knowledge basis for inference engines.

Musician	Style	Instrument	Age
Miles Davis	Jazz	Trumpet	deceased
John Zorn	Avant Garde	Saxophone	35
Frank Zappa	Rock	Guitar	deceased
John Mclaughlin	Jazz	Guitar	47

Figure: Simple Relational Knowledge

We can ask things like: Who is dead? Who plays Jazz/Trumpet *etc.*? This sort of representation is popular in database systems.

- **Inheritable knowledge**

Relational knowledge is made up of objects consisting of attributes and corresponding associated values.

Inheritable knowledge extends the base more by allowing inference mechanisms:

- Property inheritance

  Elements inherit values from being members of a class.

  data must be organized into a hierarchy of classes as shown in the figure below

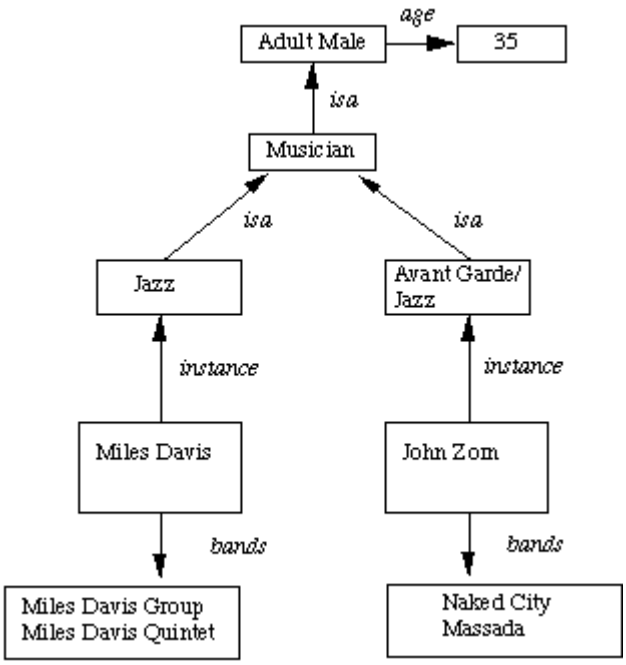


Fig. Property Inheritance Hierarchy

Boxed nodes represent objects and values of attributes of objects.
Values can be objects with attributes and so on.
Arrows point from object to its value.
This structure is known as a slot and filler structure, semantic network or a collection of frames.

The algorithm to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of instance if none fail
4. Otherwise go to that node and find a value for the attribute and then report it
5. Otherwise search through using *isa* until a value is found for the attribute.

Inferential Knowledge

Represent knowledge as *formal logic*: *All dogs have tails* $\forall x$: $dog(x) \rightarrow hasatail(x)$ Advantages:

- A set of strict rules.
  - Can be used to derive more facts.
  - Truths of new statements can be verified.
  - Guaranteed correctness.
- Many inference procedures available to in implement standard rules of logic.
- Popular in AI systems. *e.g* Automated theorem proving.

Procedural Knowledge

Basic idea:

- Knowledge encoded in some procedures
  - small programs that know how to do specific things, how to proceed.
  - *e.g* a parser in a natural language understander has the knowledge that a *noun phrase* may contain articles, adjectives and nouns. It is represented by calls to routines that know how to process articles, adjectives and nouns.

Advantages:

- *Heuristic* or domain specific knowledge can be represented.
- *Extended logical inferences*, such as default reasoning facilitated.
- *Side effects* of actions may be modelled. Some rules may become false in time. Keeping track of this in large systems may be tricky.

Disadvantages:

- Completeness -- not all cases may be represented.
- Consistency -- not all deductions may be correct.

  e.g If we know that *Fred is a bird* we might deduce that *Fred can fly*. Later we might discover that *Fred is an emu*.

- Modularity is sacrificed. Changes in knowledge base might have far-reaching effects.
- Cumbersome control information.

Issue in Knowledge Representation

Below are listed issues that should be raised when using a knowledge representation technique:

Important Attributes
Are there any attributes that occur in many different types of problem?

There are two *instance* and *isa* and each is important because each supports property inheritance.

Relationships
What about the relationship between the attributes of an object, such as, inverses, existence, techniques for reasoning about values and single valued attributes. We can consider an example of an inverse in

band(John Zorn,Naked City)

This can be treated as John Zorn plays in the band *Naked City* or John Zorn's band is *Naked City*.

Another representation is *band = Naked City*

band-members = John Zorn, Bill Frissell, Fred Frith, Joey Barron, . . .

Granularity
At what level should the knowledge be represented and what are the primitives. Choosing the Granularity of Representation Primitives are fundamental concepts such as holding, seeing, playing and as English is a very rich language with over half a million words it is clear we will find difficulty in deciding upon which words to choose as our primitives in a series of situations.

If *Tom feeds a dog* then it could become:

feeds(tom, dog)

If *Tom gives the dog a bone* like:

gives(tom, dog,bone) Are these the same?

In any sense does giving an object food constitute feeding?

If *give*(*x, food*) $\rightarrow$*feed(x)* then we are making progress.

But we need to add certain inferential rules.

In the famous program on relationships *Louise is Bill's cousin* How do we represent this? *louise = daughter (brother or sister (father or mother(bill)))* Suppose it is *Chris* then we do not know if it is *Chris* as a male or female and then *son* applies as well.

Clearly the separate levels of understanding require different levels of primitives and these need many rules to link together apparently similar primitives. Obviously there is a potential storage problem and the underlying question must be what level of comprehension is needed.

Logic Knowledge Representation

Here we will highlight major principles involved in knowledge representation. In particular *predicate logic* will be met in other knowledge representation schemes and reasoning methods.

Symbols used The following standard logic symbols we use in this course are:

For all ∀
There exists ∃

Implies →
Not ¬
Or ∨

And ∧

Let us now look at an example of how predicate logic is used to represent knowledge. There are other ways but this form is popular.

Predicate logic

An example

Consider the following:

- Prince is a mega star.
- Mega stars are rich.
- Rich people have fast cars.
- Fast cars consume a lot of petrol.

and try to draw the conclusion: *Prince's car consumes a lot of petrol*.

So we can translate *Prince is a mega star* into: *mega_star(prince)* and *Mega stars are rich* into: ∀*m: mega_star(m)* →*rich(m)*

Rich people have fast cars, the third axiom is more difficult:

- Is *cars* a relation and therefore *car(c,m)* says that case *c* is *m's* car. **OR**
- Is *cars* a function? So we may have *car_of(m)*.

Assume *cars* is a relation then axiom 3 may be written: ∀*c,m:car(c,m)* ∧*rich(m)*→*fast(c).*

The fourth axiom is a general statement about *fast cars*. Let *consume(c)* mean that car *c* consumes a lot of petrol. Then we may write: ∀*c:[fast(c)* ∧∃ *m:car(c,m)*→*consume(c)]*.

Is this enough? NO! -- Does prince have a car? We need the *car_of* function after all (and addition to *car*): ∀ *c:car(car_of(m),m)*. The result of applying *car_of* to *m* is *m*'s car. The final set of predicates is: *mega_star(prince)* ∀*m: mega_star(m)* →*rich(m)*

$\forall c:car(car\_of(m),m)$. $\forall c,m:\ car(c,m) \wedge rich(m) \rightarrow fast(c)$. $\forall c:[\ fast(c) \wedge \exists m:car(c,m) \rightarrow consume(c)]$. Given this we could conclude: *consume(car_of(prince)).*

Isa and *instance* relationships

Two attributes *isa* and *instance* play an important role in many aspects of knowledge representation. The reason for this is that they support *property inheritance*.

isa
used to show class inclusion, *e.g. isa(mega_star,rich).*
instance
used to show class membership, *e.g. instance(prince,mega_star).*

From the above it should be simple to see how to represent these in predicate logic.

Applications and extensions

- First order logic basically extends predicate calculus to allow:
  - functions -- return *objects* not just TRUE/FALSE.
  - equals predicate added.
- Problem solving and theorem proving -- large application areas.
- STRIPS robot planning system employs a first order logic system to enhance its means-ends analysis (GPS) planning. This amalgamation provided a very powerful heuristic search.
- Question answering systems.

Procedural Knowledge Representations

Declarative or Procedural?

Declarative knowledge representation:

- Static representation -- knowledge about objects, events *etc.* and their relationships and states given.
- Requires a program to know what to do with knowledge and how to do it.

Procedural representation:

- control information necessary to use the knowledge is embedded in the knowledge itself. *e.g.* how to find relevant facts, make inferences *etc.*
- Requires an interpreter to follow instructions specified in knowledge.

An Example

Let us consider what knowledge an alphabetical sorter would need:

- Implicit knowledge that *A* comes before *B etc.*
- This is easy -- really integer comparison of (ASCII) codes for A_1, B_1,….
    - All programs contain procedural knowledge of this sort.
- The procedural information here is that knowledge of *how to alphabetize* is represented explicitly in the alphabetisation procedure.
    - A declarative system might have to have explicit facts like *A* comes before *B*, *B* comes before *C etc.*.

Representing How to Use Knowledge

Need to represent *how to control* the processing:

direction

- Indicate the direction an implication could be used. *E.g.* To prove something can fly show it is a bird. $fly(x) \rightarrow bird(x)$.

Knowledge to achieve goal

- Specify what knowledge might be needed to achieve a specific goal. For example to prove something is a bird try using two facts *has_wings* and *has_feathers* to show it.

Weak Slot and Filler Structures

We have already met this type of structure when discussing inheritance in the last lecture. We will now study this in more detail.

Why use this data structure?

- It enables attribute values to be retrieved quickly
    - assertions are indexed by the entities
    - binary predicates are indexed by first argument. *E.g. team(Mike-Hall , Cardiff).*
- Properties of relations are easy to describe .
- It allows ease of consideration as it embraces aspects of object oriented programming.

So called because:

- A *slot* is an attribute value pair in its simplest form.
- A *filler* is a value that a slot can take -- could be a numeric, string (or any data type) value or a pointer to another slot.
- A *weak* slot and filler structure does not consider the *content* of the representation.

We will study two types:

- Semantic Nets.
- Frames.

Semantic Nets

The major idea is that:

- The meaning of a concept comes from its relationship to other concepts, and that,
- The information is stored by interconnecting nodes with labelled arcs.

Representation in a Semantic Net

The physical attributes of a person can be represented as in the figure bellow

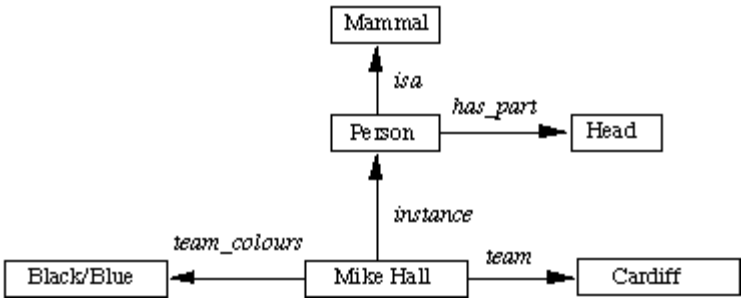


Fig. A Semantic Network

These values can also be represented in logic as: *isa(person, mammal), instance(Mike-Hall, person) team(Mike-Hall, Cardiff)*

We have already seen how conventional predicates such as *lecturer(dave)* can be written as *instance (dave, lecturer)* Recall that *isa* and *instance* represent inheritance and are popular in many knowledge representation schemes. But we have a problem: *How we can have more than 2 place predicates in semantic nets? E.g. score(Cardiff, Llanelli, 23-6)* Solution:

- Create new nodes to represent new objects either contained or alluded to in the knowledge, *game* and *fixture* in the current example.
- Relate information to nodes and fill up slots (Fig: 10).

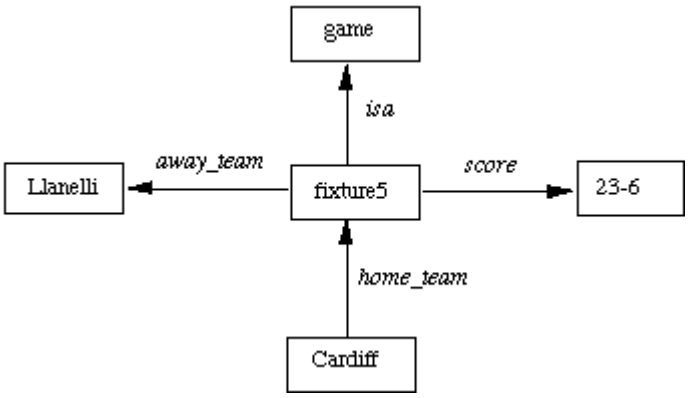


Fig. A Semantic Network for *n*-Place Predicate

As a more complex example consider the sentence: *John gave Mary the book*. Here we have several aspects of an event.

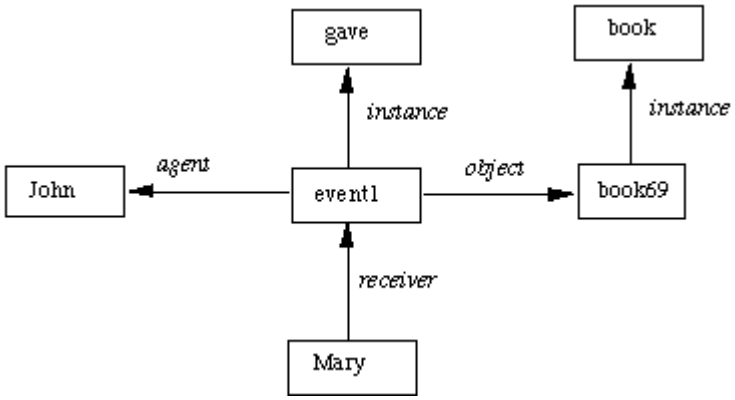


Fig. A Semantic Network for a Sentence

Inference in a Semantic Net

Basic inference mechanism: *follow links between nodes*.

Two methods to do this:

Intersection search

-- the notion that *spreading activation* out of two nodes and finding their intersection finds relationships among objects. This is achieved by assigning a special tag to each visited node.

Many advantages including entity-based organisation and fast parallel implementation. However very structured questions need highly structured networks.

Inheritance

-- the *isa* and *instance* representation provide a mechanism to implement this.

Inheritance also provides a means of dealing with *default reasoning*. *E.g.* we could represent:

- Emus are birds.
- Typically birds fly and have wings.
- Emus run.

in the following Semantic net:

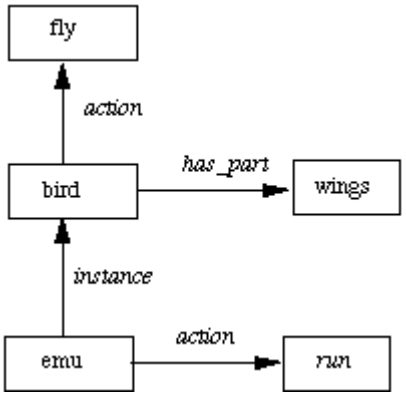


Fig. A Semantic Network for a Default Reasoning

In making certain inferences we will also need to *distinguish between the link that defines a new entity and holds its value and the other kind of link that relates two existing entities*. Consider the example shown where the height of two people is depicted and we also wish to compare them.

We need extra nodes for the concept as well as its value.

Fig. Two heights

Special procedures are needed to process these nodes, but without this distinction the analysis would be very limited.

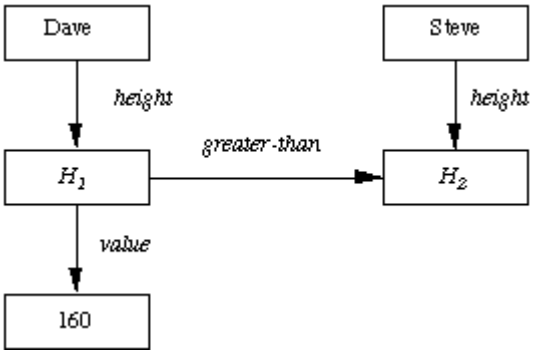


Fig. Comparison of two heights

Extending Semantic Nets

Here we will consider some extensions to Semantic nets that overcome a few problems (see Exercises) or extend their expression of knowledge.

Partitioned Networks *Partitioned* Semantic Networks allow for:

- propositions to be made without commitment to truth.
- expressions to be quantified.

Basic idea: *Break network into spaces which consist of groups of nodes and arcs and regard each space as a node.*

Consider the following: *Andrew believes that the earth is flat.* We can encode the proposition *the earth is flat* in a *space* and within it have nodes and arcs the represent the fact (Fig. 15). We can the have nodes and arcs to link this *space* the the rest of the network to represent Andrew's belief.

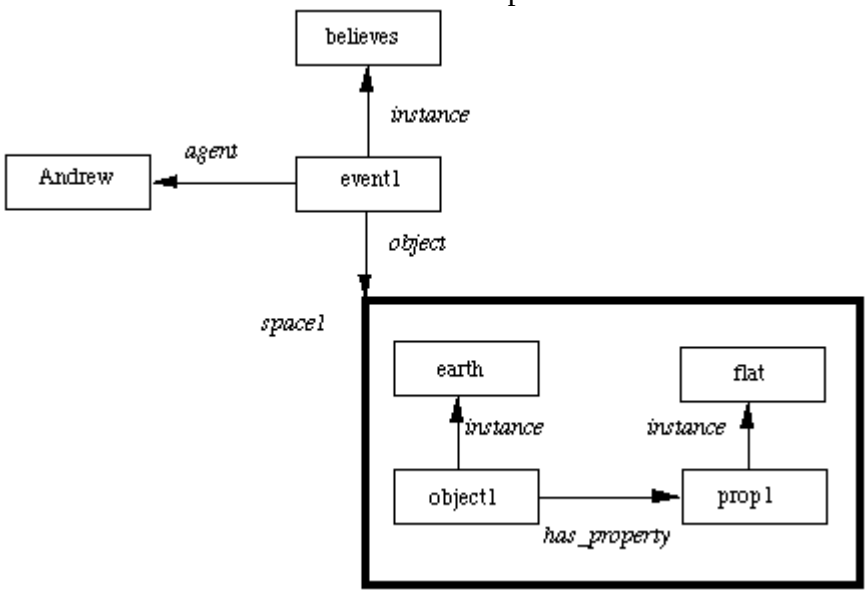


Fig. Partitioned network

Now consider the quantified expression: *Every parent loves their child* To represent this we:

- Create a *general statement*, GS, special class.
- Make node g an instance of GS.
- Every element will have at least 2 attributes:
  - a *form* that states which relation is being asserted.
  - one or more *forall* (∀) or *exists* (∃) connections -- these represent universally quantifiable variables in such statements *e.g. x, y* in $\forall x$ *parent(x)*$\rightarrow\exists y$: *child(y)* $\wedge$ *loves(x,y)*

Here we have to construct two *spaces one* for each *x,y*. **NOTE:** We can express ∃variables as *existentially qualified* variables and express the event of *love* having an agent *p* and receiver *b* for every parent *p* which could simplify the network (See Exercises).

Also If we change the sentence to *Every parent loves <u>child</u>* then the node of the object being acted on (*the child*) lies outside the form of the general statement. Thus it is not viewed as an existentially qualified variable whose value may depend on the agent. (See Exercises and Rich and Knight book for examples of this) So we could construct a partitioned network as in the figure below

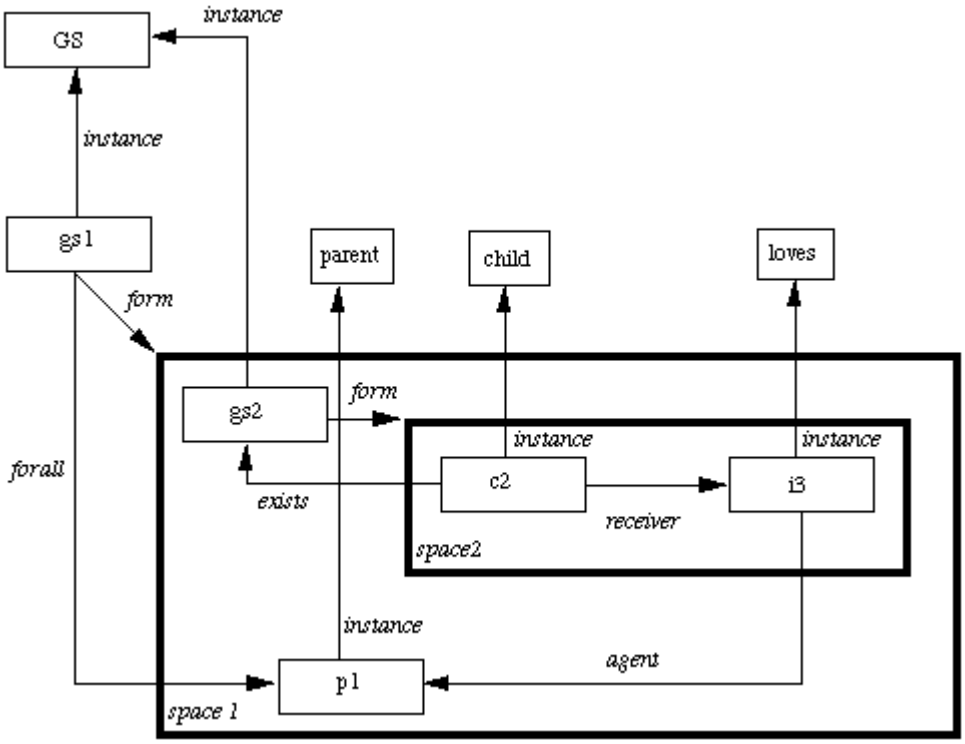


Fig. Partitioned network

Frames

Frames can also be regarded as an extension to Semantic nets. Indeed it is not clear where the distinction between a semantic net and a frame ends. Semantic nets initially we used to represent labelled connections between objects. As tasks became more complex the representation needs to be more structured. The more structured the system it becomes more beneficial to use frames. A *frame* is a collection of attributes or slots and associated values that describe some real world entity. Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning. Set theory provides a good basis for understanding frame systems. Each frame represents:

- a class (set), or
- an instance (an element of a class).

Frame Knowledge Representation

Consider the example first discussed in Semantics Nets:

Person
isa: *Mammal*
Cardinality: ...
Adult-Male
isa: *Person*
Cardinality: ...
Rugby-Player
isa: *Adult-Male*
Cardinality: ...
Height:
Weight:
Position:
Team:
Team-Colours:
Back
isa: *Rugby-Player*
Cardinality: ...
Tries:
Mike-Hall
instance: *Back*
Height: *6-0*
Position: *Centre*
Team: *Cardiff-RFC*
Team-Colours: *Black/Blue*
Rugby-Team
isa: *Team*
Cardinality: ...
Team-size: *15*
Coach:

Cardiff-RFC
Instance: Rugby-Team
Team –size:15
Coach: Terry Holmes

Figure: A simple frame system

Here the frames *Person, Adult-Male, Rugby-Player and Rugby-Team* are all **classes** and the frames *Robert-Howley* and *Cardiff-RFC* are instances.

Note

- The *isa* relation is in fact the subset relation.
- The *instance* relation is in fact *element of*.
- The *isa* attribute possesses a transitivity property. This implies: *Robert-Howley* is a *Back* and a *Back* is a *Rugby-Player* who in turn is an *Adult-Male* and also a *Person*.
- Both *isa* and *instance* have inverses which are called subclasses or all instances.
- There are attributes that are associated with the class or set such as cardinality and on the other hand there are attributes that are possessed by each member of the class or set.

DISTINCTION BETWEN SETS AND INSTANCES

It is important that this distinction is clearly understood.

Cardiff-RFC can be thought of as a set of players or as an instance of a *Rugby-Team*.

If *Cardiff-RFC* were a *class* then

- its instances would be players
- it could not be a subclass of *Rugby-Team* otherwise its elements would be members of Rugby-Team which we do not want.

Instead we make it a subclass of *Rugby-Player* and this allows the players to inherit the correct properties enabling us to let the *Cardiff-RFC* to inherit information about teams.

This means that *Cardiff-RFC* is an instance of *Rugby-Team*.

BUT There is a problem here:

- A class is a set and its elements have properties.
- We wish to use inheritance to bestow values on its members.
- But there are properties that the set or class itself has such as the manager of a team.

This is why we need to view *Cardiff-RFC* as a subset of one class players and an instance of teams. We seem to have a CATCH 22. *Solution*: **MetaClasses**

A metaclass is a special class whose elements are themselves classes.

Now consider our rugby teams as:

Class

instance:	*Class*
isa:	*Class*
Cardinality:	...

Team

instance:	*Class*
isa:	*Class*
Cardinality:	{ The number of teams }
Team-Size:	*15*

Rugby-Team

isa:	*Team*
Cardinality:	{ The number of teams }
Team-size: 15	
Coach:	

Cardiff-RFC

instance:	*Rugby-Team*
Team-size:	*15*
Coach:	*Terry Holmes*

Robert-Howley

instance:	*Back*
Height:	*6-0*
Position:	*Scrum Half*
Team:	*Cardiff-RFC*
Team-Colours:	*Black/Blue*

Figure: A Metaclass frame system

The basic metaclass is *Class*, and this allows us to

- define classes which are instances of other classes, and (thus)
- inherit properties from this class.

Inheritance of default values occurs when one element or class is an instance of a class.

Slots as Objects

How can we to represent the following properties in frames?

- Attributes such as *weight, age* be attached and make sense.
- Constraints on values such as *age* being less than a hundred

- Default values
- Rules for inheritance of values such as children inheriting parent's names
- Rules for computing values
- Many values for a slot.

A slot is a relation that maps from its domain of classes to its range of values.

A relation is a set of ordered pairs so one relation is a subset of another.

Since slot is a set the set of all slots can be represent by a metaclass called *Slot*, say.

Consider the following:

SLOT

isa:	*Class*	
instance:		*Class*
domain:		
range:		
range-constraint:		
definition:		
default:		
to-compute:		
single-valued:		

Coach

instance:		*SLOT*
domain:		*Rugby-Team*
range:	*Person*	
range-constraint:		*λx (experience x.manager)*
default:		
single-valued:		*TRUE*

Colour

instance:		*SLOT*
domain:		*Physical-Object*
range:	*Colour-Set*	
single-valued:		*FALSE*

Team-Colours

instance:		*SLOT*
isa:	*Colour*	
domain:		*team-player*
range:	*Colour-Set*	
range-constraint:		*not Pink*
single-valued:		*FALSE*

Position

instance:		*SLOT*

domain:	*Rugby-Player*
range:	*{ Back, Forward, Reserve }*
to-compute:	*λx x.position*
single-valued:	*TRUE*

NOTE the following:

- Instances of *SLOT* are slots
- Associated with *SLOT* are attributes that each instance will inherit.
- Each slot has a domain and range.
- Range is split into two parts one the class of the elements and the other is a constraint which is a logical expression if absent it is taken to be true.
- If there is a value for default then it must be passed on unless an instance has its own value.
- The *to-compute* attribute involves a procedure to compute its value. *E.g.* in *Position* where we use the dot notation to assign values to the slot of a frame.
- Transfers through lists other slots from which values can be derived from inheritance.

Interpreting frames

A frame system interpreter must be capable of the following in order to exploit the frame slot representation:

- Consistency checking -- when a slot value is added to the frame relying on the domain attribute and that the value is legal using range and range constraints.
- Propagation of *definition* values along *isa* and *instance* links.
- Inheritance of default. values along *isa* and *instance* links.
- Computation of value of slot as needed.
- Checking that only correct number of values computed.

4 Expert system

Expert system is programs that attempt to perform the duty of an expert in the problem domain in which it is defined.

Expert systems are computer programs that have been constructed (with the assistance of human experts) in such a way that they are capable of functioning at the standard of (and sometimes even at a higher standard than) human experts in given fields that embody a depth and richness of knowledge that permit them to perform at the level of an expert.

Rule based system

Using a set of assertions, which collectively form the 'working memory', and a set of rules that specify how to act on the assertion set, a rule-based system can be created. Rule-based systems are fairly simplistic, consisting of little more than a set of if-then statements, but provide the basis for so-called "expert systems" which are widely used in many fields. The concept of an expert system is this: the knowledge of an expert is encoded into the rule set. When exposed to the same data, the expert system will perform in a similar manner as the expert.

Element of rule based Expert System

Rule-based systems are a relatively simple model that can be adapted to any number of problems. To create a rule-based system for a given problem, you must have (or create) the following:

- A set of facts to represent the initial working memory. This should be anything relevant to the beginning state of the system.
- A set of rules. This should encompass any and all actions that should be taken within the scope of a problem, but nothing irrelevant. The number of rules in the system can affect its performance, so you don't want any that aren't needed.
- A condition that determines that a solution has been found or that none exists. This is necessary to terminate some rule-based systems that find themselves in infinite loops otherwise.

In fact, there are three essential components to a fully functional rule based expert system: the knowledge base, the working memory and the inference engine.

- **The knowledge base.**

The knowledge based is the store in which the knowledge in the particular domain is kept. The knowledge base stores information about the subject domain. However, this goes further than a passive collection of records in a database. Rather it contains symbolic representations of experts' knowledge, including definitions of domain terms, interconnections of component entities, and cause-effect relationships between these components. The knowledge in the knowledge based is expressed as a collection of

fact and rule. Each fact expresses relationship between two or more object in the problem domain and can be expressed in term of predicates

IF condition THEN conclusion where the condition or conclusion are fact or sets of fact connected by the logical connectives NOT, AND, OR. Note that we need to create a variable name list to help to deal with long clumsy name and to help writing the rule.

Variable name	**Meaning**
Interest rise	interest rate rise
Interest fall	interest rate fall
Stock rise	market stock fall
Stock fall	market stock fall
Naira rise	exchange rate rise
Naira fall	exchange rate fall
Inflation rise	cost of market product rise
Fedmont add	increment of federal reserve money in circulation
Tax add	taxation on market product increase
Tax fall	taxation on market product decrease

Using the above variable name, the following set of rule can then be constructed.

Rule 1: IF interest rise THEN stock fall

Rule 2: IF interest fall THEN stock rise

Rule 3: IF naira fall THEN interest fall

Rule 4: IF naira rise THEN interest rise

Rule 5 IF stock fall THEN inflation rise

Rule 6 IF fedmont add AND tax fall THEN naira rise

We shall note that the IF THEN rules are treated very differently from similar constructs in a natural programming language. While natural programming languages treats IF-THEN construct as part of a sequence of instructions, to be considered in order, the rule based system treats each rule as an independent chunk of knowledge, to be invoked when needed under the control of the interpreter. The rules are more like implication in logic.(e.g. naira rise →interest rise).

- The working memory

The working memory is a temporal store that hold the fact produced during processing and possibly awaiting further processing produced by the Inference engine during its activities. Note that the working memory contains only facts and these fact are those produced during the searching process.

- The inference engine.

The core of any expert system is its inference engine. This is the part of expert system that manipulates the knowledge based to produce new fact in order to solve the given problem. An inference engine consists of search and reasoning procedures to enable the system to find solutions, and, if necessary, provide justifications for its answers. In this process it can used either forward or backward searching as a direction of search while applying some searching technique such as depth first search, breath first search etc.

The roles of inference engine are:

- It identified the rule to be fired. The rule selected is the one whose conditional part is the same as the fact been considered in the case of forward chaining or the one whose conclusion part is the one as the fact been considered in the case of backward chaining.

- It resolve conflict when more than one rule satisfy the matching this is called conflict resolution which is based on certain criteria mentioned further.

- It recognizes the goal state. When the goal state is reached it report the conclusion of searching.

Theory of Rule-Based Systems

The rule-based system itself uses a simple technique: It starts with a knowledge-base, which contains all of the appropriate knowledge encoded into IF-THEN rules, and a working memory, which may or may not initially contain any data, assertions or initially known information. The system examines all the rule conditions (IF) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy. When the rule is fired, any actions specified in its THEN clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This

loop of firing rules and performing actions continues until one of two conditions is met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate.

Which rule is chosen to fire is a function of the conflict resolution strategy. Which strategy is chosen can be determined by the problem or it may be a matter of preference. In any case, it is vital as it controls which of the applicable rules are fired and thus how the entire system behaves. There are several different strategies, but here are a few of the most common:

First Applicable: If the rules are in a specified order, firing the first applicable one allows control over the order in which rules fire. This is the simplest strategy and has a potential for a large problem: that of an infinite loop on the same rule. If the working memory remains the same, as does the rule-base, then the conditions of the first rule have not changed and it will fire again and again. To solve this, it is a common practice to suspend a fired rule and prevent it from re-firing until the data in working memory, that satisfied the rule's conditions, has changed.

Random: Though it doesn't provide the predictability or control of the first-applicable strategy, it does have its advantages. For one thing, its unpredictability is an advantage in some circumstances (such as games for example). A random strategy simply chooses a single random rule to fire from the conflict set. Another possibility for a random strategy is a fuzzy rule-based system in which each of the rules has a probability such that some rules are more likely to fire than others.

Most Specific: This strategy is based on the number of conditions of the rules. From the conflict set, the rule with the most conditions is chosen. This is based on the assumption that if it has the most conditions then it has the most relevance to the existing data.

Least Recently Used: Each of the rules is accompanied by a time or step stamp, which marks the last time it was used. This maximizes the number of individual rules that are fired at least once. If all rules are needed for the solution of a given problem, this is a perfect strategy.

Best rule: For this to work, each rule is given a 'weight,' which specifies how much it should be considered over the alternatives. The rule with the most preferable outcomes is chosen based on this weight.

Direction of searching

There are two broad kinds of direction of searching in a rule-based system: forward chaining systems, and backward chaining systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go. Forward

chaining systems are primarily data-driven, while backward chaining systems are goal-driven. We'll look at both, and when each might be useful.

- Forward Chaining Systems

In a forward chaining system the facts in the system are represented in a working memory which is continually updated as rules are invoked. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually patterns that must match items in the working memory, while the actions usually involve adding or deleting items from the working memory.

The inference engine controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a recognize-act cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. (The selection of a rule to fire is based on fixed strategies, known as conflict resolution strategies.) The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied.

Example:

Rule 1: IF interest rise THEN stock fall

Rule 2: IF interest fall THEN stock rise

Rule 3: IF naira fall THEN interest fall

Rule 4: IF naira rise THEN interest rise

Rule 5: IF stock fall THEN inflation rise

Rule 6 IF fedmont add AND tax fall THEN naira rise

Question: what is the impact if the federal government increases the amount of money in circulation? I.e. fedmont add

The working memory is thus having the fact

Fedmont add

The inference engine will first go through all the rules checking which ones has the conditional part which is the same as the fact in the current working memory. It finds it at rule 6. Rule 6 is thus selected. But the second clause of rule 6 is not yet in the working memory, the system will thus prompt for the value of tax, let assume the user supply fall

as a answer then since the two clauses are true then the rule is executed and the conclusion part become a new fact which is added to the working memory which is now:

Naira rise

Tax fall

Fedmont add

Now the cycle begins again. Rule 4 has its precondition satisfied that is it is the same as the fact “naira rise”. Rule is chosen and fires, so “Interest rise” is added to the working memory which is now

Interest rise

Naira rise

Tax fall

Fedmont add

Now the cycle begins again. This time rule 1has its precondition satisfied that is it is the same as” interest rise”. Rule 1 is chosen and fires, so” stock fall” is added to the working memory which is now:

Stock fall

Interest rise

naira rise

tax fall

fedmont add

Now rules 5 can apply. And in the next cycle rule 5 is chosen and fires, and “inflation rise” is added to the working memory.

The system continue and search for the conditional part of the rule which is the same as” inflation rise”, since no such rule exist then the system stop, and the report of the impact of the government adding the amount of money in circulation is:” inflation rate rise,”

A number of conflict resolution strategies are typically used to decide which rule to fire. These strategies may help in getting reasonable behavior from a forward chaining system,

but the most important thing is how we write the rules. They should be carefully constructed, with the preconditions specifying as precisely as possible when different rules should fire. Otherwise we will have little idea or control of what will happen. Sometimes special working memory elements are used to help to control the behavior of the system. For example, we might decide that there are certain basic stages of processing in doing some task, and certain rules should only be fired at a given stage

- Backward Chaining Systems

So far we have looked at how rule-based systems can be used to draw new conclusions from existing data, adding these conclusions to a working memory. This approach is most useful when you know all the initial facts, but don't have much idea what the conclusion might be.

If you DO know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient. You COULD keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory. But in the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to working memory.

This can be done by backward chaining from the goal state (or on some state that we are interested in). Given a goal state to try and prove (e.g., inflation rise) the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions (previously referred to as actions) match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. Note that a backward chaining system does not need to update a working memory. Instead it needs to keep track of what goals it needs to prove its main hypothesis.

In principle we can use the same set of rules for both forward and backward chaining. However, in practice we may choose to write the rules slightly differently if we are going to be using them for backward chaining. In backward chaining we are concerned with matching the conclusion of a rule against some goal that we are trying to prove. So the 'then' part of the rule is usually not expressed as an action to take but as a state which will be true if the premises are true.

Suppose we have the following rules:

Rule 1: IF interest rise THEN stock fall

Rule 2: IF interest fall THEN stock rise

Rule 3: IF naira fall THEN interest fall

Rule 4: IF naira rise THEN interest rise

Rule 5: IF stock fall THEN inflation rise

Rule 6 IF fedmont add AND tax fall THEN naira rise.

Suppose we want to find the cause of the increment of inflation. I.e. inflation rise

Initial fact

Fedmont add and tax fall

Naira fall

First we check if inflation rise is in the initial fact. If it is not there, try matching it against the conclusions of the rules. It matches rule5 then rule 5 is chosen and the conditional part becomes the new goal state to target. This introduce thus “stock fall”. It will try to prove “stock fall”. Since “stock fall” is found at the conclusion part of Rule 1, then rule 1 is selected and the conditional part of rule 1 that is “interest rise” is the new goal state, and the system will try to prove “interest rise”. This is found in the conclusion part of rule 5. Then rule 5 is selected and the conditional part (naira rise) becomes the new goal state to prove. This is found at the conclusion part of rule 6 then rule 6 is selected. The conditional part of rule 6 introduces two facts: “tax fall” and “fedmont rise”. Since no such facts are in the conclusion part of any rule then the search stop. We have thus found the cause of the inflation to rise. The system will thus output: “fedmont rise” and “tax fall”. That is the government has increased the amount of money in circulation.

One way of implementing this basic mechanism is to use a stack of goals still to satisfy. You should repeatedly pop a goal of the stack, and try and prove it. If it’s in the set of initial facts then it’s proved. If it matches a rule which has a set of preconditions then the goals in the precondition are pushed onto the stack. Of course, this doesn't tell us what to do when there are several rules, which may be used to prove a goal. If we were using Prolog to implement this kind of algorithm we might rely on its backtracking mechanism. It will try one rule, and if that results in failure it will go back and try the other. However, if we use a programming language without a built in search procedure we need to decide explicitly what to do. One good approach is to use an agenda, where each item on the agenda represents one alternative path in the search for a solution. The system should try `expanding' each item on the agenda, systematically trying all possibilities until it finds a solution (or fails to). The particular method used for selecting items off the agenda determines the search strategy - in other words, determines how you decide on which options to try, in what order, when solving your problem.

- Forward versus Backward Reasoning

Whether you use forward or backward reasoning to solve a problem depends on the properties of your rule set and initial facts. Sometimes, if you have some particular goal (to test some hypothesis), then backward chaining will be much more efficient, as you avoid drawing conclusions from irrelevant facts. However, sometimes backward chaining

can be very wasteful - there may be many possible ways of trying to prove something, and you may have to try almost all of them before you find one that works. Forward chaining may be better if you have lots of things you want to prove (or if you just want to find out in general what new facts are true); when you have a small set of initial facts; and when there tend to be lots of different rules which allow you to draw the same conclusion. Backward chaining may be better if you are trying to prove a single fact, given a large set of initial facts, and where, if you used forward chaining, lots of rules would be eligible to fire in any cycle.

Techniques of searching

The order that rules fire may be crucial, especially when rules may result in items being deleted from working memory. The system must implement a searching technique that is used to process the knowledge base. Some of these technique are:

- Depth first search.

In depth first search technique, the most recently fact added to the working memory is first selected for processing. We can thus implement it using stack so that the rule we have recently added to the working memory will be the one to be selected for the next cycle.

- Breadth first search.

In the breath first search technique, the fact selected in the working memory for processing are selected in the order in which they were added in the working memory. We can use queue data structure to implement it. Since the rule to be processed will be selected in the front of the queue and the new fact are added at the rear of the queue.

Reasoning under uncertainty.

So far, when we have assumed that if the preconditions of a rule hold, then the conclusion will certainly hold. In fact, most of our rules have looked pretty much like logical implications, and the ideas of forward and backward reasoning also apply to logic-based approaches to knowledge representation and inference.

Of course, in practice you rarely conclude things with absolute certainty. For example, we may have a rule IF fuel=rise THEN transport=rise. This rule may not be totally true. They may be a case that the risen of the transport fees is not caused by the risen of fuel price. If we were reasoning backward, we suppose to conclude that the fuel has risen which is not the case. For this sort of reasoning in rule-based systems we often add certainty values to a rule, and attach certainties to any new conclusions. We might conclude fuel rise if the transport rise maybe with certainty 0.6). The approaches used are generally loosely based on probability theory, but are much less rigorous, aiming just for a good guess rather than precise probabilities.

- Method based on probability

Bayes developed probability technique that is based on prediction that something will happen because of the evidence that something has happened in the pass. This probability is called conditional probability.

Review of Bayesian probability

By definition, the probability of occurrence of an event say B knowing that an event A has occurred in the pass in called conditional probability of occurrence of B and is denoted P(A/B) and is expressed by the formula

P(B/A) = P(A AND B)/P(A) from which we can derive

P(A AND B)= P(B/A)*P(A). (*)

We assume that A has occurred first. If B was the first to occur then we obtain.

P(A/B) = P(A AND B)/P(B) from which we can derive

P(A AND B)= P(A/B)*P(B). (**)

By combining the two formulas (*) and (**) to eliminate P(A and B) we obtain

P(B/A)*P(A) = P(A/B)*P(B) thus,

P(B/A)= P(A/B)*P(B)/P(A) (***)

Let two events B and NOT B by applying the formula (***), we obtain

P(B/A)= P(A/B)*P(B)/P(A)

P(NOT B/A)= P(A/NOT B)*P(NOT B)/P(A)

Since P(B/A) + P(NOT B/A) = 1 then we obtain

P(A/B)*P(B)/P(A) + P(A/NOT B)*P(NOT B)/P(A) =1 thus,

P(A)=P(A/B)*P(B)+P(A/NOT B)P(NOT B).

By replacing P(A) by in equation P(B/A)= P(A/B)*P(B)/P(A) we obtain

P(B/A)= P(A/B)*P(B)/[P(A/B)*P(B)+P(A/NOT B)P(NOT B)] which is the probability of an event say B to occurs knowing that an event A has occurs. That is the probability of the rule IF A THEN B to hold

EXAMPLE.

Consider the following rules

Rule 1: IF fuel rise THEN transport rise

Rule 2: IF naira fall THEN fuel Fall

Rule 3: IF fuel fall THEN transport fall

Rule 4: IF naira rise THEN fuel rise

Let us find the probability that transport rise knowing that fuel rise. That is the probability of rule 1.

Working backward, we find transport rise at a conclusion part of rule 1. by applying the above formula, we obtain

P(transport rise/fuel rise)= P(fuel rise/transport rise)*P(transport rise) /

[P(fuel rise/transport rise)*P(transport rise) +P(fuel rise/transport fall)* P(transport fall)]

Since there is no rule satisfying fuel rise knowing that transport rise and fuel rise knowing the transport fall, which is the rule IF transport rise then fuel rise and IF transport fall THEN fuel rise respectively, then the search stop and their probability have to be supplied. We shall note that, if they were rules satisfying them, the cycle will continue an in the same way, we will evaluate their own probability.

Let assume the following value are given.

P(fuel rise/transport rise)=0.6

P(transport rise) =0.7

P(fuel rise/transport fall)=0.2

P(transport fall)=0.3

Then we can calculate find the probability that transport rise knowing that fuel rise.

P(transport rise/fuel rise)=(0.6*0.7)/(0.6*07+0.2*0.3)= 0.875

Thus the probability for rule 1 to hold is 0.875.

- Method based on fuzzy logic

In this method of inexact knowledge, we associate to each fact an rule a number indicating the degree at which one is certain of its truthfulness. This number is called certainty factor (CF) and is taken in the interval[-1 1]. CF=1 means that the conclusion is certain to be true if the conditions are completely satisfied. While CF= -1 means that the conclusion is certain to be false under the same conditions. Otherwise, a positive value for CF denotes that the conditions constitute suggestive evidence for the conclusion to hold while a negative value denotes that the conditions are evidence against the conclusion.

We denote the fact that the certainty factor on A is 0.2 by CF(A)=0.2 or A(CF=0.2) and the fact that the certainty factor of the rule IF A and B THEN C is 0.3 by IF A and B THEN C(with CF=0.3)

Theorem of certainty factor

The CF of conjunction of fact is the minimum of the CF among the CF of each of the facts. i.e. CF(A AND B AND C AND……)= min CF(A ,B , C……)

The CF of disjunction of fact is the maximum of the CF among the CF of each of the facts. i.e. CF(A OR B OR C OR……)= max CF(A , B , C……)

The CF(IF A then B)=CF(B)/CF(A). from this we can derive

CF(B)= CF(IF A then B)* CF(A)

Consider the set of rule with the same conclusion part say B. The CF of the conclusion part is the maximum of the CF of all those B. i.e. CF(B)= max CF(B's)

Example. Consider the following rules.

Rule 1: IF A(CF=0.4) AND B(CF=0.5) OR C(CF=0.6) THEN D (with CF=0.6)

Rule2: IF E (CF=0.7) THEN D (with CF=0.3)

CF(A AND B OR C)= max(CF(A AND B),CF(C)) =max(min(0.4, 0.5), 0.8)=0.8

Calculating CF(D) from rule 1 gives

CF(IF A AND B OR C THEN D)* CF(A AND B OR C) =0.8 *0.6=0.48

Calculating CF(D) from rule 2 gives CF(IF E THEN D)* CF(E) =0.7*0.3=0.21

Calculating the CF(D) using the two rules gives CF(D)= max(0.48, 0.21)=0.48

Pathfinder was one of the system developed using probability theory. It was developed to assist pathologist in the diagnosis of lymph- node related diseases. Given a number of findings, it would suggest possible diseases. Pathfinder explored a range of problem solving methods and techniques for handling uncertainty including simple Bayes, certainty factor and the scoring scheme used in internist. They were compared by developing system based on the different methods and determine which gave more accurate diagnose, Bayes did best.

Limitation of rule based system

Knowledge acquisition is the process of extracting knowledge from experts. Given the difficulty involved in having experts articulate their intuition in terms of a systematic process of reasoning; this aspect is regarded as the main bottleneck in expert systems development.

rule-based systems are really only feasible for problems for which any and all knowledge in the problem area can be written in the form of if-then rules

Rule based system is only applicable for problem in which the area is not large. If there are too many rules, the system can become difficult to maintain and can suffer a performance hit.

Rule-based systems are a relatively simple model that can be adapted to any number of problems. A rule-based system has its strengths as well as limitations that must be considered before deciding if it is the right technique to use for a given problem. Overall, rule-based systems are really only feasible for problems for which any and all knowledge in the problem area can be written in the form of if-then rules and for which this problem area is not large.

Case based system

In case-based reasoning (CBR) systems expertise is embodied in a library of past cases, rather than being encoded in classical rules. Each case typically contains a description of the problem, plus a solution and/or the outcome. The knowledge and reasoning process used by an expert to solve the problem is not recorded, but is implicit in the solution.

To solve a current problem: the problem is matched against the cases in the case base, and similar cases are retrieved. The retrieved cases are used to suggest a solution which is reused and tested for success. If necessary, the solution is then revised. Finally the current problem and the final solution are retained as part of a new case.

Case-based reasoning is liked by many people because they feel happier with examples rather than conclusions separated from their context. A case library can also be a

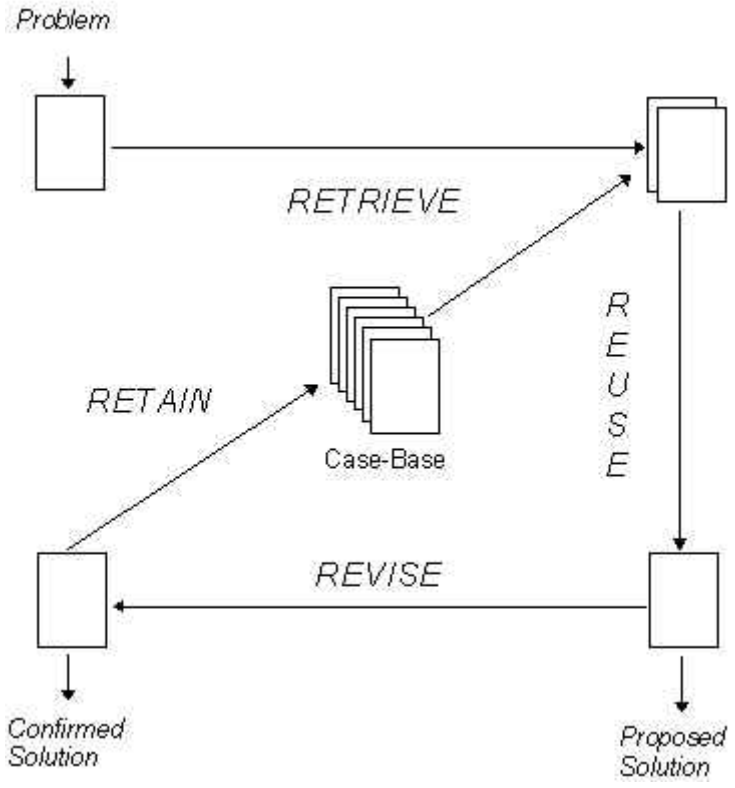
powerful corporate resource, allowing everyone in an organisation to tap into the corporate case library when handling a new problem.

Since the 1990's CBR has grown into a field of widespread interest, both from an academic and a commercial standpoint. Mature tools and application-focused conferences exist. Case-based reasoning is often used as a generic term to describe techniques including but not limited to case-based reasoning as we describe it here (e.g. analogical reasoning is often referred to as case-based reasoning).

Case based System cycle

All case-based reasoning methods have in common the following process:

- retrieve the most similar case (or cases) comparing the case to the library of past cases;
- reuse the retrieved case to try to solve the current problem;
- revise and adapt the proposed solution if necessary;
- retain the final solution as part of a new case.



There are a variety of different methods for organising, retrieving, utilising and indexing the knowledge retained in past cases.

Retrieving a case starts with a (possibly partial) problem description and ends when a best matching case has been found. The subtasks involve:

- identifying a set of relevant problem descriptors;
- matching the case and returning a set of sufficiently similar cases (given a similarity threshold of some kind); and
- selecting the best case from the set of cases returned.

Some systems retrieve cases based largely on superficial syntactic similarities among problem descriptors, while advanced systems use semantic similarities.

Reusing the retrieved case solution in the context of the new case focuses on: identifying the differences between the retrieved and the current case; and identifying the part of a retrieved case which can be transferred to the new case. Generally the solution of the retrieved case is transferred to the new case directly as its solution case.

Revising the case solution generated by the reuse process is necessary when the solution proves incorrect. This provides an opportunity to learn from failure.

Retaining the case is the process of incorporating whatever is useful from the new case into the case library. This involves deciding what information to retain and in what form to retain it; how to index the case for future retrieval; and integrating the new case into the case library.

A CBR tool should support the four main processes of CBR: retrieval, reuse, revision and retention. A good tool should support a variety of retrieval mechanisms and allow them to be mixed when necessary. In addition, the tool should be able to handle large case libraries with retrieval time increasing linearly (at worst) with the number of cases.

Applications

Case based reasoning first appeared in commercial tools in the early 1990's and since then has been sued to create numerous applications in a wide range of domains:

- Diagnosis: case-based diagnosis systems try to retrieve past cases whose symptom lists are similar in nature to that of the new case and suggest diagnoses based on the best matching retrieved cases. The majority of installed systems are of this type and there are many medical CBR diagnostic systems.
- Help Desk: case-based diagnostic systems are used in the customer service area dealing with handling problems with a product or service.
- Assessment: case-based systems are used to determine values for variables by comparing it to the known value of something similar. Assessment tasks are quite common in the finance and marketing domains.
- Decision support: in decision making, when faced with a complex problem, people often look for analogous problems for possible solutions. CBR systems have been developed to support in this problem retrieval process (often at the level of document retrieval) to find relevant similar problems. CBR is particularly good at querying structured, modular and non-homogeneous documents.

- Design: Systems to support human designers in architectural and industrial design have been developed. These systems assist the user in only one part of the design process, that of retrieving past cases, and would need to be combined with other forms of reasoning to support the full design process.

Suitability

Some of the characteristics of a domain that indicate that a CBR approach might be suitable include:

- records of previously solved problems exist;
- historical cases are viewed as an asset which ought to be preserved;
- remembering previous experiences is useful;
- specialists talk about their domain by giving examples;
- experience is at least as valuable as textbook knowledge.

Case-based reasoning is often used where experts find it hard to articulate their thought processes when solving problems. This is because knowledge acquisition for a classical KBS would be extremely difficult in such domains, and is likely to produce incomplete or inaccurate results. When using case-based reasoning, the need for knowledge acquisition can be limited to establishing how to characterise cases.

Case-based reasoning allows the case-base to be developed incrementally, while maintenance of the case library is relatively easy and can be carried out by domain experts.

6 Natural Language Understanding

A language is a system of signs having meaning by convention. Traffic signs, for example, form a mini-language, it being a matter of convention that, for example, the hazard-ahead sign means hazard ahead. This meaning-by-convention that is distinctive of language is very different from what is called natural meaning, exemplified in statements like 'Those clouds mean rain' and 'The fall in pressure means the valve is malfunctioning'.

An important characteristic of full-fledged human languages, such as English, which distinguishes them from, e.g. bird calls and systems of traffic signs, is their *productivity*. A productive language is one that is rich enough to enable an unlimited number of different sentences to be formulated within it.

It is relatively easy to write computer programs that are able, in severely restricted contexts, to respond in English, seemingly fluently, to questions and statements. An appropriately programmed computer can use language without understanding it, in principle even to the point where the computer's linguistic behaviour is indistinguishable from that of a native human speaker of the language.

What, then, is involved in genuine understanding, if a computer that uses language indistinguishably from a native human speaker does not necessarily understand? There is no universally agreed answer to this difficult question. According to one theory, whether or not one understands depends not only upon one's behaviour but also upon one's history: in order to be said to understand one must have learned the language and have been trained to take one's place in the linguistic community by means of interaction with other language-users.

The questions which need to be answered when we consider investigating language understanding by computer software are:

- what domains of discourse are rich enough to be a vehicle for studying the central issues yet simple enough to enable progress;
- what data representations are required when dealing with English sentences;
- what can be done and what can be done to enable the computations to take place;
- what is meant by understanding is an intelligent response adequate;
- what can be done to overcome problems when software limitations are exposed.

From sentences to models

Powerful language systems may require some of the following ideas:

- facts about word arrangement are explicit in parse trees;
- facts about the way acts relate to objects are explicit in thematic role frames
- facts about meaning are explicit in world models

Parse trees

Syntactic specialists bounces about in a sentence with only the modest goal of segmenting it into meaningful phrases and sentence constraints arrayed in a parse tree. Consider the sentence:

The clever robot moved the red engine to the appropriate chassis.

The parse tree for such a sentence records that the sentence is composed of a noun phrase and a verb phrase with an embedded noun phrase and an embedded prepositional phrase with an embedded noun phrase.

Parsing sentences

To embody syntactic constraints, we need some device that slows how phrases relate to one another and to words. One such device is the *context-free grammar*. Others are the transition-net grammar and the augmented transition-net grammar. Still another is the wait-and-see grammar. We will examine each, briefly. First, however, we need a glossary. Linguists rarely write out the full names for sentence constituents. Instead, they use mostly abbreviations:

Full Name Abbreviation

Sentence S

Noun phrase NP

Determiner DET

Adjective ADJ

Adjectives ADJS

Noun NOUN

Verb phrase VP

Verb VERB

Preposition PREP

Prepositional phrase PP

Prepositional phrases PPS

Context-free Grammars Capture Simple Constraints

The first rule means that a sentence is a noun phrase followed by something denoted by the funny-looking VP-PPS symbol. The purpose of the VP-PPS symbol is revealed by the fifth and sixth rules, which show that VP-PPS is a compound symbol that can spin off any number of prepositional phrases, including none, before disappearing into a verb phrase.

The second rule says that a noun phrase is a determiner followed by whatever is denoted by ADJS-NOUN. The third and fourth rules deal with the ADJS-NOUN symbol, showing that it can spin off any number of adjectives, including none, before becoming a noun. And finally, the seventh rule says that a verb phrase is a verb followed by a noun phrase, and the eighth rule says that a prepositional phrase is a preposition followed by a noun phrase. The first eight rules involve only nonterminal symbols that do not appear in completed sentences, the remaining rules determine how some of these uppercase symbols are associated with lower case symbols that relate to words.

Context-free grammars consists of *context-free rules* like the following:

1 S -> NP VP-PPS

2 NP -> DET ADJS-NOUN

3 ADJS-NOUN -> ADJ ADJS-NOUN

4 ADJS-NOUN -> NOUN

5 VP-PPS -> VP-PPS PP

6 VP-PPS -> VP

7 VP -> VERB NP

8 PP -> PREP NP

9 DET -> a [[??]] the [[??]] this [[??]] that

10 ADJ -> silly [[??]] red [[??]] big

11 NOUN -> robot [[??]] pyramid [[??]] top [[??]] brick

12 VERB -> moved

13 PREP -> to [[??]] of

Because of the arrows it is normal to think of using the rules generatively, starting with sentence S via NP VP-PPS until a string of terminal symbols is reached.

Scan the string from the left to the right until a nonterminal is reached replace it using a rule and repeat until no nonterminals are left. Such grammars are known as context free because the left hand side consists only of the symbol to be replaced. All terminal-only strings produced by the grammar are well-formed sentences. Using top-down moving from the rules to the words

S

NP VP-PPS

DET ADJS-NOUN VP-PPS

The ADJS-NOUN VP-PPS

The ADJ ADJS-NOUN VP-PPS

The clever ADJS-NOUN VP-PPS

The clever NOUN VP-PPS

The clever robot VP-PPS

The clever robot VP-PPS PP

The clever robot VERB NP PP

The clever robot moved NP PP

.

The clever robot moved the red engine to the appropriate chassis.

Whilst it appears natural to move in this way our purpose now is to progress from the sentence to the parse tree using appropriate rules or bottom up. One way of doing this is to use the same rules backwards. Instead of starting with S we start with the words and end up with S hopefully with no spare words over.

The clever robot moved the red engine to the appropriate chassis.

DET clever robot moved the red engine to the appropriate chassis .

DET ADJ robot moved the red engine to the appropriate chassis.

DET ADJ NOUN moved the red engine to the appropriate chassis.

DET ADJS-NOUN moved the red engine to the appropriate chassis.

NP moved the red engine to the appropriate chassis.

NP VERB the red engine to the appropriate chassis.

NP VERB DET red engine to the appropriate chassis.

NP VERB DET ADJ engine to the appropriate chassis.

NP VERB DET ADJ NOUN to the appropriate chassis.

NP VERB DET ADJS-NOUN to the appropriate chassis.

NP VERB NP to the appropriate chassis.

NP VP to the appropriate chassis.

NP VP-PPS to the appropriate chassis.

NP VP-PPS PREP the appropriate chassis.

NP VP-PPS PREP DET appropriate chassis.

NP VP-PPS PREP DET AFJ chassis.

NP VP-PPS PREP DET ADJ NOUN.

NP VP-PPS PREP NP.

NP VP-PPS PP.

NP VP-PPS.

S.

TRANSITION NETS CAPTURE SIMPLE SYNTACTIC CONSTRAINTS

All parser interpreters must involve mechanisms for building phrase describing nodes and for connecting these nodes together. All parsers must consider the following questions:

- when should a parser **start** work on a new node, creating it;
- when should a parser **stop** work on an existing node, completing it.

- where should a parser **attach** a completed node to the rest of the already built tree structure.

In a traditional parser built using context free grammar rules a simple procedure specifies how to start and to stop nodes and the grammar rules specify where each node is attached.

An alternative method which is equivalent is called the transition net. They are made up of nodes and directed links called arcs. The transition net interpreter gives straightforward answers to the questions stop start and attach. Work on an existing node stops whenever a net is traversed or a failure occurs and a node is attached whenever any net is traversed other than the top level.

To move through the sentence net we must first traverse the noun phrase net the first word must be a determiner. This procedure consists of top down parsing. It is called top down because everything starts with the creation of a sentence node at the top of the parse tree and moves down toward an eventual examination of the words in a sentence.

Consider the sentence:

The clever robot moved the red engine to the appropriate chassis.

Moving through the sentence net a sentence node is created. Next we encounter a noun phrase arc labelled T1. This creates a noun phrase node and initiates an attempt to traverse the noun phrase net. This in turn initiates an attempt on the determiner arc T3, in the noun phrase net . The first word is a determiner the consequently a determiner node is created and attached to the noon phrase. The word the is also attached to the determiner node. Now we need to take a choice either the adjective arc T4 or the noun arc T5. There is an adjective clever so we take the adjective path. The path T5 is taken for the noun robot. We are now in the double circle success node and this takes us back to the sentence node and we move one st age further on. The next thing to look for is a verb phrase T2. Moving quickly through the arcs T3 T4 T5 with the phrase *The appropriate chassis*, we return to the verb phrase net. We now have the option of a prepositional phrase transition net. This is T10 in the verb phrase transition net. We now move to the prepositional phrase transition net. The first arc is a preposition and the first word encountered is to a preposition, eventually the phrase to the appropriate chassis is claimes as a prepositional phrase and we return to the sentence node and success at S3. As there are no more words in the sentence a successful parse has occurred.

Summary of rules

To parse a sentence using transition nets

1 create a parse tree node named S

2 determine if it is possible to traverse a path of arcs from the initial node to a success node denoted by a dotted circle. If so and if all the sentences words are consumed in the process announce success otherwise failure.

To parse phrases using transition nets

1 create a parse tree node with the same name as that of that of the transition net.

2 determine if it is possible to traverse a path of arcs from the initial node to a success node denoted by a dotted circle. If so and if all the sentences words are consumed in the process announce success otherwise failure.

To traverse an arc

1a if the arc has a lower case symbol on it the next word in the sentence must have that symbol as a feature otherwise fail the word is consumed as the arc is traversed.

1b If the arc has a downward arrow, [[arrowdown]], go off and try to traverse the subnet named just after the downward pointing arrow. If the subnet is successfully traversed attach the subnet's node to the current node otherwise fail.

7. Introduction to computer pattern recognition

Pattern recognition is the act of taking in raw data and taking an action based on the category of the data.
Pattern recognition aims to classify data (patterns) based either on a priori knowledge or on statistical information extracted from the patterns. The patterns to be classified are usually groups of measurements or observations, defining points in an appropriate multidimensional space. This is in contrast to pattern matching, where the pattern is rigidly specified.

A complete pattern recognition system consists of a sensor that gathers the observations to be classified or described, a feature extraction mechanism that computes numeric or symbolic information from the observations, and a classification or description scheme that does the actual job of classifying or describing observations, relying on the extracted features.
The classification or description scheme is usually based on the availability of a set of patterns that have already been classified or described. This set of patterns is termed the training set, and the resulting learning strategy is characterized as supervised learning. Learning can also be unsupervised, in the sense that the system is not given an a priori labeling of patterns, instead it itself establishes the classes based on the statistical regularities of the patterns.
The classification or description scheme usually uses one of the following approaches: statistical (or decision theoretic) or syntactic (or structural).
Statistical pattern recognition is based on statistical characterizations of patterns, assuming that the patterns are generated by a probabilistic system.
Syntactical (or structural) pattern recognition is based on the structural interrelationships of features. A wide range of algorithms can be applied for pattern recognition, from very simple Bayesian classifiers to much more powerful neural networks.
An intriguing problem in pattern recognition is the relationship between the problem to be solved (data to be classified) and the performance of various pattern recognition algorithms (classifiers).
Typical applications are automatic speech recognition, classification of text into several categories (e.g. spam/non-spam email messages), the automatic recognition of handwritten postal codes on postal envelopes, or the automatic recognition of images of human faces. The last two examples form the subtopic image analysis of pattern recognition that deals with digital images as input to pattern recognition systems. Within medical science, pattern recognition is the basis for computer-aided diagnosis (CAD) systems. CAD describes a procedure that supports the doctor's interpretations and findings.

Image analysis

Image analysis is the extraction of meaningful information from images; mainly from digital images by means of digital image processing techniques. Image analysis tasks can be as simple as reading bar coded tags or as sophisticated as identifying a person from their face.

Computers are indispensable for the analysis of large amounts of data, for tasks that require complex computation, or for the extraction of quantitative information. On the other hand, the human visual cortex is an excellent image analysis apparatus, especially for extracting higher-level information, and for many applications including medicine, security, and remote sensing human analysts still cannot be replaced by computers. For this reason, many important image analysis tools such as edge detectors and neural networks are inspired by human visual perception models.

Computer image analysis largely contains the fields of computer or machine vision, and medical imaging, and makes heavy use of pattern recognition, digital geometry, and signal processing..

It is the quantitative or qualitative characterization of two-dimensional (2D) or three-dimensional (3D) digital images. 2D images are, for example, to be analyzed in computer vision, and 3D images in medical imaging.

The applications of digital image analysis are continuously expanding through all areas of science and industry, including:

- medicine
- microscopy
- remote sensing
- astronomy
- defense
- materials science
- manufacturing
- security
- robotics
- document processing
- assay plate reading
- metallography

8 Some learning algorithm

Genetic algorithm

A genetic algorithm (GA) is a search technique used in computing to find exact or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. Genetic algorithms are a particular class of evolutionary algorithms (also known as evolutionary computation) that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover

Genetic algorithms are implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype of the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

Genetic algorithms find application in bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics and other fields.

A typical genetic algorithm requires two things to be defined:

1. a genetic representation of the solution domain,
2. a fitness function to evaluate the solution domain.

A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, that facilitates simple crossover operation. Variable length representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in Genetic programming and graph-form representations are explored in Evolutionary programming.

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the knapsack problem we want to maximize the total value of objects that we can put in a knapsack of some fixed capacity. A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the object is in the knapsack. Not every such representation is valid, as the size of objects may exceed the capacity of the knapsack. The fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid, or 0 otherwise. In some problems, it is hard or even impossible to define the fitness expression; in these cases, interactive genetic algorithms are used.

Once we have the genetic representation and the fitness function defined, GA proceeds to initialize a population of solutions randomly, then improve it through repetitive application of mutation, crossover, inversion and selection operators.

Initialization

Initially many individual solutions are randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, covering the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

Selection

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as this process may be very time-consuming.

Most functions are stochastic and designed so that a small proportion of less fit solutions are selected. This helps keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include roulette wheel selection and tournament selection.

Reproduction

The next step is to generate a second generation population of solutions from those selected through genetic operators: crossover (also called recombination), and/or mutation.

For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously. By producing a "child" solution using the above methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents". New parents are selected for each child, and the process continues until a new population of solutions of appropriate size is generated.

These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small proportion of less fit solutions, for reasons already mentioned above.

Termination

This generational process is repeated until a termination condition has been reached. Common terminating conditions are

- A solution is found that satisfies minimum criteria
- Fixed number of generations reached
- Allocated budget (computation time/money) reached
- The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results
- Manual inspection
- Combinations of the above.

Pseudo-code algorithm

1. Choose initial population
2. Evaluate the fitness of each individual in the population
3. Repeat until termination:
1. Select best-ranking individuals to reproduce
2. Breed new generation through crossover and/or mutation (genetic operations) and give birth to offspring
3. Evaluate the individual fitnesses of the offspring
4. Replace worst ranked part of population with offspring

Artificial Neural Network

An artificial neural network is a system based on the operation of biological neural networks, in other words, is an emulation of biological neural system. Why would be necessary the implementation of artificial neural networks? Although computing these days is truly advanced, there are certain tasks that a program made for a common microprocessor is unable to perform; even so a software implementation of a neural network can be made with their advantages and disadvantages.
Advantages:

- A neural network can perform tasks that a linear program can not.
- When an element of the neural network fails, it can continue without any problem by their parallel nature.
- A neural network learns and does not need to be reprogrammed.
- It can be implemented in any application.
- It can be implemented without any problem.

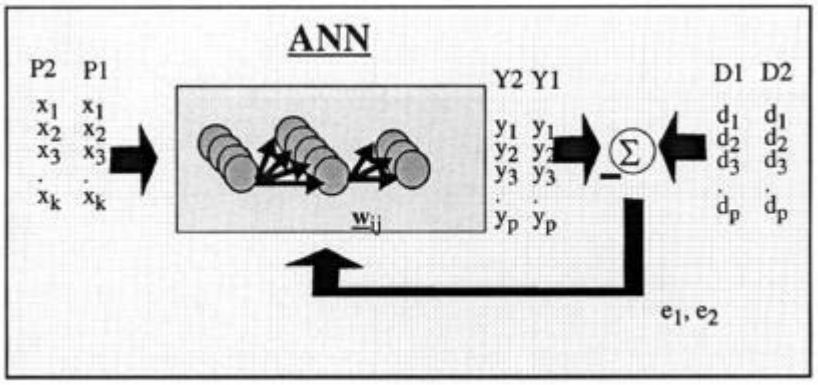
Disadvantages:

- The neural network needs training to operate.
- The architecture of a neural network is different from the architecture of microprocessors therefore needs to be emulated.
- Requires high processing time for large neural networks.

Another aspect of the artificial neural networks is that there are different architectures, which consequently requires different types of algorithms, but despite to be an apparently complex system, a neural network is relatively simple.

Artificial neural networks (ANN) are among the newest signal-processing technologies in the engineer's toolbox. The field is highly interdisciplinary, but our approach will restrict the view to the engineering perspective. In engineering, neural networks serve two important functions: as pattern classifiers and as nonlinear adaptive filters. We will provide a brief overview of the theory, learning rules, and applications of the most important neural network models. Definitions and Style of Computation An Artificial Neural Network is an adaptive, most often nonlinear system that learns to perform a

function (an input/output map) from data. Adaptive means that the system parameters are changed during operation, normally called the training phase . After the training phase the Artificial Neural Network parameters are fixed and the system is deployed to solve the problem at hand (the testing phase). The Artificial Neural Network is built with a systematic step-by-step procedure to optimize a performance criterion or to follow some implicit internal constraint, which is commonly referred to as the learning rule . The input/output training data are fundamental in neural network technology, because they convey the necessary information to "discover" the optimal operating point. The nonlinear nature of the neural network processing elements (PEs) provides the system with lots of flexibility to achieve practically any desired input/output map, i.e., some Artificial Neural Networks are universal mappers . There is a style in neural computation that is worth describing.



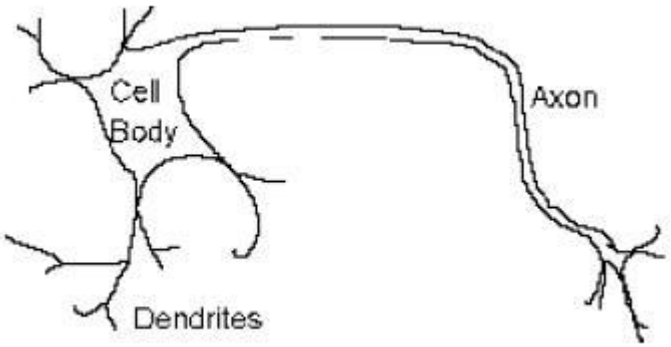
The style of neural computation.

An input is presented to the neural network and a corresponding desired or target response set at the output (when this is the case the training is called supervised). An error is composed from the difference between the desired response and the system output. This error information is fed back to the system and adjusts the system parameters in a systematic fashion (the learning rule). The process is repeated until the performance is acceptable. It is clear from this description that the performance hinges heavily on the data. If one does not have data that cover a significant portion of the operating conditions or if they are noisy, then **neural network** technology is probably not the right solution. On the other hand, if there is plenty of data and the problem is poorly understood to derive an approximate model, then neural network technology is a good choice. This operating procedure should be contrasted with the traditional engineering design, made of exhaustive subsystem specifications and intercommunication protocols. In artificial neural networks, the designer chooses the network topology, the performance function, the learning rule, and the criterion to stop the training phase, but the system automatically adjusts the parameters. So, it is difficult to bring a priori information into the design, and when the system does not work properly it is also hard to incrementally refine the solution. But ANN-based solutions are extremely efficient in terms of development time and resources, and in many difficult problems artificial neural networks provide performance that is difficult to match with other technologies. Denker 10 years ago said

that "artificial neural networks are the second best way to implement a solution" motivated by the simplicity of their design and because of their universality, only shadowed by the traditional design obtained by studying the physics of the problem. At present, artificial neural networks are emerging as the technology of choice for many applications, such as pattern recognition, prediction, system identification, and control.

The Biological Model

Artificial neural networks emerged after the introduction of simplified neurons by McCulloch and Pitts in 1943 (McCulloch & Pitts, 1943). These neurons were presented as models of biological neurons and as conceptual components for circuits that could perform computational tasks. The basic model of the neuron is founded upon the functionality of a biological neuron. "Neurons are the basic signaling units of the nervous system" and "each neuron is a discrete cell whose several processes arise from its cell body".



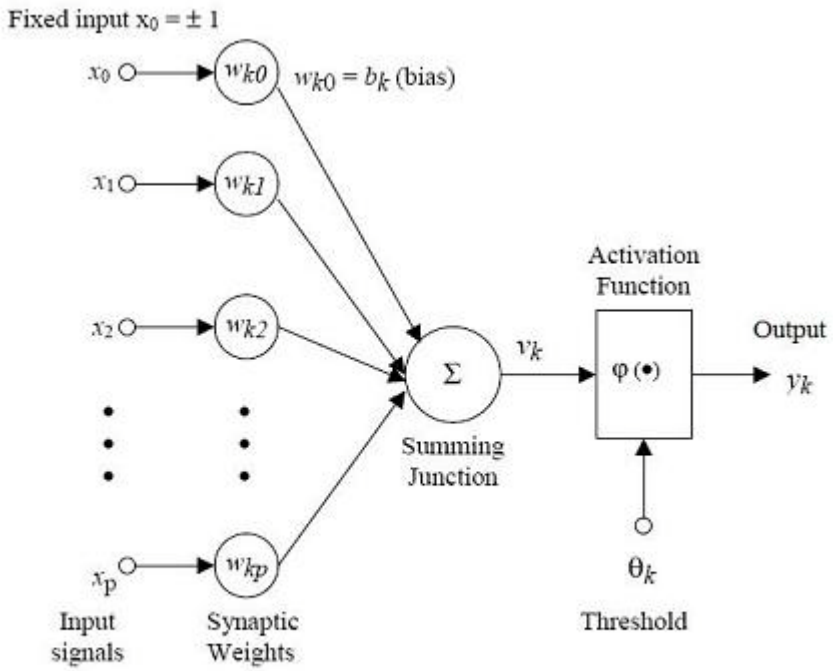
The neuron has four main regions to its structure. The cell body, or soma, has two offshoots from it, the dendrites, and the axon, which end in presynaptic terminals. The cell body is the heart of the cell, containing the nucleus and maintaining protein synthesis. A neuron may have many dendrites, which branch out in a treelike structure, and receive signals from other neurons. A neuron usually only has one axon which grows out from a part of the cell body called the axon hillock. The axon conducts electric signals generated at the axon hillock down its length. These electric signals are called action potentials. The other end of the axon may split into several branches, which end in a presynaptic terminal. Action potentials are the electric signals that neurons use to convey information to the brain. All these signals are identical. Therefore, the brain determines what type of information is being received based on the path that the signal took. The brain analyzes the patterns of signals being sent and from that information it can interpret the type of information being received. Myelin is the fatty tissue that surrounds and insulates the axon. Often short axons do not need this insulation. There are uninsulated parts of the axon. These areas are called Nodes of Ranvier. At these nodes, the signal travelling down the axon is regenerated. This ensures that the signal travelling down the axon travels fast and remains constant (i.e. very short propagation delay and no weakening of the signal). The synapse is the area of contact between two neurons. The

neurons do not actually physically touch. They are separated by the synaptic cleft, and electric signals are sent through chemical 13 interaction. The neuron sending the signal is called the presynaptic cell and the neuron receiving the signal is called the postsynaptic cell. The signals are generated by the membrane potential, which is based on the differences in concentration of sodium and potassium ions inside and outside the cell membrane. Neurons can be classified by their number of processes (or appendages), or by their function. If they are classified by the number of processes, they fall into three categories. Unipolar neurons have a single process (dendrites and axon are located on the same stem), and are most common in invertebrates. In bipolar neurons, the dendrite and axon are the neuron's two separate processes. Bipolar neurons have a subclass called pseudo-bipolar neurons, which are used to send sensory information to the spinal cord. Finally, multipolar neurons are most common in mammals. Examples of these neurons are spinal motor neurons, pyramidal cells and Purkinje cells (in the cerebellum). If classified by function, neurons again fall into three separate categories. The first group is sensory, or afferent, neurons, which provide information for perception and motor coordination. The second group provides information (or instructions) to muscles and glands and is therefore called motor neurons. The last group, interneuronal, contains all other neurons and has two subclasses. One group called relay or projection interneurons have long axons and connect different parts of the brain. The other group called local interneurons are only used in local circuits.

The Mathematical Model

When creating a functional model of the biological neuron, there are three basic components of importance. First, the synapses of the neuron are modeled as weights. The strength of the connection between an input and a neuron is noted by the value of the weight. Negative weight values reflect inhibitory connections, while positive values designate excitatory connections [Haykin]. The next two components model the actual activity within the neuron cell. An adder sums up all the inputs modified by their respective weights. This activity is referred to as linear combination. Finally, an activation function controls the amplitude of the output of the neuron. An acceptable range of output is usually between 0 and 1, or -1 and 1.

Mathematically, this process is described in the figure



From this model the interval activity of the neuron can be shown to be:

$$v_k = \sum_{j=1}^{p} w_{kj} x_j$$

The output of the neuron, yk, would therefore be the outcome of some activation function on the value of vk.

Activation functions

As mentioned previously, the activation function acts as a squashing function, such that the output of a neuron in a neural network is between certain values (usually 0 and 1, or -1 and 1). In general, there are three types of activation functions, denoted by Φ(.) . First, there is the Threshold Function which takes on a value of 0 if the summed input is less than a certain threshold value (v), and the value 1 if the summed input is greater than or equal to the threshold value.

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$

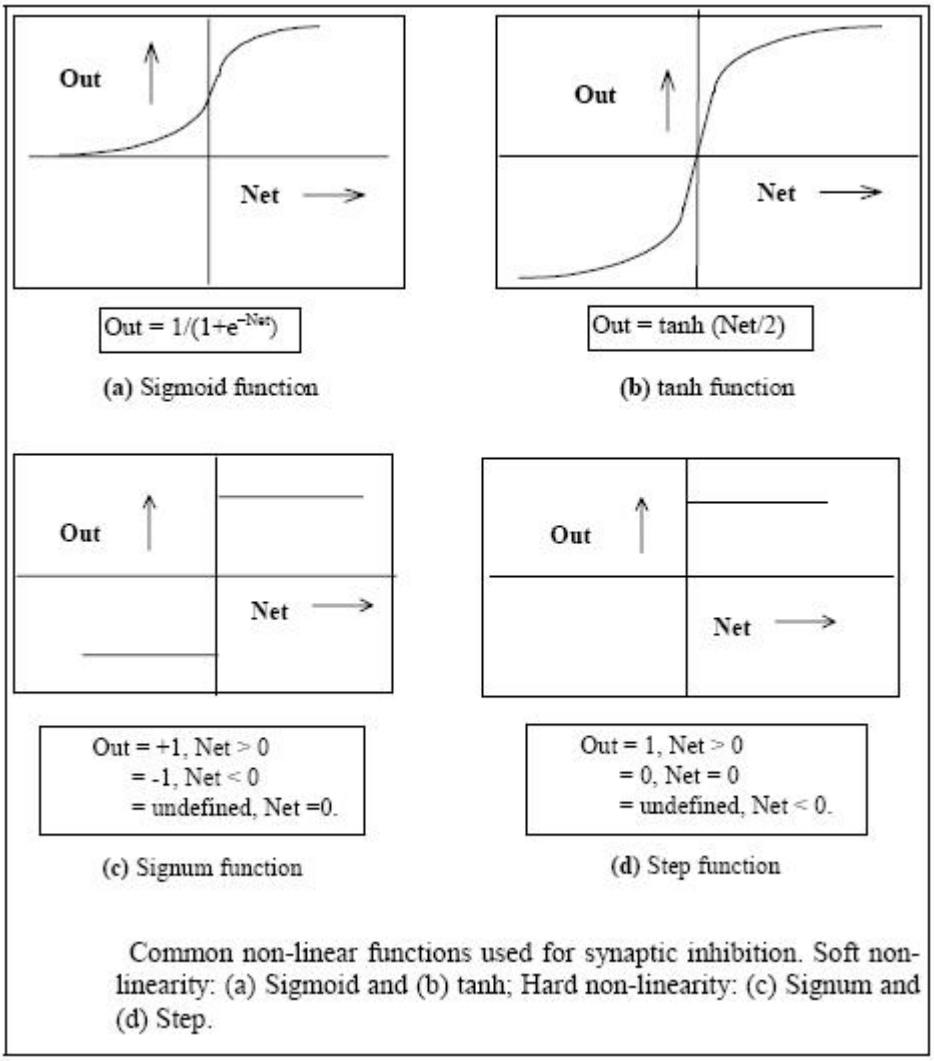
Secondly, there is the Piecewise-Linear function. This function again can take on the values of 0 or 1, but can also take on values between that depending on the amplification

factor in a certain region of linear operation.

$$\varphi(v)=\begin{cases}1 & v\geq\frac{1}{2}\\ v & -\frac{1}{2}>v>\frac{1}{2}\\ 0 & v\leq-\frac{1}{2}\end{cases}$$

Thirdly, there is the sigmoid function. This function can range between 0 and 1, but it is also sometimes useful to use the -1 to 1 range. An example of the sigmoid function is the hyperbolic tangent function.

$$\varphi(v)=\tanh\left(\frac{v}{2}\right)=\frac{1-\exp(-v)}{1+\exp(-v)}$$



Common non-linear functions used for synaptic inhibition. Soft non-linearity: (a) Sigmoid and (b) tanh; Hard non-linearity: (c) Signum and (d) Step.

The artifcial neural networks which we describe are all variations on the parallel distributed processing (PDP) idea. The architecture of each neural network is based on very similar building blocks which perform the processing. In this chapter we first discuss these processing units and discuss diferent neural network topologies. Learning strategies as a basis for an adaptive system

A framework for distributed representation

An artifcial neural network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections. A set of major aspects of a parallel distributed model can be distinguished :

- a set of processing units ('neurons,' 'cells');
- a state of activation yk for every unit, which equivalent to the output of the unit;
- connections between the units. Generally each connection is defined by a weight wjk which determines the effect which the signal of unit j has on unit k;
- a propagation rule, which determines the effective input sk of a unit from its external inputs;
- an activation function Fk, which determines the new level of activation based on the efective input sk(t) and the current activation yk(t) (i.e., the update);
- an external input (aka bias, offset) øk for each unit;
- a method for information gathering (the learning rule);
- an environment within which the system must operate, providing input signals and|if necessary|error signals.

Processing units

Each unit performs a relatively simple job: receive input from neighbours or external sources and use this to compute an output signal which is propagated to other units. Apart from this processing, a second task is the adjustment of the weights. The system is inherently parallel in the sense that many units can carry out their computations at the same time. Within neural systems it is useful to distinguish three types of units: input units (indicated by an index i) which receive data from outside the neural network, output units (indicated by an index o) which send data out of the neural network, and hidden units (indicated by an index h) whose input and output signals remain within the neural network. During operation, units can be updated either synchronously or asynchronously. With synchronous updating, all units update their activation simultaneously; with asynchronous updating, each unit has a (usually fixed) probability of updating its activation at a time t, and usually only one unit will be able to do this at a time. In some cases the latter model has some advantages.

Neural Network topologies

In the previous section we discussed the properties of the basic processing unit in an artificial neural network. This section focuses on the pattern of connections between the units and the propagation of data. As for this pattern of connections, the main distinction we can make is between:

- **Feed-forward neural networks**, where the data ow from input to output units is strictly feedforward. The data processing can extend over multiple (layers of) units, but no feedback connections are present, that is, connections extending from outputs of units to inputs of units in the same layer or previous layers.

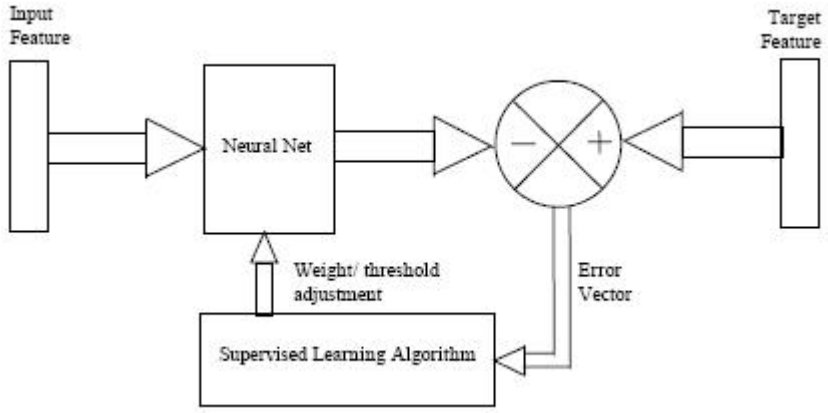
- **Recurrent neural networks** that do contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the neural network will evolve to a stable state in which these activations do not change anymore. In other applications, the change of the activation values of the output neurons are significant, such that the dynamical behaviour constitutes the output of the neural network.

Training of artificial neural networks

A **neural network** has to be configured such that the application of a set of inputs produces (either 'direct' or via a relaxation process) the desired set of outputs. Various methods to set the strengths of the connections exist. One way is to set the weights explicitly, using a priori knowledge. Another way is to **'train' the neural network** by feeding it teaching patterns and letting it change its weights according to some learning rule.

We can categorise the learning situations in two distinct sorts. These are:

- **Supervised learning** or Associative learning in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the neural network (self-supervised).



- **Unsupervised learning** or Self-organisation in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no a priori set of categories into which the patterns are to be classified; rather the system must develop its own representation of the input stimuli.

- **Reinforcement Learning** This type of learning may be considered as an intermediate form of the above two types of learning. Here the learning machine does some action on the environment and gets a feedback response from the environment. The learning system grades its action good (rewarding) or bad (punishable) based on the environmental response and accordingly adjusts its parameters. Generally, parameter adjustment is continued until an equilibrium state occurs, following which there will be no more changes in its parameters. The self organizing neural learning may be categorized under this type of learning.

References:

Alison Cawsey 1998 The Essence of Artificial Intelligence Prentice Hall ISBN:0135717795

Phil Mars, J.R. Chen and R.Nambiar. Learning Algorithms: Theory and applications in signal processing, control and communications. CRC press New York

Stuart Russell Peter Norvig Artificial Intelligence: A Modern Approach (2nd Edition)

W F B Jones, Artificial Intelligence I [http://www.cs.cf.ac.uk/Dave/AI1/AI1.html]