Unit 1. Introduction

Learning Outcomes. You should be able to

- define the term ''Artificial Intelligence''
- outline the major topics of AI
- describe several example real-world AI applications
- suggest AI techniques appropriate for implementing various features of your example applications
- list and describe 6 eras in the history of AI from 1943, naming at least one landmark in each
- describe the Turing test and discuss its weaknesses

Contents.

- AI objectives, definition, scope
- classification of AI techniques
- landmarks in AI history
- Turing test and Loebner Prize

The term Artificial Intelligence (AI) is reputed to have been coined by John McCarthy in 1956.

The long-term objective of AI is to build intelligent machines, that can reason, learn, see, communicate and operate in complex environments at the human level, or possibly, even better. The essence of AI is not to replace human intelligence but rather to complement it — intelligent humans and intelligent machines can realise opportunities together that neither can realise alone (e.g. in business computers can help us to locate pertinent information, to allocate resources, to schedule work and to discover meaningful regularities in databases).

There are two common approaches to AI. The first is to use computers to simulate the human mental processes. Psychologists, for example, are interested in AI because it provides new insights into *natural* intelligence and the workings of the human brain. As a common approach to constructing intelligent machines this has some inherent problems, e.g.:

1. Human intelligence includes mental processes that are difficult if not impossible to understand and describe.
2. There are considerable differences between the structure and capabilities of the human brain and the computer.

The second more common approach, adopted by computer scientists, is to attempt to create intelligent systems independent of the human condition — **this is the approach that will be described in this series of lectures**. We will demonstrate in this brief excursion into AI the power of the human brain — we will see that:

> Some tasks considered difficult for humans to accomplish are relatively easy to program for the computer, whilst other tasks that are taken for granted by humans are extremely difficult to implement computationally.

It is very difficult to give a concise description, or definition, of the field of Artificial Intelligence (AI) since workers in many areas claim to be involved in AI — computer scientists, psychologists, brain theorists, linguists etc. Instead of becoming involved in the merits of various definitions we will adopt a practical approach to answering the question "What is AI?", that is, we will look at those topics that are 'unquestionably' considered to be parts of AI.

AI Topics and Major Subdivisions

A commonly accepted structure of AI is that proposed by **Nilsson** (1974) where the area is divided into

– CORE TOPICS — concerned with basic AI techniques and methodologies
– APPLICATION AREAS — draw upon research in one of the core areas

Core topics

– Modelling and the Representation of Knowledge
– Heuristic Search
– AI Systems and Language

Application Areas

– Game Playing
– Maths, Science and Engineering Aids
– Automatic Theorem Proving
– Expert Systems
– Intelligent Teaching Systems
– Automatic programming
– Robots
– Machine Vision
– Natural Language Systems
– Information Processing Psychology

The following more **recent structure** for AI was proposed by **Gevarter** (1985):

Core Topics

– Modelling and the Representation of Knowledge

– Heuristic Search
– Common Sense, Reasoning and Logic
– AI Languages and Tools

Application Areas

– Expert Systems
– Vision
– Natural Language Processing
– Problem Solving and Planning

Update. To the above we might now add: Neural Networks, Genetic Algorithms and Intelligent Agents.

Philosophical Issues

AI research makes the fundamental assumption that human intelligence can be reduced to the (complex) manipulation of symbols conducted by some appropriate processor — not necessarily a biological brain. This assumption does not go unchallenged!!

Searle states that just **behaving** intelligently is not necessarily a sign of intelligence. In his "Chinese room" argument he states that given a sufficiently large rule book it would be possible for a person who could not speak Chinese to reply to Chinese sentences by simply looking up the appropriate responses in the rule book. Thus computers operating on this principle could not be said to be intelligent.

The following observation made by Marvin Minsky a research pioneer of AI is appropriate:

> *What is intelligence anyway? It is only a word that people use to name those unknown processes with which our brains solve problems we call hard. But whenever you learn a skill yourself, you're less impressed or mystified when other people do the same. This is why the meaning of "intelligence" seems so elusive: it describes not some definite thing but only the momentary horizon of our ignorance about how minds might work.*

Some AI landmarks

Early AI (1943 – 1956)

– McCulloch and Pitts — artificial neuron (1943)
– Chess playing programs — Alan Turing, Claude Shannon (early 50's)
– Dartmouth College workshop (1956) — Allen Newell and Herbert Simon developed the Logic Theorist (LT) — term AI proposed by John McCarthy

Great initial expectations (1952 – 1969)

– A Newell, H Simon — General Problem Solver (GPS)
– A Samuel — checker playing program (1952 onwards)
– J McCarthy — defined programming language LISP (1958)
– Shakey robot built at Stanford Research Institute (1966 onwards)
– Frank Rosenblatt — Perceptron (1962)

Reality (1966 – 1974)

– Machine translation, Natural Language Processing — difficulties
– Combinatorial explosion
– Perceptron deficiencies — M Minsky and S Papert (1969)

Knowledge-based systems (1969 – 1979)

– Recognition of strengths of strong methods over weak methods
– Expert systems — Dendral, Mycin
– Frames

AI — industrial importance (1980 – 1988)

– Commercialisation of expert systems — XCON (R1) in 1982
– Japanese Fifth Generation — 10 year plan (1981)

Recent history

– advances in Genetic Algorithms, Neural Networks
– new directions: Planning, Speech recognition, Intelligent Agents
– IBM Deep Blue beats Kasparov in chess (1997)

AI in Practice

– AI is concerned with automating both mundane tasks (e.g. planning, vision, natural language understanding) and tasks requiring specialised skills and training (medical diagnosis, financial planning, equipment repair, computer configuration).
– **Paradoxically, it turns out that the mundane tasks are far harder to compute — tasks which a young child can do without thinking are beyond or at the limits of current AI research.**
– Although AI is still in its infancy it has already achieved some significant accomplishments with more likely in the near and distant future.
– Thus AI systems currently serve a wide variety of practical purposes in diagnosing patient illnesses and suggesting treatments, controlling assembly robots in factories, generating investment strategies by attempting to predict stock market trends.
– In the future, although some of them exist now, we will have language translation systems, air traffic control systems, supervisory systems, automated personal assistants, intelligent transport systems, robots for hazardous work environments.

The Industrial Importance of AI

- became apparent to many of the world's leading countries during the late 1970s.
- The Japanese launched a very ambitious programme, called the **Fifth Generation**, which was officially announced in October 1981, in which AI played a central role — the programme involved a ten-year plan to develop intelligent super computers.
- Other leading countries of the world also announced plans for some form of AI programme:
  - the British with the Alvey project,
  - the Europeans with the ESPRIT programme,
  - the United States had no formal plan, but several organisations grouped together on ambitious AI projects e.g. Microelectronics and Computer Technology Corporation (MCC) at Austin, Texas.

Turing Test

Alan Turing, a British mathematician, who helped to design the world's first operational electronic digital computer during the 1940s, effectively launched the field of AI with a 1950 paper entitled: "Computing Machinery and Intelligence" in which he proposed a test — the Turing test — for determining whether a machine was intelligent.

The Turing test attempts to answer the question:
How will we know if we have constructed an intelligent machine?

- an interrogator communicates with a computer and another person — the interrogator being placed in a separate room from the human and computer
- by asking a series of questions the interrogator attempts to determine which is the person and which is the machine
- the goal of the machine is to fool the interrogator into believing that it is the person — if the machine can succeed at this task then we can conclude that the machine can think

The Turing test has been criticised because its outcome depends on the gullibility of the interrogator and because it only examines natural language processing ability.

Other criticisms relate to the fact that it is not sufficient to simply judge a machine on how it acts — we also need to know what internal "mental" states it has — that is the machine has to be aware of its own mental state and actions — Turing himself acknowledges this by stating that a machine equals a brain when it is capable of not only writing a sonnet or composing a concerto **but knows too that it has accomplished this**.

Loebner Prize Contest

– introduced in 1991 by Hugh Loebner as a kind of competitive Turing test
– in this contest judges can type in questions at a computer terminal which are replied to either by a human or by one of the computer contestants
– the author of the best computer program receives a prize of $2000 and if the computer program is judged to give better responses than one of the humans then it receives a prize of $100,000
– the $100,000 prize has not been claimed yet although one program fooled five judges out of ten into believing that it was a human

Unit 2. Search 1/3

Learning Outcomes. You should be able to

- re-formulate an appropriate problem as a searching problem (i.e. identify the problem states and operators)
- explain differences between informed and uninformed searching
- define breadth-first and depth-first search algorithms
- apply these algorithms to concrete problems
- explain the differences between the two algorithms
- decide which one of the two to use for given problems
- define depth-first search with iterative deepening

According to Newell and Simon, intelligent activity in either human or machine, is achieved through the use of:

1. *patterns of symbols* to represent significant aspects of a problem domain,
2. *operations* on those patterns to generate potential solutions to problems,
3. *search* to select a solution from among these possibilities.

Search clearly forms an important aspect of intelligent activity and is likely to be required in AI e.g. a robot vehicle searching for a given route to a given destination, a chess program determining its next move, theorem proving, solving problems etc.

Problem Solving

– Well-defined problems are typically defined in terms of states, and solutions correspond to goal states.
– Solving a problem then amounts to **searching** through the different states until one or more of the goal states are found.
– One of the ways of achieving this is through the **state-space search method** — this will now be considered.

Problem states and operators

– The notion of problem states and operators can be introduced by considering the 8-puzzle (9-puzzle in some texts)

2	4	6
*	7	8
3	1	5

$\longrightarrow$

1	2	3
8	*	4
7	6	5

– Each tile configuration is a possible state e.g. consider other games and puzzles.
– The initial configuration and the desired configuration are known as the **initial** state and **goal** state respectively — the set of all possible states reachable from the initial state by any sequence of actions is known as the **state-space** of the problem — in many problems the state space is very large so search methods must be very efficient.
– An **operator** transforms one state into another — there are 4 operators in the 8-puzzle:
  – move blank up
  – move blank down
  – move blank right
  – move blank left
– An operator may not be applicable to every state.
– The problem then is to change the initial state into the goal state by moving the tiles around according to the available operators.
– A **solution** to a problem is thus that sequence of operators which will transform the initial state into the goal state.
– One approach would be to move the tiles at random — the so-called **British Museum algorithm** — we shall consider more intelligent approaches to the problem.

Search methods

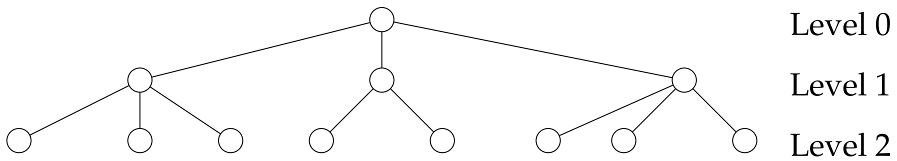
– Problem solving using either state-space or the problem reduction approach involves two major aspects — the representation of the states and the method of search.
– A state-space can be conveniently represented as a graph, where each node of the graph describes a state and arcs between nodes describe the transformation performed by applying an appropriate operator.
– Search methodologies can be classified into **uninformed** (or blind) search and **informed** (or directed) search.

Uninformed Search

– The search path selected is blindly or mechanically followed — no information is used to determine the preference of one child node over another.
– For all the uninformed methods considered in this Unit we shall adopt the convention that processed nodes are placed on a **closed** list and nodes waiting to be processed are placed on an **open** list.

Breadth-First Search

– Expands nodes in the order in which they are generated:



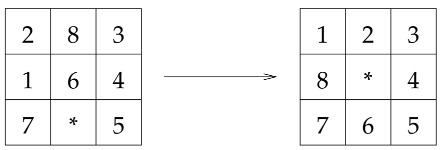
– The breadth-first search method would generate all nodes at level 1, then all nodes at level 2 and so on.

– By convention, the nodes on any particular level are generated in order from left to right.

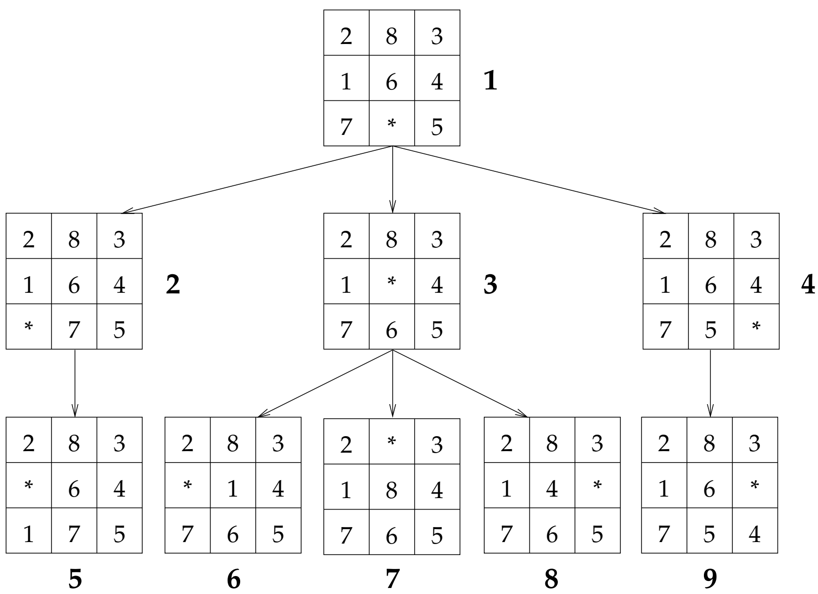
– The algorithm to achieve this, is as follows:

```
1. Place the start node s on the open list.
2. If the open list is empty, return failure and exit.
3. If the first element on the open list is a goal node g,
   return success and stop, otherwise.
4. Remove and expand the first element (n) from the open list
   and place all the successors (children) at the end of the
   queue - provide pointers back to node n which has been
   placed on the closed list.
5. Return to step 2.
```

– The procedure applied to the 8-puzzle where the problem is:



would give:



- The 8-puzzle search-space is in fact a graph (i.e. some nodes have multiple parents) but this can be ignored provided we never generate a particular node more than once.
- In the above search tree a solution will be found after 47 nodes have been generated and 26 nodes expanded — the solution is at level 5.
- The space utilisation of the breadth-first search is approximately B^n at level n, where B is the average branching factor at each level — at level n all these states will have been placed on the open list — thus space requirements can be prohibitive if solution paths are long.
- The time complexity of the algorithm is also B^n.

Depth-first Search

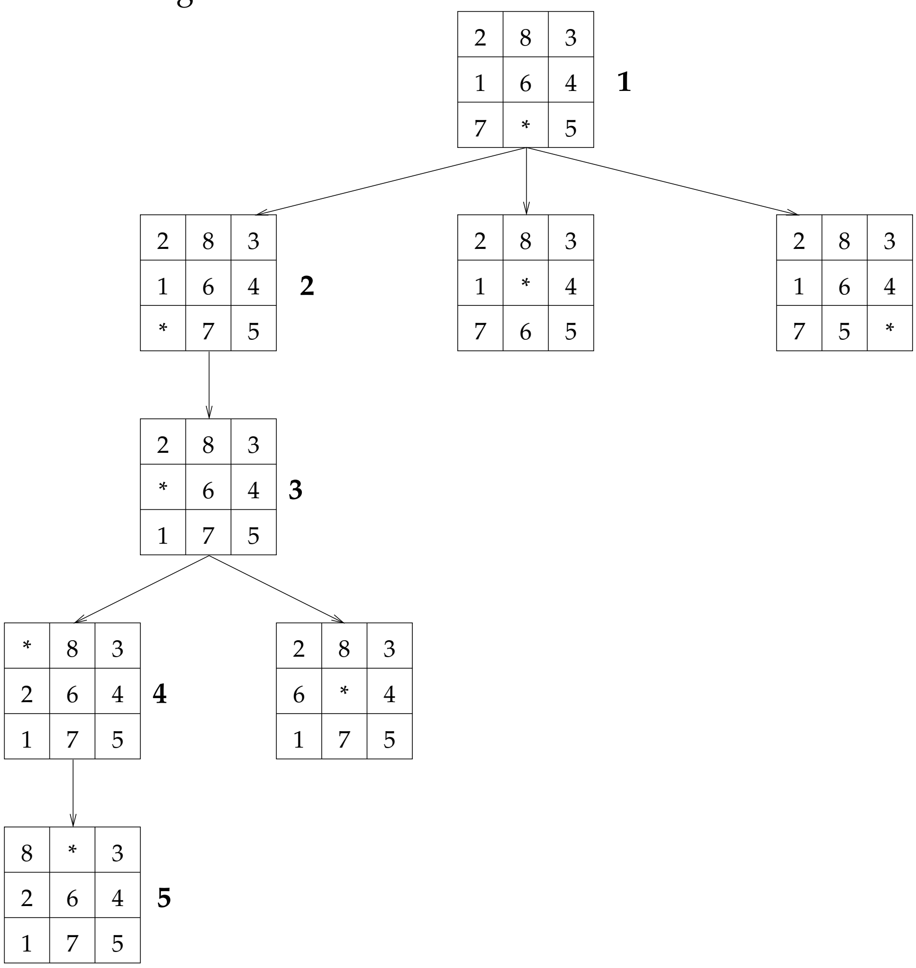
- Builds the search tree by expanding the most recently generated nodes first.
- the figure below shows part of the search tree generated by a depth-first search method for the 8-puzzle. Problem is as before to transform:

2	8	3
1	6	4
7	*	5

$\longrightarrow$

1	2	3
8	*	4
7	6	5

and the search tree generated is:



– The depth-first algorithm is as follows:

1. `Place the start node` s `of depth zero on the open list.`
2. `If the open list is empty, return failure and exit.`
3. `If the first element on the open list is a goal node` g`, return success and stop, otherwise.`
4. `Remove and expand the first element (`n`) from the open list and place all the successors (children) at the` **`front`** `of the queue - provide pointers back to node` n `which has been placed on the closed list.`
5. `Return to step 2.`

– The difficulty here is that we may expand along some search path which does not lead to the goal state — to prevent this we provide a 'backtracking' procedure which makes use of the idea of a **depth bound**.

– The **depth** of a node is defined as follows:

  – the depth of the root node is zero,
  – the depth of any other node is the depth of its parent plus one.

– The depth-first search is halted before it generates a node whose depth would exceed some depth bound and the search is then continued from the deepest node not exceeding this bound.

– In the example above, for a depth-bound of 6, 18 nodes have to be generated.

– The depth-first search is preferred over the breadth-first when the search tree is known to have a plentiful number of goals.

– The algorithm is less demanding in space requirements than the breadth-first method — the space complexity is just $B \times d$ (where d is the depth cut-off) — that is, it is a linear function of the length of the path, since we do not have to keep all the nodes at a given level on the open list — at each level open retains only the children of a single state.

Summary

– Depth-first search is preferred for searching a tree-structured search space with goal nodes at the leaves of the tree.

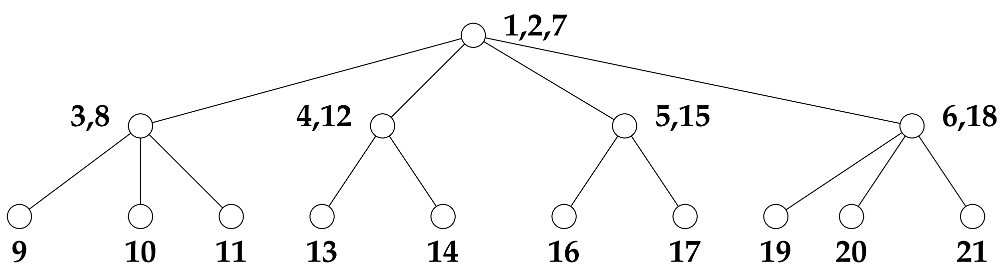
– Breadth-first search is often preferred for search trees with small branching factors, operators that are expensive to apply, and goal nodes expected at reasonable depth.

Depth-first with Iterative Deepening

- Performs a depth-first search of the space with a depth bound of one — if it fails to find a goal, it performs another depth-first search with a depth bound of two — this continues, increasing the depth-bound by one at each iteration.
- At each iteration, the algorithm performs a complete depth-first search to the current depth bound.
- No information about the space state is retained between iterations.
- Since the algorithm searches the space in a level-by-level fashion it is guaranteed to find the shortest path to a goal.
- This is the optimal algorithm in terms of space and time for uninformed search — this seems a paradox an intuitive explanation for which has been given by Korf [R.E Korf 'Search' in S.C. Shapiro Ed 1987 Encyclopaedia of AI, Wiley, New York]
- the algorithm for the depth-first with iterative deepening is as follows:

```
1. Initialise the current depth bound c to 1.
2. Place the start node, s, on the open list.
3. If the open list is empty, increment c and return to step
   2.
4. If the first element (n) on the open list is a goal node
   then return it and the path from the initial node to n ---
   return success and exit, otherwise,
5. Remove the first element (n) from the open list and put it
   on the closed list.  If the level of n is less than c then
   expand n and place all successors at the front of the
   queue --- provide pointers back to node n.
6. Return to step 3.
```

The diagram below shows a search tree and the order in which nodes of the tree are examined using the above algorithm. Nodes may be examined multiple times, once per iteration corresponding to different depth bound c.



Bi-directional Search

- Can be used when a problem has a single explicit goal state and all node generation operators have inverses, e.g. 8-puzzle.
- Performed by searching forward from the initial state and backward from the goal state simultaneously.
- To do this, the program must store the nodes generated on both search frontiers until a common node is found.
- All three of the blind search methods considered above may be used in the bi-directional search method.

Non-systematic search

- Search the problem space in a non-systematic manner — thus some portions of the search space could be searched several times whilst other portions will not be searched at all.
- Thus there is no guarantee that a goal node will be found, however studies have shown that on LARGE search problems the performance of non-systematic methods is often comparable to that achieved by systematic methods of the type described above.

Modifications required for searching graphs

- In a graph a node may have more than one parent, contrast this with a tree in which each node had at most one parent.
- The modifications required for searching graphs are:

  Breadth-first

  - If a newly generated node already exists on the list then discard it, i.e. do not place it on the list.

  Depth-first

  - this is more complicated since we need to redefine the depth of a node — it is now:
    - the depth of the root node is zero,
    - the depth of any other node is the depth of its shallowest parent plus one.
  - A depth-first search now selects for expansion the deepest node on the open list (subject to the depth bound) — when the node is expanded and the successors that are generated are on either the open or closed list then some re-computation of depth may be necessary.

Unit 3. Search 2/3

Learning Outcomes. You should be able to
- define the terms 'heuristic' and 'heuristic search'
- explain evaluation function and how it is used in a best-first search
- define and apply best-first search and the A* algorithms
- define admissibility; state when A* is admissible

Informed Search

– are search strategies which use some of the problem domain information in an attempt to increase the efficiency of the search process — they often depend on the use of **heuristic** information. (Heuristic means a rule of thumb, a method that provides guidance in decision making.)

– The search methods that use heuristic information are known as **heuristic search methods**.

– There are many ways in which heuristic information may be introduced e.g. one way to limit the search effort is to provide more effective and 'informed' operators that generate fewer successors — this technique can be effective for both depth-first and breadth-first based approaches.

– a second way in which heuristic information can be used is to order the nodes on the open list in such a way that the most 'promising' node is developed first, that is, the node judged to be the closest to the goal state is always expanded first — in general we term heuristics that measure the promise of a node as **evaluation functions**.

Note

– Heuristics may discard nodes on the solution path, this means that we are no longer guaranteed to find a solution even if one exists.

Evaluation Functions

– An evaluation function is a heuristic that assigns a numerical value to a node to indicate the 'promise' of that node.

– Given an evaluation function, it is then possible to rank the nodes such that the most promising one can be considered first.

– Evaluation functions may operate on one of a number of principles e.g., they may calculate the distance between the 'current' node and the goal node or they may evaluate the probability that the node is on the solution path.

– Construction of a good evaluation function is difficult.

– **We shall adopt the convention that the smaller the F value delivered by an evaluation function the more promising the node.**

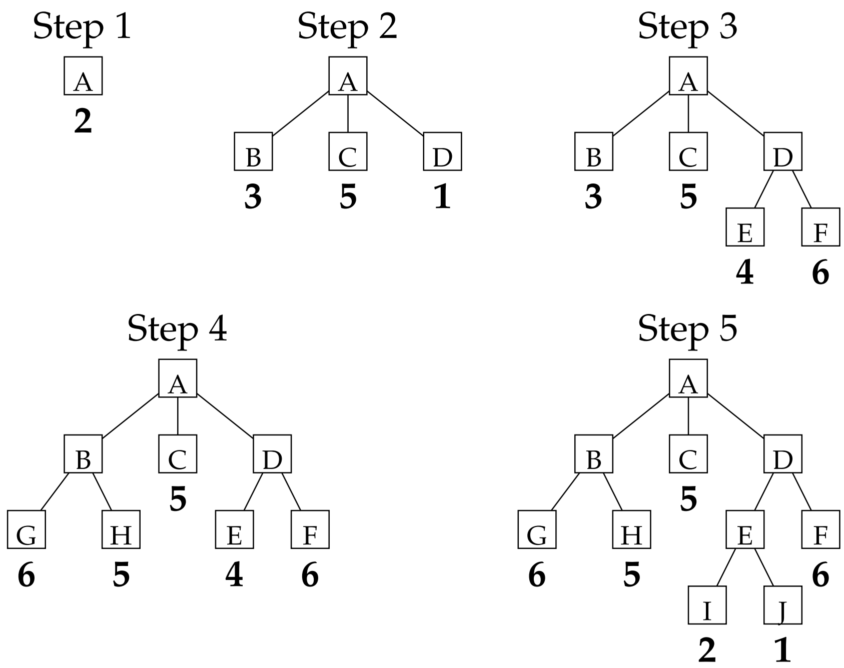
– Examples of informed search strategies now follow:

Best-First Search (ordered search)

- uses an evaluation function to order the nodes on the open list **— the node on the open list having the smallest** `F` **value is expanded first**.
- The algorithm is as follows:

```
1. Place the start node s on the open list.
2. If the list is empty, return failure and exit.
3. If the first element on the list is a goal node, return
   success and exit, otherwise,
4. Remove that element from the list with the smallest F value
   (call it n) and place it on the closed list, expand n and
   place its successors on the open list together with their
   associated F values - provide the successors with pointers
   back to n.
5. Return to step 2.
```

- The best-first search algorithm always selects the most promising node on the open list for further expansion — however, since it is using a heuristic that may prove erroneous, it does not abandon all other states but maintains them on the open list.
- The best-first approach is illustrated in the figure below where numbers by the nodes may be regarded as values determined by an evaluation function and which indicate the 'promise' of a node:



- Best-first searches will always find good paths to a goal, even when local anomalies are encountered — all that it requires is a good evaluation function which will give a reasonable measure of the distance of a node from the goal node.
- It should be noted that the cost of computing the evaluation function can, in some cases, be significant.

The A* Algorithm

- The choice of evaluation function critically determines the search results — an evaluation function that ignores the true promise of some nodes may result in non-optimal solution paths, whilst functions that over-generously acknowledge the promise of all nodes will result in too many nodes being expanded.
- One particular form of evaluation function guarantees a minimal cost path to a goal whilst at the same time maximising one measure of search efficiency.
- At each node along a path to the goal, the A* algorithm generates all successor nodes and computes an **estimate** of the distance (cost) from the start node to a goal node through each of the successors — it then chooses the successor with the shortest estimated distance for expansion — the successors for this node are then generated, their distances estimated, and the process continues until a goal is found or the search ends in failure.
- The form of the heuristic estimation function for A* is:

$$f^*(n) = g^*(n) + h^*(n)$$

where $g^*(n)$ is an estimate of the cost (or distance) from the start node to node n, and $h^*(n)$ is an estimate of the cost (or distance) from node n to a goal node — the asterisks are used to designate **estimates** of the corresponding true values

$$f(n) = g(n) + h(n) \ .$$

- the choice of $g^*(n)$ is obvious since it is the cost of the path in the search tree from s to n - this will be the lowest cost found so far and therefore implies that $g^*(n) = g(n)$ — note that this is not true in general for graphs, since alternate paths from the start node to n may exist.
- The A* algorithm is as follows:

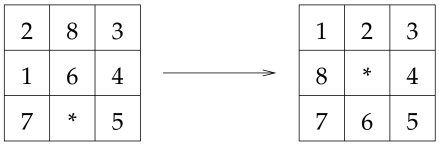
1. Place the start node on the open list.
2. If the list is empty, exit and return failure.
3. Remove from the open list that node that has the smallest value of $f^*(n)$ - if the node is a goal node, return success and exit.
4. Expand n, generating all its successors and place n on the closed list - for every successor, if it is not already on the open or closed list, attach a pointer to n, compute $f^*(n)$ and place it on the open list.
5. For each successor node that is already on the open or closed list, attach pointers which reflect the lowest $g^*(n)$ path - if the successor node was on the closed list and its pointer was changed, remove it and place it on the open list.
6. Return to step 2.

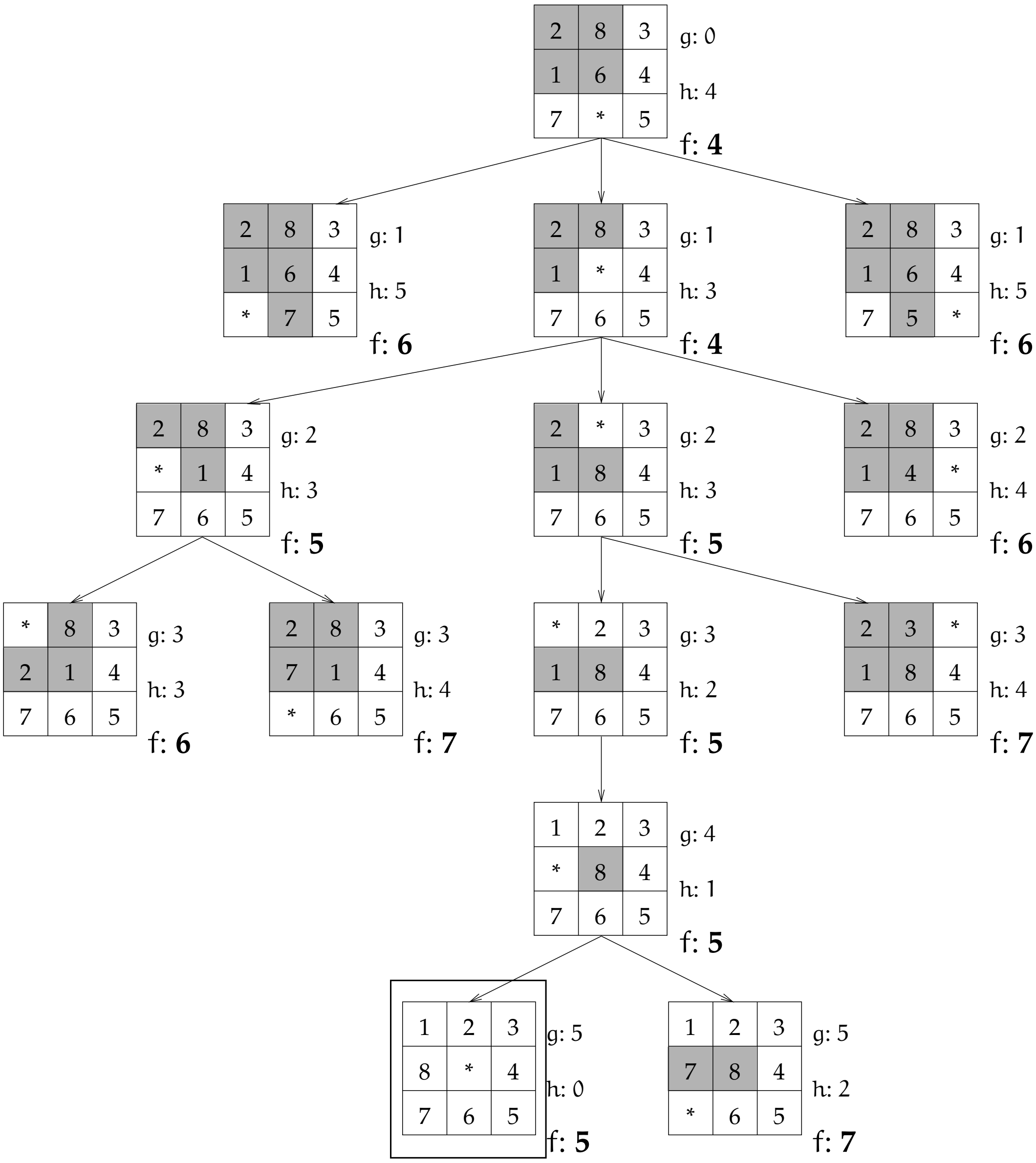
– The way that this algorithm works can best be illustrated by reconsidering the 8-puzzle. A simple evaluation function for the 8-puzzle is given by:

– $g^*(n) =$ the length of the path from the start node to node n, that is, the depth of node n,

– $h^*(n) =$ the number of misplaced tiles from their goal position.

– The graph obtained by applying the A* algorithm to the original 8-puzzle using the above evaluation function is shown below:





In this example 14 nodes were considered of which 6 were expanded — thus a reduction has been obtained from the previous algorithms. (Best-first search expanded 9, breadth-first search 58 and depth-first search even 178008 states with a particular order of operators.)

Admissibility

– An admissible heuristic is one that finds the shortest path to a goal whenever it exists.
– The breadth-first search is an admissible search strategy.
– It is possible to prove that if h^* is a lower bound on h, then A* is an admissible algorithm. So an evaluation function, h', is admissible if $h'(n) \leqslant h(n)$ for all n, where $h(n)$ is the actual distance to the nearest goal.
– The heuristic function $h(n) = 0$ is admissible since it leads to expanding nodes in order of increasing $g(n)$ — since $g(n)$ is the depth of a node then the A* algorithm becomes the breadth-first search in this case.

Informedness

– Assume that in order to solve a particular problem we adopt two *admissible* A* search strategies, $A^*{}_1$ and $A^*{}_2$ such that for node x

$$\begin{array}{ll} A^*{}_1 & f_1^*(x) = g^*(x) + h_1^*(x) \\ A^*{}_2 & f_2^*(x) = g^*(x) + h_2^*(x). \end{array}$$

If $h_1^*(x) \leqslant h_2^*(x)$ for all states x in the search space, then heuristic h_2^* is said to be more **informed** than h_1^*.
– In this case, h_2^* will evaluate fewer states than h_1^* — the set of states examined by h_2^* is a subset of those expanded by h_1^*.

Notes

– The g function determines how good a path to a node is whilst the h function determines how good the node itself is.
– Thus by incorporating g into f we will always expand the node that appears to be closest to the goal.
– If we are only interested in finding a solution then we can define $g = 0$.
– If we want to find a path involving the fewest number of steps then we can set $g =$ depth, that is, the cost of going from a node to its successor is 1.
– If we want to find the cheapest path then we can set $g =$ the sum of costs of going from one node to another along the path from the initial node.

Unit 4. Search 3/3

Learning Outcomes. You should be able to

- describe problem-reduction approach to problem solving
- represent problem reductions as AND/OR graphs
- read AND/OR graphs correctly
- define and give examples of key operators
- describe and apply the means-ends analysis
- define the three high level goals and algorithms of GPS
- explain the difference between strong and weak AI methods

The Problem-Reduction Approach

– In contrast to state-space search the problem reduction approach attempts to solve problems by breaking a problem down into simpler sub-problems — if this process is continued eventually some sub-problems will be reached whose solutions are regarded as known or trivial — the solution of these sub-problems then constitutes a solution of the original problem.

– It is often convenient to describe a problem in terms of the elements of a state-space description — these are:

S is the set of starting states,

F is the set of operators that map one state description into another,

G the set of goal states.

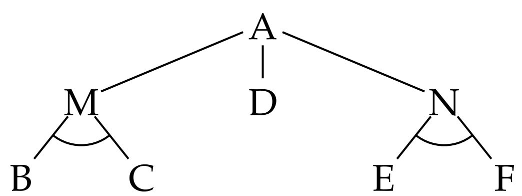
– Thus the triple (S, F, G) defines a problem and can be used as a problem description.

– When problems and sub-problems are described by (S, F, G) triples the sub-problems are easily recognised as problems of finding paths between certain stepping-stone states in the state space.

– The decomposition of a problem can be represented by a special kind of graph known as an AND/OR graph, in which nodes denote problems to be solved.

AND/OR Graphs

– In an AND/OR graph a circular mark joining the arcs represents conjunction (AND) thus the graph below indicates that a problem A can be solved either by solving problems B AND C OR by solving D OR by solving problems E AND F:

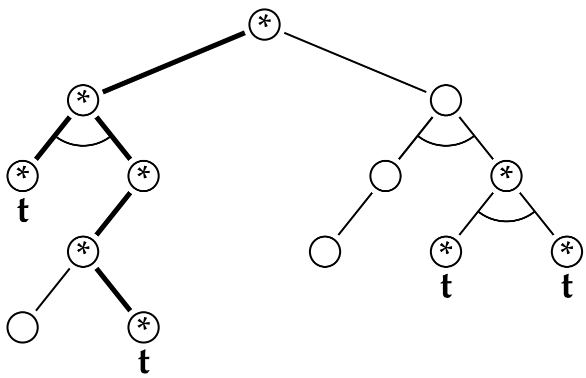


- Nodes whose children are joined with an arc are called AND-nodes and those joined without an arc are called OR-nodes.
- In the special case where there are no AND-nodes in the graph then we have a state-space type graph.
- The object of an AND/OR graph search is to show that the start node (the original problem) can be SOLVED.
- The nodes in the graph that are already solved are called **terminal** nodes.

  To solve an OR-node we only have to solve ONE of its children nodes.

  To solve an AND-node we have to solve ALL of its children nodes.

- Failure to do this will result in conceding that the original problem cannot be solved.
- Below is an example of an AND/OR graph — terminal nodes are indicated by the letter **t**, nodes that can be solved include a star and the solution graphs are indicated by the darkened arcs.



Planning Mechanisms in Problem Reduction

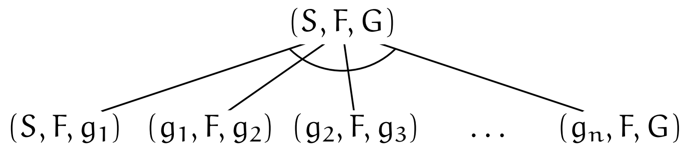
- Suppose that we are given a problem (S, F, G) and we want to reduce it to a set of simpler (state-space search) problems. If we could identify a set of 'stepping-stone' states:

  $$g_1, g_2, g_3, g_4, \ldots, g_n$$

  then the original problem could be reduced to a set of problems specified by the triples:

  $$(S, F, g_1), (g_1, F, g_2), (g_2, F, g_3), \ldots, (g_n, F, G)$$

- Solving all these problems gives a solution to the initial problem:

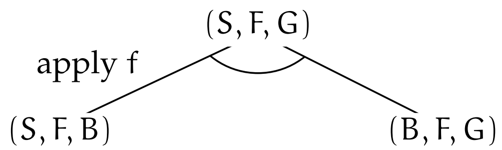


- Furthermore, if the stepping stone states are clearly specified then it makes no difference in which order the successor problems are solved.

  We have assumed in the discussion thus far that we know how to decompose the problem. Now we need to discuss how a problem can be decomposed.

Key Operators

- A problem may have many operators that can be applied and hence the task of finding the entire sequence of operators in the solution could be difficult — however it is often easier to specify the operator necessary for a crucial step in the problem solution — such operators are called **key operators**.
- If a key operator can be determined then this can be used to identify a stepping-stone state in the problem-reduction process. E.g. if f is a key operator and the first state that it can be applied to is state B (say) — there may be a set of these in practice — then we have the problem of getting from B to the goal node — this can be represented by the following AND graph:



- In practice, it may be difficult to identify a single key operator and perhaps the best that can be achieved is to identify a subset of operators thought to be crucial. In this case, each operator in the subset will create a pair of successor problems (as above).
- How can the set of key operators be identified?
- One way is to use the method based on **'differences'**
- This method of finding key operators involves calculating the difference between a problem state and the goal state, i.e. why is a particular state not the goal state? There may be several reasons for this. If they can be ranked in some order of importance then we can use the most important unsatisfied condition as the 'difference'.
- This is the approach taken in **'means-ends analysis'**.

Means-ends analysis

- A technique used by the General Problem Solver (GPS) of Newell and Simon to guide search through a problem space.
- The main principle of means-ends analysis is to detect differences between the current state and the goal state. One difference is isolated — usually the most significant — and an operator is applied to reduce that difference. If the operator cannot be applied, then a subproblem is identified of getting to a state in which it can be applied. This process of removing differences between the current state and the goal state continues until all the differences have been removed, in which case we have found a solution path from the initial state to the goal state.
- Assume, for example, that we wish to travel from location A to location B and that the difference between the locations is given simply by the geographical distance.
- The table below shows a possible list of operators available for reducing geographical distance:

Distance (miles)	Aeroplane	Train	Car	Walk
> 500	✔			
100–500		✔		
2–100			✔	
0–2				✔

– GPS was organised in terms of goals and methods for achieving these goals.

– In GPS there were the following 3 specific goals:

Transform object A into object B.

Reduce difference D between object A and object B.

Apply operator Q to object A.

– A more detailed description of each is shown in Figure 1 overleaf.

– The list of operators relevant to a difference must be ranked in order to indicate the most likely operator

– and the differences themselves had to be placed in order of difficulty so that GPS would attempt to reduce the most difficult difference first. This heuristic helped GPS to restrict the expansion of the tree of subgoals generated during the course of the problem.

Weak/Strong methods

– Problem-solving paradigms which require only a weak understanding of the domains to which they are applied are said to use **weak methods** — the reliance here is on techniques not any knowledge of the domain. GPS is an example of a problem-solving system using weak methods.

– Problem-solving paradigms which use a knowledge of the problem domain are said to use **strong methods** — expert systems, which will be discussed later, fall into this category.

– AI practitioners now recognise that strong methods are far superior to weak methods.

Means-ends analysis applied to the monkey and bananas problem

Monkey and bananas problem

A monkey is in a room containing a box and a bunch of bananas. The bananas are hanging from the ceiling out of reach of the monkey. What sequence of actions will allow the monkey to get at the bananas?

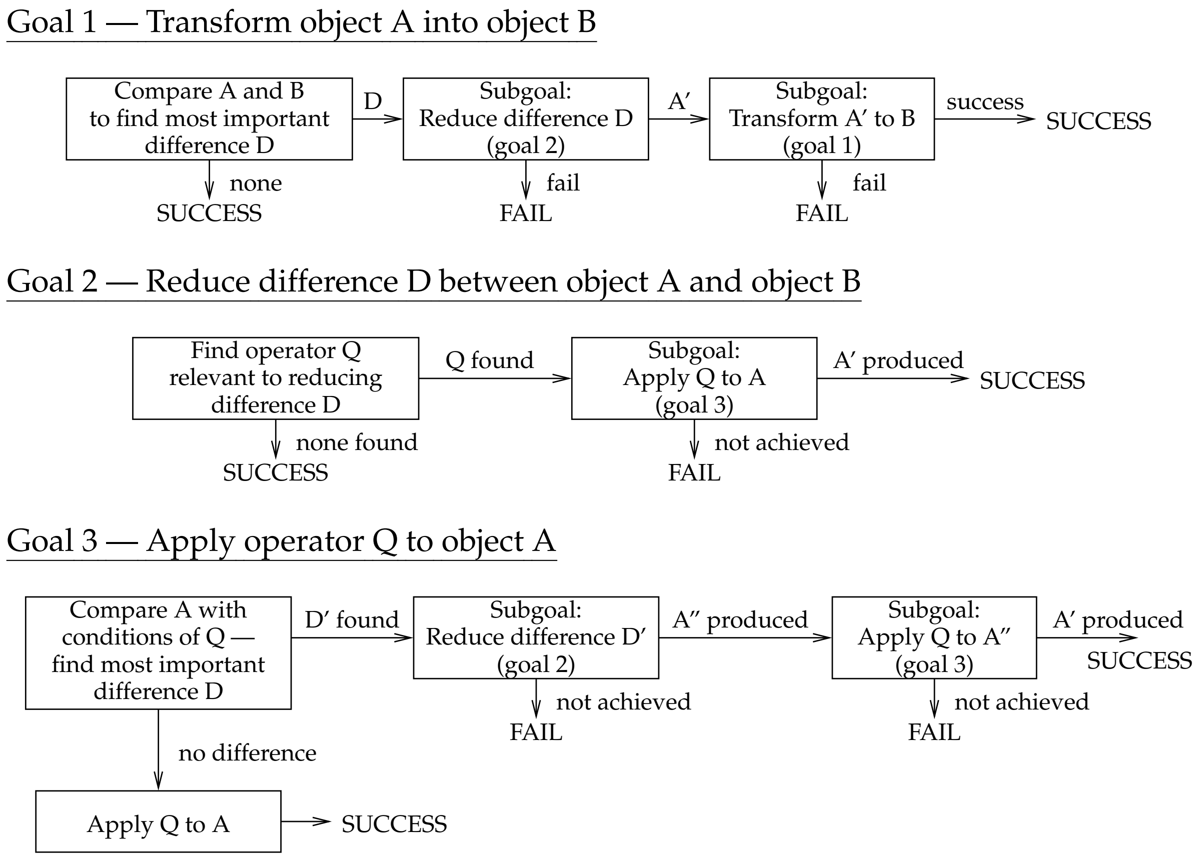


Figure 1: Goals and Algorithms of GPS

– Clearly the significant elements of the problem are:

the monkey's position in the room,
the box's position in the room,
whether the monkey has the bananas.

– For this problem state description schemas of the following form can be used:

$$(w, x, y, z)$$

where

$w =$ horizontal position of monkey (a 2-D vector),
$x = 1$ when monkey was on top of the box, 0 otherwise,
$y =$ horizontal position of the box (a 2-D vector),
$z = 1$ when the monkey has the bananas, 0 otherwise.

Thus the problem for the monkey to solve is:

$$(a, 0, b, 0) \longrightarrow (c, 1, c, 1)$$

where c is the floor location directly under the bananas.

– The 4 operators available to the monkey are:

$$\begin{array}{lrcl} \textit{goto}(u): & (w, 0, y, z) & \longrightarrow & (u, 0, y, z) \\ \textit{pushbox}(v): & (w, 0, w, z) & \longrightarrow & (v, 0, v, z) \\ \textit{climbbox}: & (w, 0, w, z) & \longrightarrow & (w, 1, w, z) \\ \textit{grasp}: & (c, 1, c, 0) & \longrightarrow & (c, 1, c, 1) \end{array}$$

– Applying means-ends analysis we note that $(a, 0, b, 0)$ fails to satisfy the goal state because the final element is not a 1 — we can reduce this difference by applying the *grasp* operator.

– However, we cannot apply the *grasp* operator because

the box is not at c, the monkey is not at c and the monkey is not on the box.

– Taking this list of statements as the difference in this case, we identify the following key operators:

$$\textit{pushbox}(c), \textit{goto}(c), \textit{climbbox}$$

– The *pushbox* operator cannot be applied to the state $(a, 0, b, 0)$ because the monkey is not at position b. To reduce this difference we have to apply the *goto*(b) operator obtaining state $(b, 0, b, 0)$.

– The *pushbox*(c) operator can now be applied giving $(c, 0, c, 0)$, and then the *climbbox* operator can be applied to give $(c, 1, c, 0)$ and finally the *grasp* operator to give the goal state $(c, 1, c, 1)$.

GPS algorithm execution for the monkey and bananas problem

A summary of the meaning of state tuples:

$$\left(\text{monkey's position}, \frac{1 = \text{monkey on box}}{0 = \text{monkey on floor}}, \text{box's position}, \frac{1 = \text{monkey has banana}}{0 = \text{monkey is hungry}}\right)$$

G1 Transform $A = (a, 0, b, 0)$ to $B = (?, ?, ?, 1)$

- step1: $D = \text{pos4}$
- step2: G2 Reduce difference $D = \text{pos4}$ between $A = (a, 0, b, 0)$ and $B = (?, ?, ?, 1)$
  - step1: $Q =$ *grasp* (i.e. $(c, 1, c, 0) \longrightarrow (c, 1, c, 1)$)
  - step2: G3 Apply operator $Q =$ *grasp* to state $A = (a, 0, b, 0)$
    - step1: $D' = \text{pos3}$
    - step2: G2 Reduce difference $D = \text{pos3}$ between $A = (a, 0, b, 0)$ and $B = (c, 1, c, 0)$
      - step1: $Q =$ *pushbox*(c) (i.e. $(w, 0, w, z) \longrightarrow (c, 0, c, z)$)
      - step2: G3 Apply operator $Q =$ *pushbox*(c) to state $A = (a, 0, b, 0)$
        - step1: $D' = (\text{pos1} \neq \text{pos3})$
        - step2: G2 Reduce difference $D = (\text{pos1} \neq \text{pos3})$ between $A = (a, 0, b, 0)$ and $B = (w, 0, w, z)$
          - step1: $Q =$ *goto*(b) (i.e. $(w, 0, y, z) \longrightarrow (b, 0, y, z)$)
          - step2: G3 Apply operator $Q =$ *goto*(b) to state $A = (a, 0, b, 0)$
            - return: $(b, 0, b, 0)$
          - return: $(b, 0, b, 0)$
        - $A'' = (b, 0, b, 0)$
        - step3: G3 Apply operator $Q =$ *pushbox*(c) to state $A = (b, 0, b, 0)$
          - return: $(c, 0, c, 0)$
        - return: $(c, 0, c, 0)$
      - return: $(c, 0, c, 0)$
    - $A'' = (c, 0, c, 0)$
    - step3: G3 Apply operator $Q =$ *grasp* to state $A = (c, 0, c, 0)$
      - step1: $D' = \text{pos2}$
      - step2: G2 Reduce difference $D = \text{pos2}$ between $A = (c, 0, c, 0)$ and $B = (c, 1, c, 0)$
        - step1: $Q =$ *climbbox* (i.e. $(w, 0, w, z) \longrightarrow (w, 1, w, z)$)
        - step2: G3 Apply operator $Q =$ *climbbox* to state $A = (c, 0, c, 0)$
          - return: $(c, 1, c, 0)$
        - return: $(c, 1, c, 0)$
      - $A'' = (c, 1, c, 0)$
      - step3: G3 Apply operator $Q =$ *grasp* to state $A = (c, 1, c, 0)$
        - return: $(c, 1, c, 1)$
      - return: $(c, 1, c, 1)$
    - return: $(c, 1, c, 1)$
  - return: $(c, 1, c, 1)$
- step3: G1 Transform $A = (c, 1, c, 1)$ to $B = (?, ?, ?, 1)$. DONE

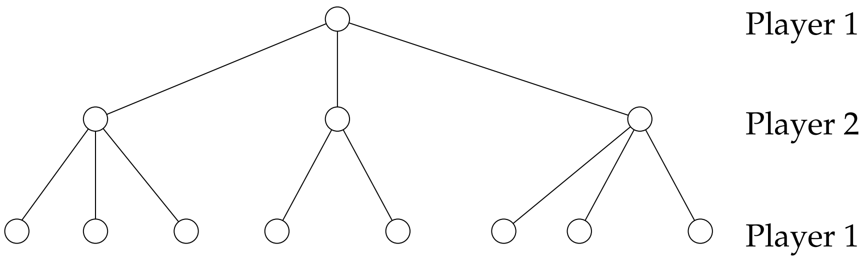
Unit 5. Game Playing

Learning Outcomes. You should be able to
- describe and apply the (depth-bounded) minimax procedure
- list two methods of assessing how good is a particular evaluation function used in the minimax procedure
- state how to compensate for a less accurate evaluation function in the minimax procedure
- describe and apply the alpha-beta procedure
- list three ways of improving the minimax and alpha-beta procedures, explain why these are improvements

Game Playing

– Games have always been an important application area for heuristic algorithms since they provide some interesting opportunities for developing and testing heuristics.

– A (two-player) game can be represented by a directed graph of the form shown below:



– Several game-playing algorithms have been proposed. Possibly the most popular one is:

Minimax Procedure

– Originally introduced into game playing by Shannon (Sci. Amer 1950, **48**, 182).

– Based on the strategy that your opponent will always make the move that causes you the maximum loss.

– Assume that we have an evaluation function through which every game position can be assigned a numerical value of 'promise'.

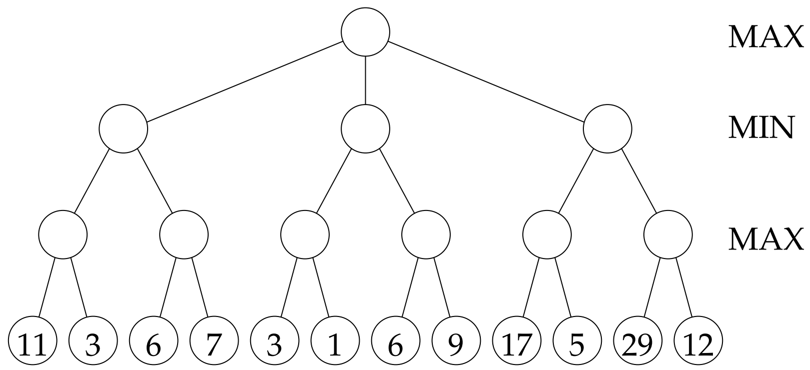
– The convention we shall adopt is that the more positive the numerical value the greater the promise or desirability of the game position — obviously the converse will apply to our opponent.

– The opponents in a game are referred to as MIN and MAX. Although this is partly for historical reasons, the significance of these names is straightforward — MAX represents the player trying to win, or to MAXimise their advantage, whilst MIN is the opponent who attempts to MINimise MAX's score.

– We assume that MIN always uses the same information and always attempts to move to a state that is worst for MAX.
– In implementing minimax, we label each level in the search tree according to whose move it is at that point in the game, MIN or MAX — in the example shown below MAX is allowed to move first.
– The values in the leaf nodes are static values derived from some (heuristic) evaluation function appropriate for the game.
– Minimax propagates these values up the graph through successive parent nodes according to the rule:

  If the parent state is a MAX node, then give it the maximum value among its children.

  If the parent state is a MIN node, then give it the minimum value among its children.



– Ideally, the computer should be able to search through the entire tree down to the leaf nodes which represent the end of the game — but because of the large number of states generated for non-trivial games this is rarely possible except near the end-game.
– The diagram in Figure 2 shows the minimax procedure applied to the game of noughts-and-crosses (tic-tac-toe).

Notes

– Suppose that the evaluation function used is infallible and so the promise of a node is calculated accurately — in such a case we would need to construct a game tree of only depth 1.
– This is difficult to achieve in practice however, so we must search the game tree down to some depth if we are to make a good move since the leaf nodes are expected to be closer to game termination than the non-leaf nodes.
– Clearly, the effectiveness of the game playing system is heavily dependent on the evaluation function — thus how can we increase the accuracy of the evaluation function?

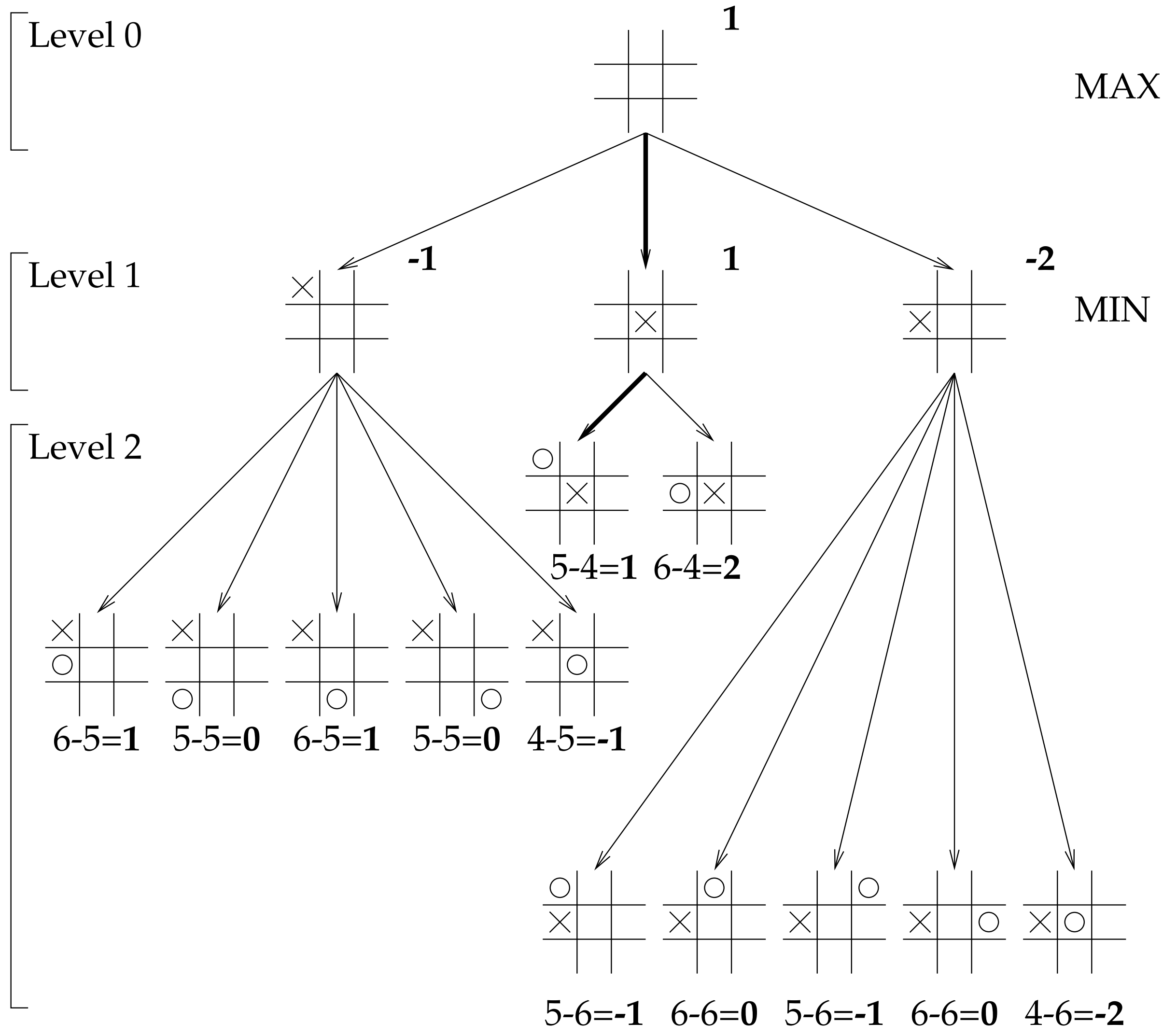


Figure 2: Minimax procedure applied to the opening moves of noughts-and-crosses. States that are symmetrical to some earlier encountered state are omitted.
The evaluation function estimates how much advantage crosses have over noughts. It is computed as the the number of potential winning triples (in rows, columns and diagonals) for crosses minus the same for noughts.

- For the root node (or current node) we can calculate the promise of the node **directly** from the evaluation function and then compare this value against that propagated up from the leaf nodes — the closer the two values the more accurate is the evaluation function — thus if over the game tree searched the difference

  $$|\text{promise of the node calculated directly} - \text{propagated value}|$$

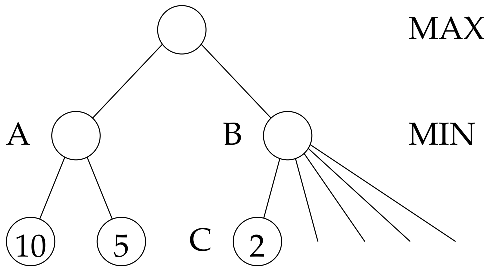
  is on average greater than some threshold value then the evaluation function may need to be amended.
- Additionally, a comparison can be made between the moves calculated by the minimax procedure and those suggested by experts or textbooks — again amendments can be made to the evaluation function to obtain agreement.

The Alpha-Beta Procedure

– In the minimax procedure, the process of position evaluation only begins after tree generation is completed. Thus the process of tree generation is completely separate from the process of position evaluation which is rather inefficient.

– A considerable reduction in the search effort can be achieved if backed up values are calculated **at the same time** as tree generation. The alpha-beta procedure accomplishes this and in so doing will **always** choose the same move as the minimax procedure but with much less search effort.

– Consider the following fragment of a game tree:



– Two values, called alpha and beta are created during the search. The alpha value associated with MAX nodes, can never decrease, and the beta value, associated with MIN nodes, can never increase.

– In the above example, MAX's alpha value is 5 for node A (taken from the backed-up value from the left fragment of the tree). As MAX will be *maximising* his score we know immediately that MAX can get a score of at least 5 without even examining node B. Evaluating node C, we find that this has a value of 2 — thus the score for node B therefore cannot exceed 2 since MIN is trying to *minimise* his score. Thus there is no point in evaluating any more nodes in this part of the game tree since it will not affect the eventual outcome, namely, that MAX will choose node A.

– To begin alpha-beta search, we descend so many levels in a depth-first fashion and apply our heuristic evaluation to the states. The maximum of these values is then backed up to the parent (assume it is a MAX node, otherwise it will be the minimum value that is backed up). This value is then offered to the grandparent as a potential beta cut-off. Next, the algorithm descends to other grandchildren and terminates exploration of their parent if any of their values is equal to or larger than this beta value.

  Two rules for terminating search, based on alpha and beta values are:

  1. Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
  2. Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN node ancestors.

– The diagram in Figure 3 shows the alpha-beta approach applied to the opening moves of noughts-and-crosses.

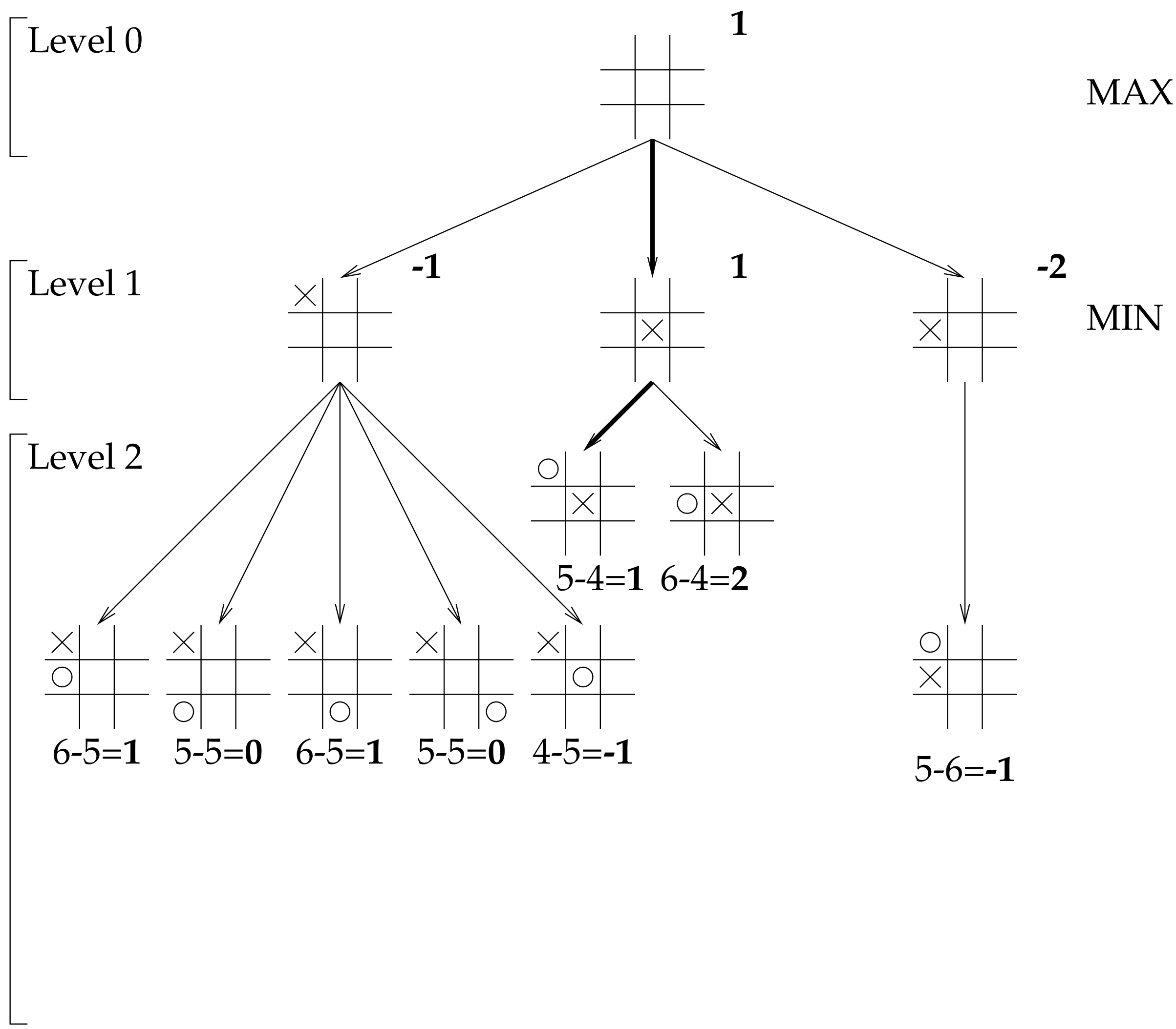


Figure 3: Alpha-beta procedure applied to the opening moves of noughts-and-crosses.

Additional refinements

– The approach to game playing just described suffers from a number of limitations.

Quiescent state

– Some portions of a game tree may be quiescent (i.e. tactically uninteresting) whilst other portions are non-quiescent, involving, for example, long tactical exchanges of pieces.

– Tactical portions of the game tree can be identified by rapidly changing values of the evaluation function.

– These portions of the game tree should be searched to greater depth until either a depth bound is exceeded or a state of quiescence reached.

Horizon effect

– If a game playing program searches all paths to the same depth then a ″horizon″ is established beyond which unknown disasters can lurk.

– The danger here is that low-valued pieces can be seized along paths which will lead — just over the horizon — to the loss of high-valued pieces later.
– More difficult to eliminate.
– One approach to combat the horizon effect is to conduct a shallow search beyond the apparently best move.

Book moves

– Book moves (sequences of moves recommended by experts in certain complex crucial situations) can be used, especially in the opening and ending sequences of a game.
– The program must recognise when the opening sequence ends and the ending sequence begins.

Game playing programs

Chess

– Received the greatest attention.
– Deep Blue uses a parallel array computer of 1024 VLSI chips — this enables it to search the equivalent of 1 billion positions/sec. and to reach a depth of 14.
– The program has a rating of 2600 putting it among the top 100 human players.

Checkers draughts

– Samuel's program is probably the most famous here — it was developed in 1952 and subsequently improved to a point where it regularly beat Samuel.
– Another program — Chinook — won the 1992 US Open in Checkers using an alpha-beta search.

Others

– Othello — computer programs better than humans.
– Backgammon — Gerry Tesauro used a neural network technique to develop a program which is ranked among the top 3 players of the world.
– Go — branching factor here approaches 360, so brute force search techniques are hopeless. This needs more sophisticated reasoning methods.

Unit 6. Knowledge Representation

Learning Outcomes. You should be able to

- outline production system architecture and explain its functioning
- define and apply forward and backward chaining
- correctly write and read semantic networks
- list several predicates common in semantic networks
- explain and use inheritance within semantic networks (including exceptions)
- describe and use intersection search
- list three problems of semantic networks
- explain what a frame is and give an example
- list the kinds of items that may be stored in frame slots
- introduce the concept of script and give an example
- list some strengths and weaknesses of frames and scripts

Several knowledge representation schemes have been used in AI programs — we shall consider some of these.

Production Systems

– A production system is a set of rules, called **productions**, that embodies knowledge about a particular domain, and which can be used for making inferences in that domain.

– In AI, production systems are also known as rule-based systems.

– A production states that if a certain set of conditions holds then a certain set of actions can be performed — the terms condition and action are interpreted broadly.

– Production systems are modular — new productions corresponding to new bits of knowledge can be added without affecting the way the production system works.

– A rule can be considered to consist of a condition and an action, also referred to as 'antecedent' and 'consequent' — the general form of which is:

IF C THEN A

which stands for: 'if the conditions C are true then perform the actions A'.

Overview

A system based on production rules will usually have 3 components:

1. A rule base, consisting of a set of production rules.
2. A working memory, consisting of one or more data structures which contain the known facts relevant to the domain of interest.
3. An interpreter which has the special task of deciding which rule to fire next and then initiating the corresponding action. It may happen that more than one rule is enabled and can fire — to establish which of the enabled rules to fire, the interpreter will invoke a **conflict resolution strategy** which will determine the candidate. Several conflict resolution strategies have been proposed.

The rules have a syntax which is known to the interpreter, which can therefore manipulate these logically deciding on their truth or otherwise.

Classification of interpreters

One way of characterising an interpreter (or inference engine) is according to the way in which it tries to apply the rules — there are two possible approaches:

In **forward chaining**, we start from a set of initial conditions and perform the actions described. This step creates a new set of conditions in which further productions are triggered. This process continues until the required conclusion is reached.

In **backward chaining**, the goal or statement to be proved is taken as the starting point. The set of productions which refer to this goal are then tested to see if their conditions are true — if so the goal has been reached, otherwise, an attempt is made to prove the conditions are true by considering other production rules which refer to the conditions. This process continues until either the goal has been reached or all the rules have been considered in which case the goal cannot be reached

Simple example

Rule Base:

R1	IF	suckles_young	THEN	mammal
R2	IF	has_feathers	THEN	bird
R3	IF	has_fur or mammal	THEN	lives_in_forest
R4	IF	bird and does_not_fly and does_not_live_in_forest	THEN	penguin
R5	IF	lives_in_forest and very_heavy	THEN	bear
R6	IF	very_heavy and mammal	THEN	whale

Initial Facts:

$$\text{FACTS} = (\text{suckles\_young}, \text{very\_heavy})$$

Execution — forward chaining

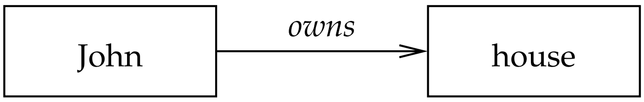
Interpreter is invoked with the conflict resolution strategy of always picking the rule with the lowest serial number.

1. R1 fires
   WM = (suckles_young, very_heavy, mammal)
2. R1, R3 and R6 are enabled — R1 is rejected because it duplicates a property already known — so R3 is selected.
   WM = (suckles_young, very_heavy, mammal, lives_in_forest)
3. R1, R3, R5 and R6 are enabled — R5 is selected.
   WM = (suckles_young, very_heavy, mammal, lives_in_forest, bear)
4. R6 is now selected, executed and the system halts.
   WM = (suckles_young, very_heavy, mammal, lives_in_forest, bear, whale)

In the case of **backward chaining** we would hypothesise that the animal was a bear and then attempt to prove this.

Semantic networks

- Used to encode information about meaning — when used for other purposes they have been referred to as *associative networks*.
- Introduced by Quillian to model the semantics of English sentences and words.
- are directed graphs in which nodes represent objects or concepts in the problem domain and the arcs represent relations or associations between them. for example:



- a number of arc relations have become common among users — they include such predicates as:

  isa, member-of, subset-of, ako (a-kind-of), has-parts, instance-of, agent, attributes, shaped-like
- An example of a semantic network is shown in Figure 4.
- When we talk about parent and children concepts, we mean those that are linked by the *isa* relation.
- Properties possessed by parent concepts are often '**inherited**' by their children. (E.g. a Bird *can* Breathe because Bird *isa* Animal and Animal *cam* Breathe.)
- Shared properties are attached only to the highest node in the structure to which they apply.
- Several benefits come from the inheritance of properties — they allow information to be stored at the highest level of abstraction, which reduces the size of the knowledge bases and helps prevent inconsistencies caused by improper updates.
- Inheritance can be recognised as a form of default reasoning.
- Semantic networks implement inheritance — for example, the canary inherits all the properties of birds — thus the network could be used with appropriate inference rules to answer a range of questions about canaries, birds and ostriches. These inferences are made by following the appropriate links to related concepts.

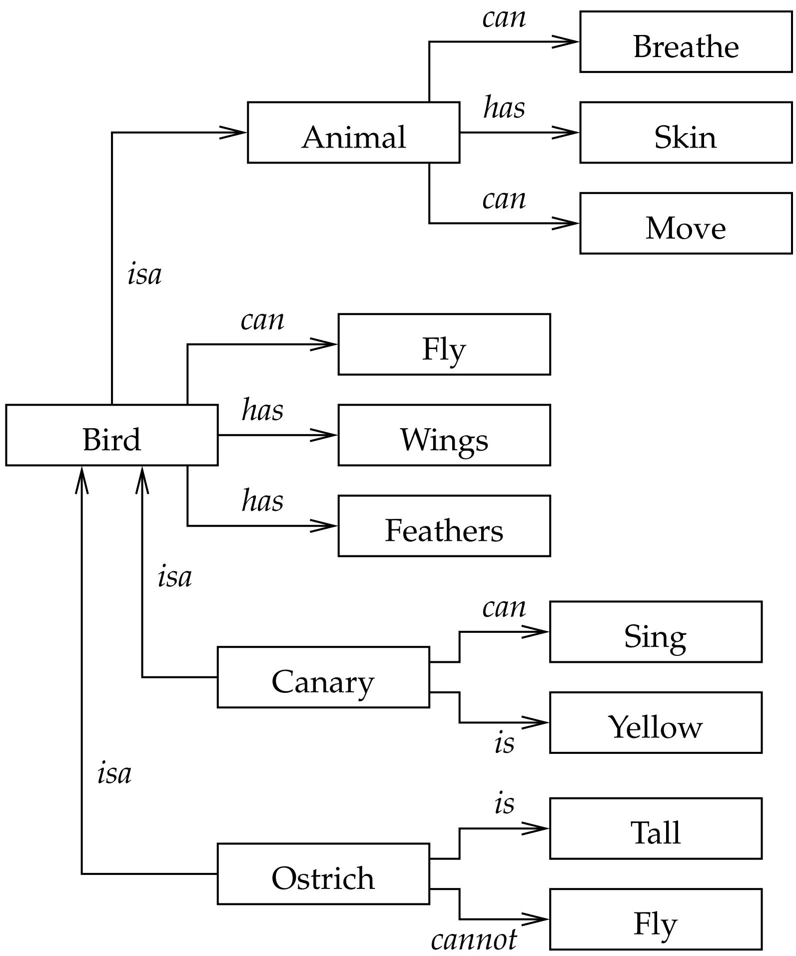


Figure 4: A Semantic Network

- When exceptions are to be dealt with then this can be done at the specific level. (E.g. Bird *can* Fly in general but Ostrich is an exception, explicitly marked in the network.)
- To establish a relationship (i.e. a path of links) between two concepts, we can use an **intersection search** in which we move outwards by one link from each concept in every step. If this spreading activation intersects then a relationship will have been established between the concepts since there is a path from each to the other. The nature of this relationship will naturally depend upon the links that were traversed in establishing the intersection. For example, consider the network in Figure 5 and note that relationships between FRED and Meat found by the intersection search are:
  - FRED is an Animal and so is Cow which gives Meat.
  - FRED owns FIDO who eats Meat because he.is a Dog.
- There are several problems with semantic nets, including:
  1. Notation has not been uniform.
  2. The intersection search 'explodes' very quickly as the number of links increases — raising the possibility of **combinatorial explosion**.

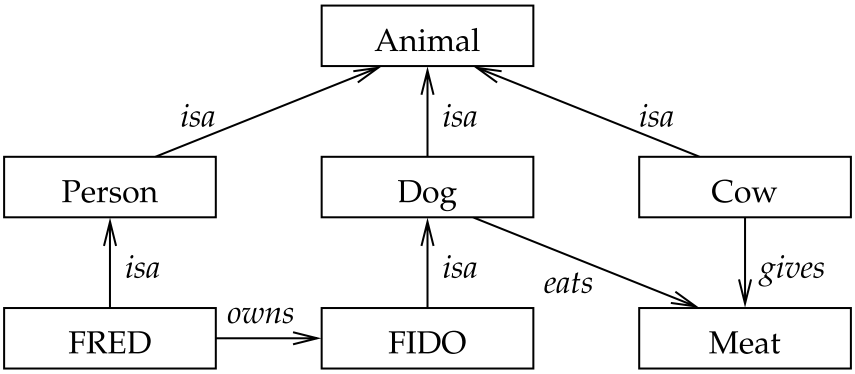
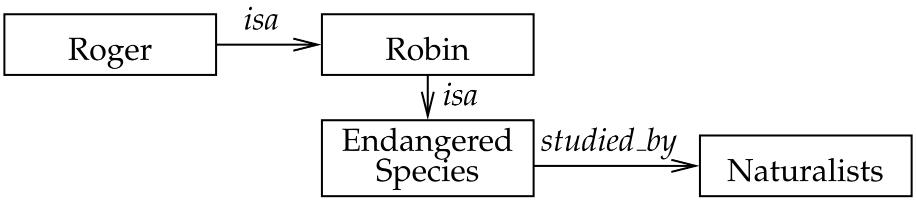


Figure 5: A Semantic Network

3. Difficult to distinguish between an individual and a class of individuals:



We conclude that Roger is studied by naturalists which may or may not be true. The problem is that Roger *is a particular* Robin while Robin *as a class of birds* is an Endangered Species.

Frames

– First introduced by Marvin Minsky (1975).

– Minsky described frames as 'data structures for representing stereotyped situations', originally proposed as a basis for understanding visual perception, natural language dialogue and other complex behaviour.

– For example, the prototypical car has four wheels, a chassis, engine etc. This information could be stored in a frame representing all the common attributes of all cars. Any specific car could then inherit this information from the prototypical representation.

– A frame is a description of an object that contains slots for all the information associated with the object.

– Slots may store values, default values, pointers to other frames, sets of rules, or procedures by which values may be obtained.

– An example of a frame of a particular car is in Figure 6.

– The top levels of a frame are fixed, and represent things that are always true about the supposed situation — the lower levels have many slots that must be filled by specific instances of data.

– Collections of related frames are linked together into **frame systems**.

– A property inheritance hierarchy may be established among the frames.

CAR

Slots	**Entries**
Owner	Jones
Type	Saloon
Colour	Blue
Wheels	Default: 4
Doors	Default: 4
Engine	1600cc
Petrol consumption	If needed — see handbook
Registration	Unknown
Insurance class	If needed — consult company

Figure 6: A frame for a car

- Default and inherited values are relatively inexpensive methods of filling in slots — they do not require powerful reasoning processes.
- The inclusion of procedures in frames joins together in a single representational strategy two complementary (and historically, competing) ways to state and store facts — **procedural and declarative representations**.
- Declarative and procedural representations are alternative strategies which achieve the same effect — frames gain power, generality and popularity by their ability to integrate both representations.
- A number of frame-based languages have been developed (e.g. KRL, FLAVORS, LOOPS) — these typically have functions to create, access, modify, update and display frames.

Schemata

- A frame is often associated with a class of objects or a category of situations. E.g. a frame "vacation" may provide slots for all the usual important features of a vacation: where, when, principal activities, cost etc.
- Frames for particular vacations are then created by **instantiating** this general frame. Such a general frame is called a **schema** and the frames produced by instantiating the schema are called **instances**.
- To instantiate a schema a new frame is created by allocating memory for it and then filling in the necessary slots — note not all slots need to be filled.

Scripts

- Used to represent sequences of commonly occurring events — introduced by Roger Schank for understanding natural language texts.
- A script is a predefined frame-like structure which contains expectations, inferences and other knowledge relevant to a stereotypical situation.

– They are constructed using basic primitive concepts and rules of formation e.g.

script name	supermarket
roles	shopper checkout person stacking attendant seafood/meat attendant other shoppers
entry conditions	shopper needs groceries supermarket open
props	shopping carts market items display aisles checkout units cashier money
scene 1	enter supermarket shopper enters supermarket shopper acquires shopping cart
scene 2	shop for items shopper moves through aisles shopper transfers items to shopping cart
scene 3	check out shopper moves to checkout stand ... supermarket sacker gives bags to shopper
scene 4	exit market shopper exits market

– Using the script, the system can then answer relevant questions to the above scenario. E.g. if it is known that John entered a supermarket, it can be inferred that he needed groceries, shopped for items, paid for them so that when he left the supermarket he had the relevant items but had less money than when he entered the supermarket.

– Scripts have the difficulty of matching the problem being considered with the relevant script. It is often difficult to determine which of two or more potential scripts should be used, no algorithm exists for guaranteeing correct choice.

– The script representation has a certain inflexibility so that it cannot deal with any untoward events, e.g. fire in supermarket curtailing acquisition and payment of items.

Unit 7. Expert Systems

Learning Outcomes. You should be able to
- define what expert system is
- describe a typical user interaction with an expert system
- describe three successful expert systems
- list eight typical tasks tackled by expert systems
- list three weaknesses of expert systems from 1980's
- decide when a task is worthy of developing an expert system for

> *'Machines that lack knowledge seem doomed to perform intellectually trivial tasks. Those that embody knowledge and apply it skilfully seem capable of equalling or surpassing the best performance of human experts. Knowledge provides the power to do work; knowledge engineering is the technology that promises to make knowledge a valuable industrial technology'.*

What is an Expert System?

An expert system (ES) has been defined as 'a computing system which embodies organised knowledge concerning some specific area of human expertise, sufficient to perform as a skillful and cost-effective consultant.'

Alternative terms with a broadly similar meaning are:

> "knowledge-based systems", "expert consulting skills", "rule-based systems", "intelligent knowledge-based systems"

Intelligent knowledge-based systems (IKBS) was one of the four fields singled out by the Alvey Report — a remarkable change in attitude since 1973 when the Lighthill Report almost single-handed stunted the growth of Artificial Intelligence in the UK — AI has gained respectability from the advent of Expert Systems.

Example Expert Systems

AI researchers have built up an impressive collection of expert system programs in diverse areas; the following show some of the achievements:

PROSPECTOR (1974–1983) — discovered a molybdenum deposit of value > \$1,000,000,000.

XCON (R1) (1980–1987) — configures VAX computer systems. Resident experts thought it would not be possible.

DENDRAL (late 1960's) — capable of chemical structure elucidation

MYCIN (mid 1970's) — early and still famous medical diagnosis tool. Identification of bacteria in blood and urine samples; prescription of antibiotics.

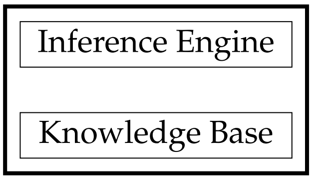
CADUCEUS — embodies more knowledge of internal medicine than any human and can correctly diagnose complex test cases that confuse human experts. In 1982 system possessed 100,000 associations representing nearly 85% of all relevant knowledge.

Example of an Expert System Consultation — MYCIN Dialogue

The example dialogue with the MYCIN expert system in Figure 7 illustrates the capability of expert systems.

Expert System Architecture

The main components of an expert system are shown below:



Knowledge Base (KB)

- Contains the knowledge about a specific domain and **is quite separate from the program**.
- A number of advantages ensue from this architecture:

1. Knowledge can be represented in a more natural fashion than computer code.
2. In developing a KB attention can be given to the acquisition and organisation of knowledge rather than the details of computer implementation.
3. Changes can be made to the KB without creating side effects in other parts of the program code.
4. Removal of the KB provides an **expert system shell** which can be used in variety of systems.

Inference Engine (IE)

- Manipulates the rules to make inferences, diagnoses etc.
- Feigenbaum states that 'expert systems must be knowledge rich even if they are methods poor.'

```
22  Was any chemotherapy recently begun specifically to combat ORGANISM-1?
    NO

23  Is Jeremiah Sample (PATIENT-1) a burn patient?
    RULE

    RULE 047

    IF:     1.  THE SITE OF THE CULTURE IS BLOOD,   AND
            2. THE IDENTITY OF THE ORGANISM IS NOT KNOWN WITH
               CERTAINTY,  AND
            3.  THE STAIN OF THE ORGANISM IS GRAMNEG,  AND
            4.  THE MORPHOLOGY OF THE ORGANISM IS ROD,  AND
            5.  THE PATIENT HAS BEEN SERIOUSLY BURNED

    THEN:      THERE IS WEAKLY SUGGESTIVE EVIDENCE  (.4)  THAT
       THE IDENTITY OF THE ORGANISM IS PSEUDOMONAS
    NO

24  Is the patient's illness with ORGANISM-1  a hospital-acquired infection?
    YES

25  Do you consider Jeremiah Sample (PATIENT-1) to be a compromised host ?
    (alcoholic,  immunosuppressed, etc):
    NO

26  Has Jeremiah Sample  (PATIENT-1)  had any gu surgery or manipulation
    of the urinary tract?  (e.g.  indwelling catheter, etc.)
    WHY

    I AM TRYING TO FIND OUT ABOUT THE IDENTITY OF THE ORGANISM.
    I ALREADY KNOW THAT:
    THE PORTAL OF ENTRY OF THE ORGANISM IS UPPER-GU
    THE MORPHOLOGY OF THE ORGANISM IS ROD
    THE STAIN OF THE ORGANISM IS GRAMNEG
    THE SITE OF THE CULTURE IS BLOOD
    THEREFORE,  IF:
    1. THE PATIENT HAS NOT HAD A GENITO-URINARY MANIPULATIVE
       PROCEDURE, AND
    2. URINARY-TRACT-INFECTION IS NOT A PROBLEM FOR WHICH THE
       PATIENT HAS BEEN TREATED

    THEN:

    THERE IS SUGGESTIVE EVIDENCE (.6)  THAT THE IDENTITY OF THE
    ORGANISM IS E.COLI

    (RULE 156)
```

Figure 7: Example of a MYCIN dialogue

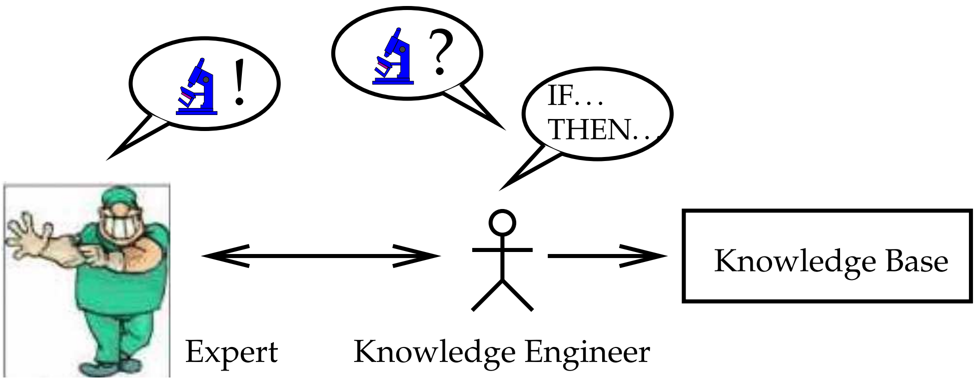
– Patched-together methods are likely to run into trouble with failure to detect all the relevant (or the most relevant) situations. Difficulties include circular applications of subsets of rules, duplications of steps and general computational inefficiency due to **'combinatorial explosion'**. (In a world where there is a combinational rate of increase of paths through the rule-set as the number of steps on the paths increase.)

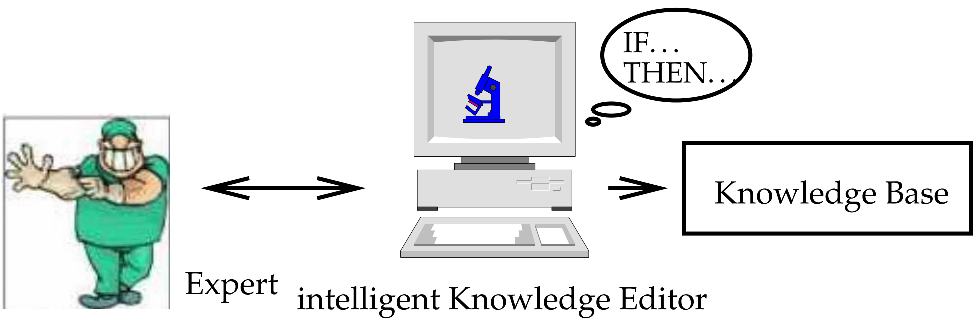
Explanation

– Expert systems should provide an '**explanation**' facility which enables the user to consult the system interactively to ask for explanations of the system's decisions, diagnoses etc. and the 'lines of reasoning' used to obtain them.
– Such explanations (which may involve listing of rules etc.) are themselves given in the form of English text meaningful to the user — not internal codes. Some systems (e.g. MYCIN) provide the capability for the user to examine the system's line of reasoning in detail and suggest modifications to rules or include new rules if this is felt to be appropriate.

Knowledge Acquisition

– It is the task of a "knowledge engineer" to extract the expert knowledge and organise it. To achieve this generally involves a period of intensive discussion with one or more subject experts and the analysis of a selected set of test cases.
– At the moment, KNOWLEDGE ACQUISITION is a bottleneck in the construction of expert systems.
– Knowledge for an expert system can be acquired in several ways — figures below illustrate two possible interactions.





– A knowledge-acquisition method that may become feasible in the future is to acquire the knowledge directly from textbooks using an induction program to build the knowledge base. It is not currently feasible.

Types of Problems Tackled by Expert Systems

Useful to sort applications in terms of problem-solving paradigms:

Control — controls the behaviour of a given system (e.g. control the manufacturing process, the treatment of a patient)

Design — designing a system given possible problem constraints (e.g. design of electronic circuits)

Diagnosis — diagnose faults from given information — sometimes offers prescription (e.g. diagnosis of patient's ills and possible treatment of)

Instruction — to guide the education of a student in a given topic (e.g. GUIDON for teaching students about bacterial infections)

Interpretation — attempts to produce an understanding of a situation from available information — the information may derive from such sources as sensors, instruments, test results etc. (e.g. FXAA provides auditing assistance in foreign exchange trading)

Monitoring — compare observable information on the behaviour of a system against that considered crucial for normal behaviour (e.g. NAVEX monitors position and velocity of the space shuttle)

Planning — planning systems attempt to form plans, actions to achieve a given goal subject to certain constraints (e.g. PLANPOWER provides a wide range of financial plans for wealthy households)

Prediction — attempt to predict the future by inferring likely consequences from a given situation (e.g. likely effect of natural catastrophes on plant production)

Prescription — recommend solutions to overcome given system malfunction (e.g. BLUEBOX recommends appropriate therapy for patients suffering from depression)

Selection — selection systems attempt to identify the best choice from a list of possibilities (e.g. IREX assists in the selection of industrial robots in a work environment)

Simulation — simulation systems model a process or system to permit operational studies under various conditions (e.g. STEAMER simulates and also explains the operation of the Navy's frigate steam propulsion plant to aspiring naval engineers)

Importance and Benefits of Expert Systems

1. Have the capacity to improve qualitative factors (better decisions, enforce consistent methods, preserve expertise).
2. Encapsulate codified knowledge which can then be evaluated.
3. Expert Systems offer standardised problem-solving.
4. Always available.

Selecting a Problem for Expert System Development

It is appropriate to invest in developing an expert system if:

1. Benefits justify the cost and effort of building an ES.
2. The problem may not be solved using traditional computing methods.
3. The problem domain is well defined.
4. The problem may be solved using symbolic reasoning techniques and does not require commonsense reasoning.
5. The problem is of a realistic size and scope.
6. Experts exist who are co-operative and articulate.

Representation of Knowledge

– In AI, a REPRESENTATION OF KNOWLEDGE is a combination of data structures and interpretative procedures that, if used in the right way in a program, will lead to 'knowledgeable behaviour'
– The different expert knowledge representation approaches used in AI are as follows:
  – Logic
  – Procedural representation — e.g. in a LISP, C or ADA program
  – Semantic networks
  – Production systems (= rule-based systems)
  – Frames
  – Scripts

Unit 8. Computer Vision 1/2

Learning Outcomes. You should be able to

- define the goal of computer vision, name several application areas
- define the usual five stages of computer vision, briefly explain each
- list three reasons why computer vision is hard to implement, comparing it with the human vision system
- explain how thresholding works and what it is good for
- explain what smoothing is for
- apply a given filter mask
- describe edge detection process that uses a given difference operator
- describe and apply basic line finding procedure based on observing changes in line direction
- describe the main idea of the Hugh transform line finding procedure
- describe Guzman's procedure and apply it to simple scenes

- Computer vision is an intensely studied area of AI
- The ultimate goal of computer vision is to build systems that equal or exceed the capabilities of the human visual system. Ideally, a computer vision system would be capable of interpreting and describing any complex scene in complete detail.
- Some typical areas of application: screening of medical images, robotics, sorting picking and bin packing of items, terrestrial image mapping, weapons guidance, parts inspection for quality control, face and fingerprint identification.
- A typical computer vision system should be able to perform: image acquisition, image segmentation, shape interpretation and recognition.
- These activities are realised in stages as follows.

Stage 1: digitisation (getting a picture into the computer)

$$\text{an analog camera picture} \longrightarrow \text{brightness array}$$

- It requires a camera for image acquisition — two or more images are required for stereoscopic vision.
- The brightness array is a 2-D array whose elements describe **pixels** (picture elements). To obtain a brightness array from a picture or a scene, its elements are extracted from some form of light sensitive surface. A typical computer display screen can be viewed as a brightness array.

- The colour of each pixel is stored as a sequence of 1 or more bits. In the simplest case of monochrome pictures, only 1 bit is required to hold the colour information (0 normally corresponds to black, 1 to white). 8 bits will describe one of 256 colours.
- These sequences of bits encode numbers. In case of B&W pictures, these numbers correspond to the grey level intensity of each pixel. For colour images, the pixel's value typically comprises three separate numbers, one for the intensity value of each of the three basic colours: red, green and blue.
- The size of a brightness array is called its **resolution**.
- In order to give a quality comparable to the traditional TV, we need a resolution of at least 500x500. A typical resolution of a computer display screen is between 800x600 and 1600x1200.
- Required resolution always depends on the application. In some cases a lower resolution (30x50) may be acceptable whilst in other cases 1000x1000 is insufficient.

Stage 2: signal processing

$$\text{brightness array} \quad \longrightarrow \quad \text{better brightness array}$$

Low-level processing of the digital image in order to enhance significant features. E.g. smoothing of neighbouring points to reduce noise and thresholding.

Stage 3: edge and region detection

$$\text{brightness array} \quad \longrightarrow \quad \text{edge point description}$$

- Here, simple features are identified such as lines or edges in the image to produce an output, called the **primal sketch**, which is essentially a line drawing of the image.
- Edges or boundaries = sites of significant intensity or colour gradients (i.e. changes in colour or intensity).

Stage 4: object recognition

$$\text{edge point description} \longrightarrow \text{object shape representation}$$

- Fit line segments to edge points and identify closed regions.
- It involves connecting, filling in, and combining boundaries, determining regions and then assigning descriptive labels to objects that have been identified during this process.
- Objects will be identified as a line drawing with lines, regions and junctions.
- The output is sometimes referred to as the **2 1/2D sketch**.

Stage 5: image understanding

object shape representation $\longrightarrow$ spatial knowledge representation

- Making sufficient sense of the image to use it.
- Consists of identifying the *important* objects in the image and their relationships. (For example: object X is a ball, object Y is a goal, X is near Y.)
- Some systems may also require 3-D analysis and motion detection. (E.g. X is not yet really inside Y, X is moving towards Y.)

Human Vision

- very sophisticated and highly evolved.
- In terms of image acquisition, the eye is totally superior to any camera system yet developed.
- The retina contains 2 classes of discrete light receptors — **rods** and **cones**.
- The cones of which there are 6-7M in the eye are highly sensitive to colour. Each cone is connected by its own nerve to the brain.
- There are at least 75M rods in the eye distributed across the retina surface. They are sensitive to light intensity but not colour. They share nerve endings.
- The range of intensities to which the eye can adapt is of the order of $10^{10} = 10$ billion from the lowest visible light to the highest bearable glare.

Problems of Computer Vision

Vision is a challenge for AI because:

1. The world is 3-D (i.e., as we imagine it, not in reality) and the images from which a description must be formed are only 2-D projections.
2. Technical problems encountered in getting a "clean" image.
3. large amount of data to process

Low level processing (stages 2 and 3)

- A raw digitised image will contain some noise and distortion.
- Low level processing will often require local smoothing of the array to eliminate this noise.
- Other low level operations include threshold processing to help define homogeneous regions and different forms of edge detection to help define boundaries.

Thresholding

- used to sharpen object regions by enhancing some portions and reducing others (e.g. noise and unwanted features).
- It is the process of transforming a grey-level representation to a binary representation of the image.
- All digitised array values above some threshold T are set equal to the maximum grey level value (black) otherwise set equal to zero (white).
- Selecting an appropriate threshold level settings can be done by producing a histogram of the image grey-level intensities. This is then analysed to determine where concentrations of different intensity levels occur. A value of T is chosen at which the histogram shows a clear separation between intensity levels.

Smoothing

- used to reduce noise and other unwanted features.
- Various techniques have been employed.
- One common method of smoothing is to replace each pixel in the array with a weighted average of the pixel and its neighbouring values. This can be accomplished with the use of **filter masks**.
- Two typical masks are shown below — one using 4 neighbourhood pixels, the other 8, where the central pixel is the one to be processed.

	1/8	
1/8	**1/2**	1/8
	1/8	

1/32	3/32	1/32
3/32	**1/2**	3/32
1/32	3/32	1/32

For example, a block of pixels having the values shown below

1	1	0
1	1	0
1	1	0

would, when applying the latter filter, give a value for the central pixel of:

$$\begin{aligned} &1 \times \tfrac{1}{32} + 1 \times \tfrac{3}{32} + 0 \times \tfrac{1}{32} + \\ &1 \times \tfrac{3}{32} + 1 \times \tfrac{1}{2} + 0 \times \tfrac{3}{32} + \\ &1 \times \tfrac{1}{32} + 1 \times \tfrac{3}{32} + 0 \times \tfrac{1}{32} \\ &= \tfrac{27}{32} \end{aligned}$$

which would be assigned the value 1 (i.e. black)

- This calculation can be repeated for almost all the remaining pixels. Note that the technique cannot be applied at the boundaries and so the smoothed image is smaller than the original by the width of the smoothing filter.
- Applying a mask to an image array has the effect of reducing spurious noise as well as sharp boundaries.
- The sliding and summing operations are called **convolution**.

Edge detection

- Edges of objects can be found by looking for places where the intensity at nearby pixels is significantly different.
- At the simplest level, we could measure the difference between two adjacent pixels in the image array and mark an edge if the difference exceeds some threshold.
- A better technique is to use difference operators which are small masks (2x2 or 3x3 arrays) that are placed over groups of points in the image. The difference operation now involves taking each mask value multiplying it by the corresponding image intensity value and summing the result. E.g. if we have a mask

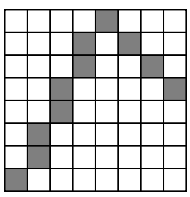
$$\begin{matrix} -1 & +1 \\ -1 & +1 \end{matrix} \quad \text{operating on} \quad \begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix}$$

  then the result is: $+1 \times 2 + 1 \times 6 - 1 \times 1 - 1 \times 5 = 8 - 6 = 2$. We can conclude there is an edge if the magnitude of the resulting number is greater than some threshold.
- A variety of difference operators have been proposed in the literature, e.g. **Sobel operator** listed below for the horizontal (x) and vertical (y) directions. Some operators, including the Sobel operator, combine finding differences with smoothing.

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad \qquad \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

Line Finding

- Several techniques have been used here.
- A simple approach is to follow edge pixels noting when there are dramatic changes in direction.
- For example, consider the following array:



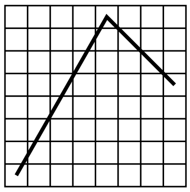
– Let the angle between adjacent pixels be measured in a clockwise direction from the vertical and then starting from the lower left hand corner we obtain the following angle changes:

$$45 \quad 0 \quad 45 \quad 0 \quad 45 \quad 0 \quad 45 \quad 135 \quad 135 \quad 135$$

and the calculated angle differences between the above being:

$$-45 \quad 45 \quad -45 \quad 45 \quad -45 \quad 45 \quad 90 \quad 0 \quad 0$$

– If we now apply a threshold of 90° to the above angle differences, then we find that the topmost pixel corresponds to a junction, that is, a point at which 2 or more lines meet. When the junctions have been found, we can approximate the two lines as shown below:



– This is a very simple approach which has limitations: in practice, the threshold angle is dynamically calculated in terms of the average angle between neighbouring pixels.

Hough Transform

– Another way to find lines is to start with an edge point and then travel along, looking for other connected edge points. This unfortunately does not work well in practice because there are usually breaks in the line.

– A better approach for line finding is the **Hough transform**.

– The idea in the Hough transform is to consider equations of line or curved segments. Thus, if we are looking for straight lines, we know that they can be represented by the equation $y = mx + c$ for some coefficients m and c.

– Starting with an edge point, we consider all possible straight lines (i.e. all values of m and c in suitable discrete steps) and then check for each resulting line how many points fall on it. We then assume that lines containing the most points are actual lines in the image.

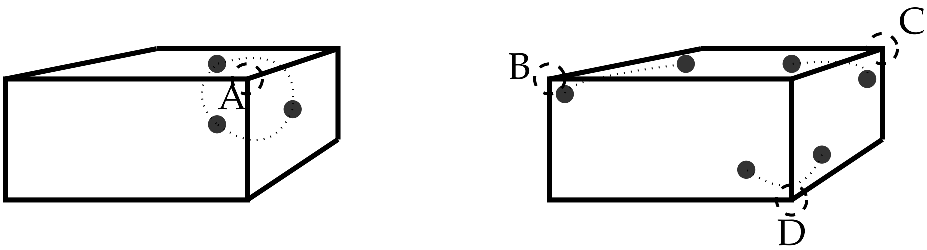
– This can be repeated for curves where the equation is:

$$y = ax^3 + bx^2 + cx + d$$

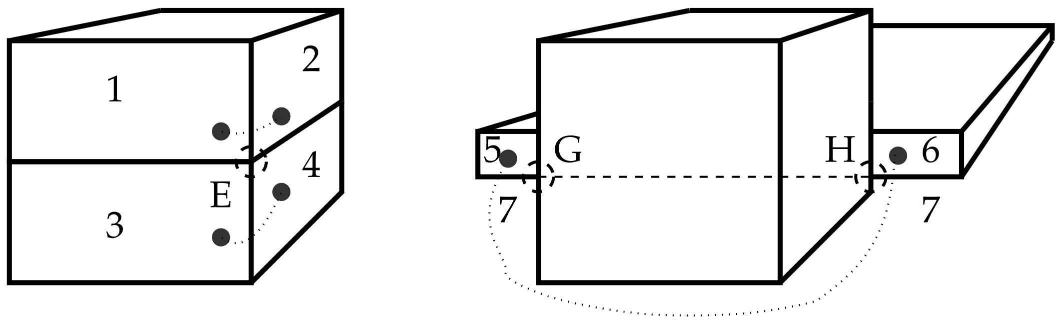
with various combinations of a, b, c and d.

Guzman 1968

adopted an **empirical** approach to *scene segmentation* (stage 4). Consider the following simple scene:

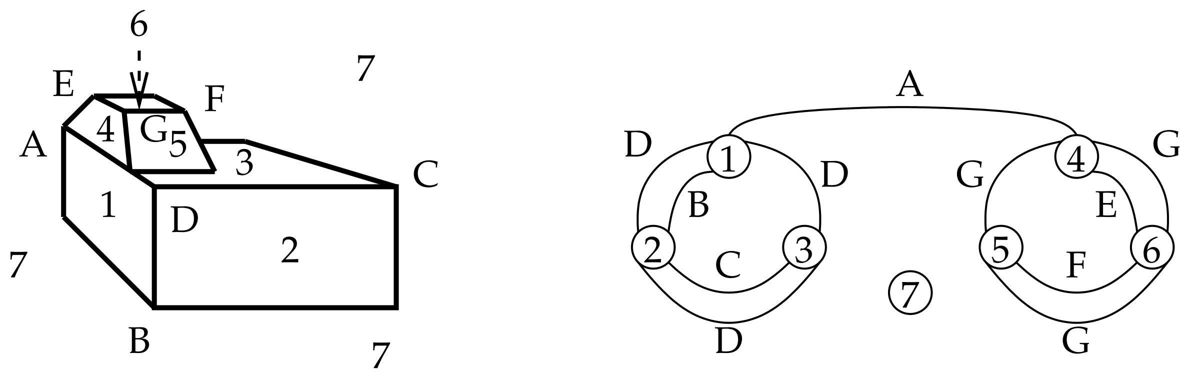


1. A Y-junction (i.e. one with 3 edges resembling the letter "Y", e.g. point A in the scene above) gives evidence that the three regions meeting at the junction should be grouped together.
2. A W-junction (e.g. the points B, C, D in the scene above) gives evidence that the two regions included between the narrow angles of the "W" should be grouped together.

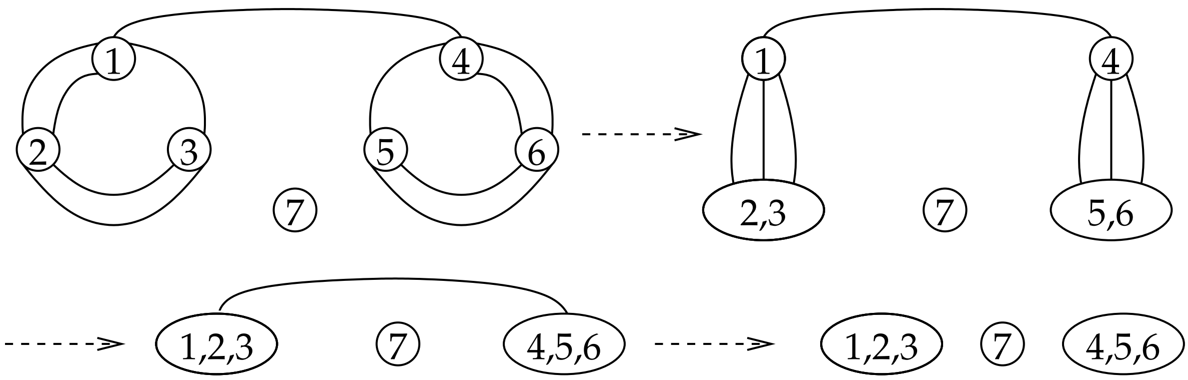


3. An X-juction (e.g. point E in the scene above) gives evidence for grouping together the regions on each side of the 'straight-through' line of the "X". I.e. group together regions 1 and 2 and regions 3 and 4 in the left-hand-side scene above.
4. Two collinear T-junctions (e.g. points G, H in the scene above) give evidence for grouping together the regions on the same side of the stem of the "T"s (e.g. region 5 is grouped with region 6).

   Below is another scene and a graph showing all grouping evidence links derived using the above rules:



5. Two nodes (i.e. regions) are to be grouped together if there are at least 2 links between them.



Unit 9. Computer Vision 2/2

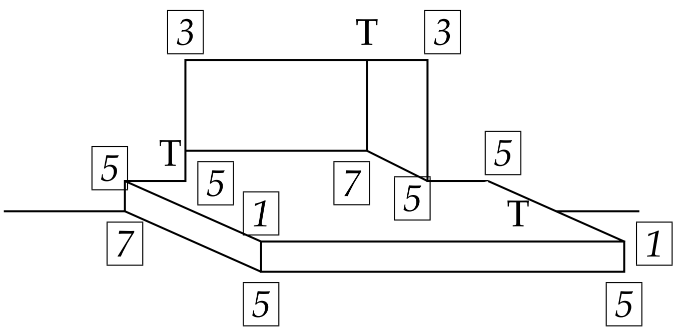
Learning Outcomes. You should be able to

- describe and recognise the four types of polyhedron vertices considered by Huffman in his work on Impossible Solid Objects
- list Huffman's labels used for edges in a line drawing and define their meaning
- correctly label (using Huffman's approach) the edges in a given line drawing of a scene with polyhedra
- do the previous as a state-space search, thus being able to find all possible labellings or prove that there is no correct labelling for a given drawing
- recognise vertices and points whose outgoing edges are labelled in a way which cannot be achieved in a real scenario
- list three ways in which Waltz extended Huffman's labelling
- give at least one reason why Waltz's algorithm is fast

Huffman's work on Impossible Solid Objects

D A Huffman *'Impossible Objects as Nonsense Sentences'*
Machine Intelligence 1971, vol 6, 295-325.

- Huffman considered the analysis of scenes containing solid polyhedrons.
- The solid polyhedrons were assumed to have exactly 3 planar surfaces at each of the vertices and 2 surfaces associated with each edge.
- There are 4 basic ways in which 3 plane surfaces can come together at a vertex. We call them **vertex types** and use the numbers $\boxed{1}$, $\boxed{3}$, $\boxed{5}$, $\boxed{7}$ to identify them. All four can be illustrated in the picture of a fireplace and raised hearth shown in Fig. below.



One by one, the 4 vertex types are illustrated in Figure 8.

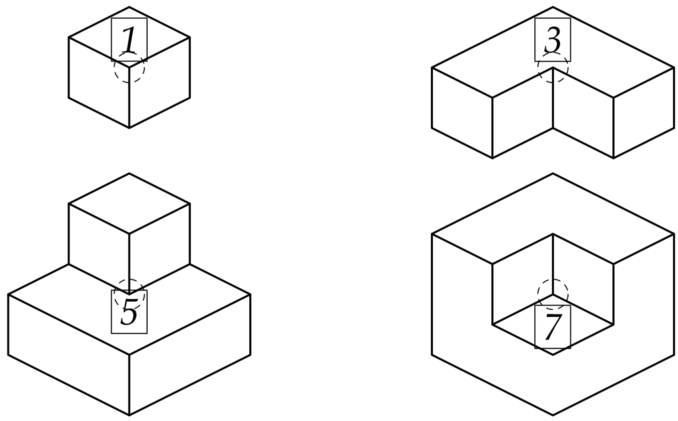


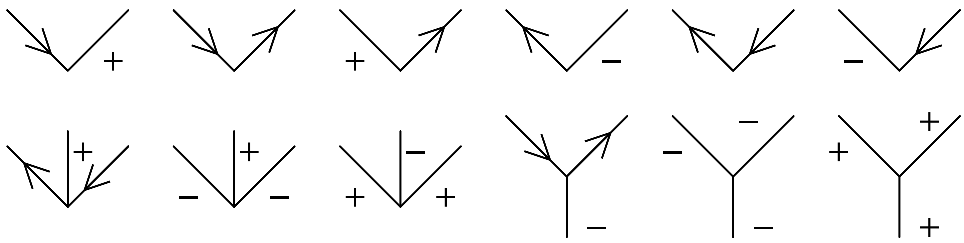
Figure 8: Huffman's four vertex types

– By analysis of the above four scenes and with the following definitions

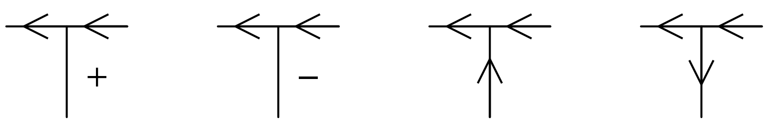
+	convex edge
–	concave edge
$\longrightarrow$	looking in the direction of the arrow the face to the right is visible

we obtain the scenes shown in Figure 9 listing all the possible ways of viewing the given vertices.

– From this, we finally derive the following **catalogue** of all possible representations of the vertices of trihedral solids:



– Moreover, at each T-junction in a scene, the three edges normally have one of the following labellings:



– Figure 10 shows a search for a valid labelling in an example scene. The junctions in the image are processed one by one in the order A, B, C,... For each junction, the catalogue is looked up to find all possible labellings of its neighbouring edges. Some of these edges might have already been labelled by the other junctions. Only the labellings of the new junction consistent with the previous labelling are considered. For each of the possible consistent labellings, a new branch in the search tree is created. If there is no consistent labelling, the previous labelling was wrong and the current branch is terminated.

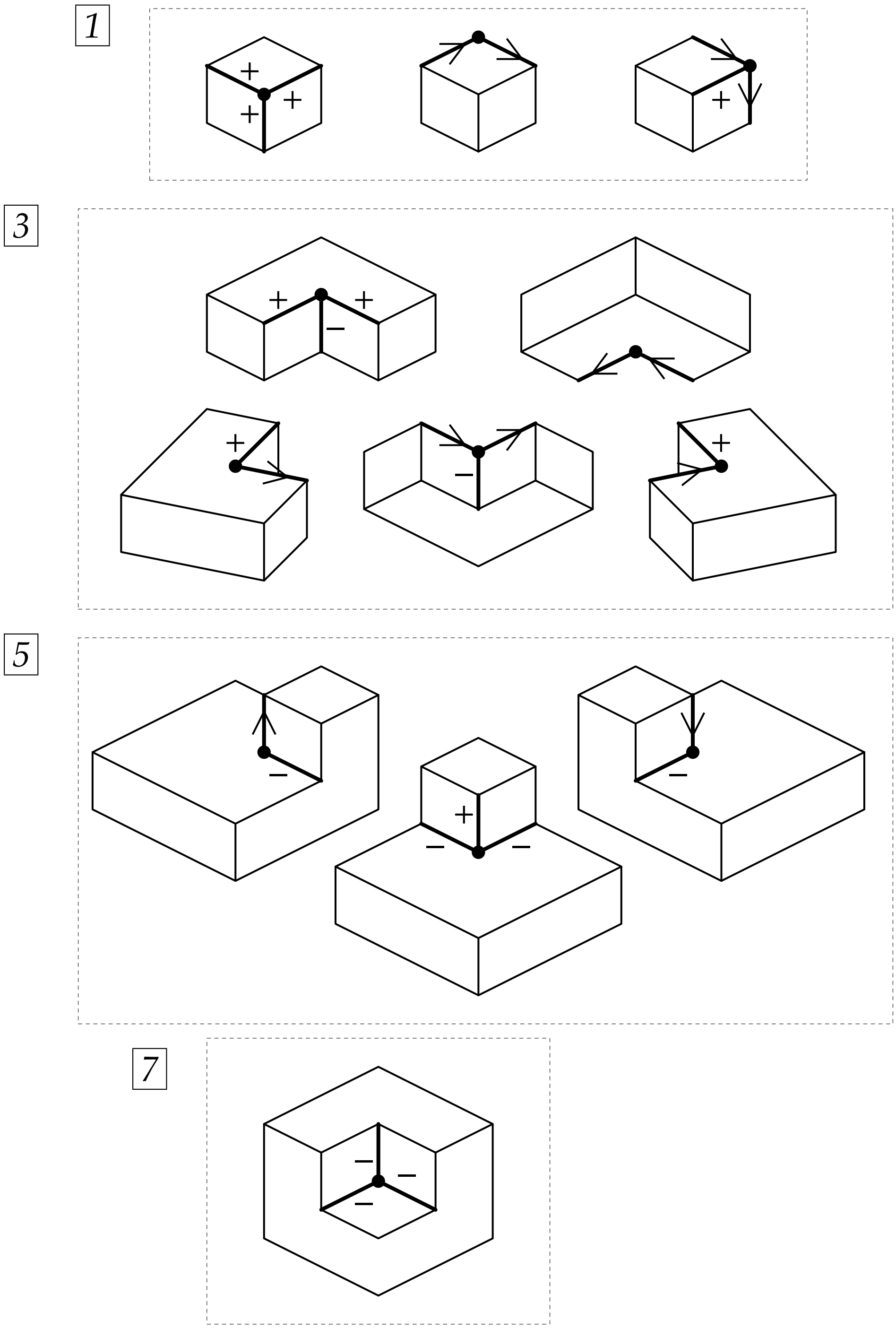


Figure 9: Huffman's four vertex types viewed from various angles as Y, W and L vertices with edge type annotations.

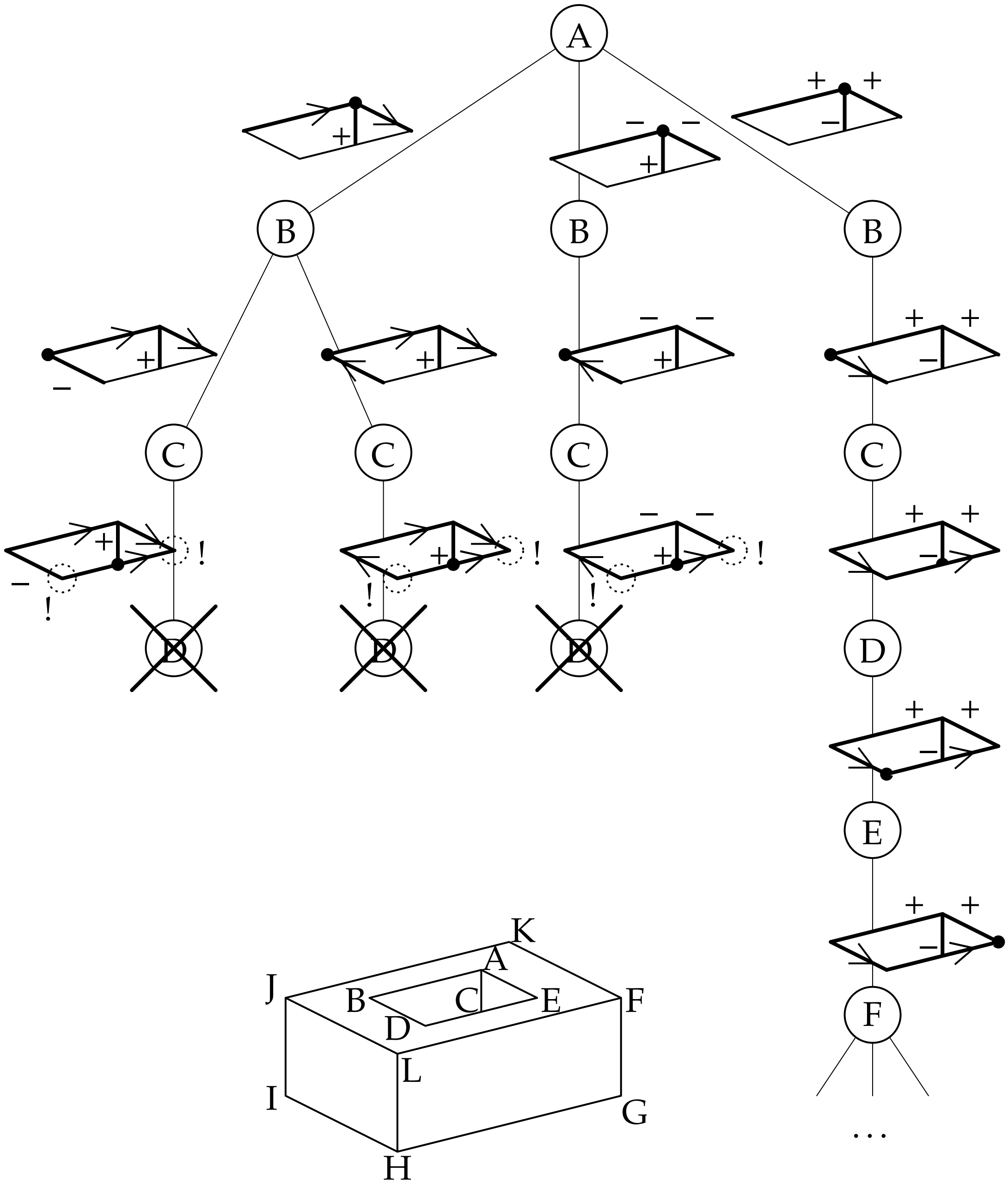
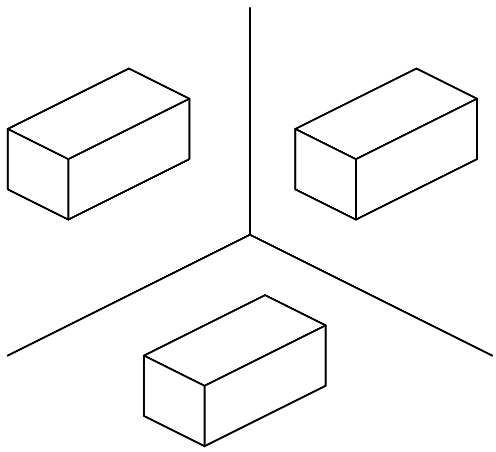


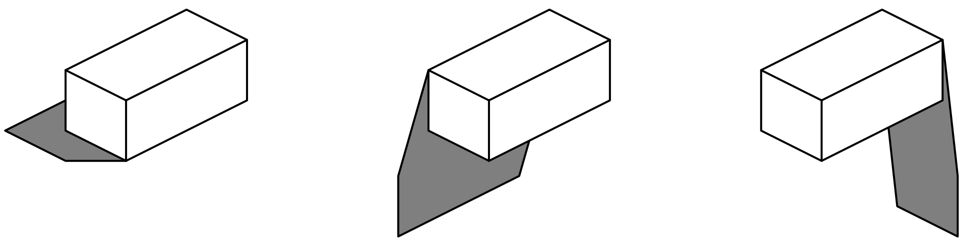
Figure 10: Application of Huffman's method on an example scene.

Waltz

- builds upon the work of Huffman/Clowes and demonstrates that the physical world severely constrains the way links and vertices can fit together in line drawings. E.g. a constraint imposed by Huffman's work and carried through in Waltz's work is that a line cannot change its nature from one end to the other.
- Huffman analysed drawings as if they were suspended in isolation and hence no shadows are present in the drawings. Without shadows there are several ways of interpreting a cube. For example, each of the following cubes looks exactly the same, even though one is resting on the floor, one is attached to the right-hand wall and one is attached to the left-hand wall:

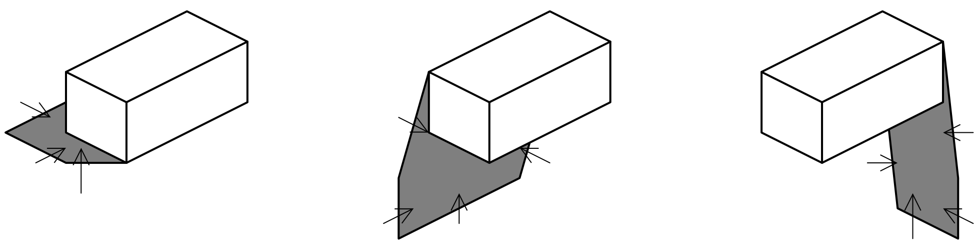


- Waltz's program introduces shadows and this resolves the ambiguity, e.g.:

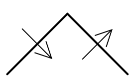


block on table block on LH wall block on RH wall

- Waltz's program expanded the Clowes-Huffman theory by introducing labels for shadows and introducing appropriate constraints for their interaction with other line labels. The shadow edges are labelled by a perpendicular arrow:



- The arrow of the shadow line always points into the darker area. The number of the possible interpretations is limited, e.g. the following would be impossible:



– The theory was also extended to include 'cracks' which were denoted by **C** labels.

– Altogether, Waltz recognised eleven different line types.

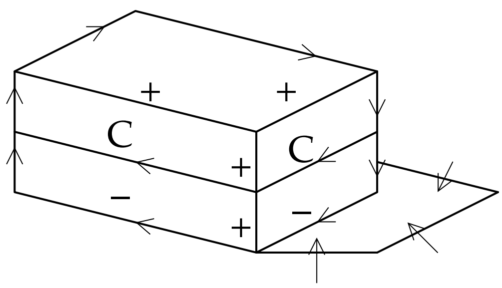
– It seems reasonable to assume that such forbidden combinations might be rare, but in fact the reverse is the case: The rarity of valid labellings is illustrated by the table of possible and physically realisable labellings for the simplest of the possible vertex types:

Junction type	**Approx. No of possible labellings**	**Physically possible labellings**
L	>1000	92
W	>35,000	86
Y	>35,000	826
T	>35,000	623

– This totally unexpected result has become known as '**The Waltz Effect**'.

– Although the number of possible labellings is relatively small, it is still too large to make searching feasible. Waltz's program uses **elimination**, i.e. it assigns all possible labellings to each of the nodes in isolation, it then considers adjacent nodes and eliminates labellings not consistent with those on the adjacent nodes. This is iteratively performed until a single labelling is left at each vertex.

– A typical result of an analysis by Waltz's program is shown below.



– Waltz's program works very well and an encouraging aspect is that analysis time increases **linearly** with scene complexity.

Final Note

Recognising and finding the precise position of objects in practice is a challenging problem: objects may be very complex, they may be occluded, shiny and appear in different orientations.

Unit 10. Logic 1/2

Learning Outcomes. You should be able to

- recognise examples of well-formed propositional formulas from badly-formed ones, describe how to do it in general
- translate propositional formulas into plain English
- translate suitable English statements into propositional formulas
- write a truth table for a given propositional formula, explain its results
- describe and apply the propositional resolution algorithm to prove or disprove a theorem

Logic — the formal treatment of knowledge and thought, as developed in philosophy, has been applied to the development of computer programs that can reason.

Logic has two important and inter-locking branches:

1. *language* — what can be said — what relations and implications one can formalise
2. *deductive structure* — the rules of inference that determine what can be inferred if certain axioms are taken to be true.

Propositional Calculus

– is the simplest common form of logic.

– A properly formed statement, or **proposition**, has one of two possible truth values: **true** or **false**.

– Typical propositions are:

It is raining.
Five plus three equals nine.

– Note that each of the sentences is a proposition, **NOT** to be broken down into its constituent parts.

– The symbols of propositional calculus are:

the propositional symbols:	$P, Q, R, S, \ldots$
the truth symbols:	*TRUE*, *FALSE*
and the connectives:	$AND = \wedge$
	$OR = \vee$
	$NOT = \neg$
	$IMPLIES = \Rightarrow$
	$EQUIVALENT = \Leftrightarrow$

– Propositions on their own are not particularly interesting. However, they can be combined using the connectives shown above, for example:

The book is on the table or it is on the chair.
If Socrates is a man, then he is mortal.

– A precedence rule is applied to the connectives, which may be overridden by the use of parentheses.

– The precedence in **decreasing** priority is:

1. *NOT* ($\neg$)
2. *AND* ($\wedge$), *OR* ($\vee$)
3. *IMPLIES* ($\Rightarrow$), *EQUIVALENT* ($\Leftrightarrow$)

– Note: the symbols ( ) have the highest priority and are used to group symbols into subexpressions and so control their order of evaluation and meaning.

– If X and Y are any two propositions, the connectives have the following meaning:

$X \wedge Y$	is *TRUE* if both X and Y are *TRUE*, otherwise *FALSE*.
$X \vee Y$	is *TRUE* if either, or both, X or Y is *TRUE*, otherwise *FALSE*.
$\neg X$	is *TRUE* if X is *FALSE*, and is *FALSE* if X is *TRUE*.
$X \Rightarrow Y$	is *FALSE* only when X is true and the value of Y is false, otherwise it is always *TRUE*.
$X \Leftrightarrow Y$	is *TRUE* only when X and Y have the same truth values, otherwise it is *FALSE*.

– Formulas in propositional logic can be constructed from syntactic combinations of variables and connectives:

$$(X \Rightarrow (Y \wedge Z)) \Leftrightarrow ((X \Rightarrow Y) \wedge (X \Rightarrow Z))$$

"X implies Y and Z is the same as saying that X implies Y and X implies Z"

– Every propositional symbol on its own is a formula, although tiny. Also *TRUE* and *FALSE* on their own are formulas. If P and Q are any formulas, then the following are also formulas:

$$\neg P \qquad P \wedge Q \qquad Q \vee P \qquad P \Rightarrow Q \qquad P \Leftrightarrow Q$$

– Formulas are also called **well-formed formulas** or WFFs ('wuffers') in contrast to "badly-formed formulas", i.e. sequences of symbols which do not conform to the rules above and are no formulas at all even if they may look similar to a formula. (E.g. $Z \vee Y\neg$ and $\Rightarrow X$ are not formulas.)

– Only expressions that are formed of legal symbols through some proper sequence of the connectives are well-formed formulas, for example:

$$((P \wedge Q) \Rightarrow R) \Leftrightarrow \neg P \vee \neg Q \vee R$$

is a well-formed formula in the propositional calculus since:

P,Q and R are propositions and thus formulas.

$P \wedge Q$, the conjunction of two formulas, is a formula.

$((P \wedge Q) \Rightarrow R)$, the implication of a formula for another, is a formula.

$\neg P$ and $\neg Q$, the negations of formulas, are formulas.

$\neg P \vee \neg Q$, the disjunction of two formulas, is a formula.

$\neg P \vee \neg Q \vee R$, the disjunction of two formulas, is a formula.

$((P \wedge Q) \Rightarrow R) \Leftrightarrow \neg P \vee \neg Q \vee R$, the equivalence of two formulas, is also a formula.

Rules of Inference

– An inference rule allows the deduction of a new formula from previously given formulas.

– The best-known inference rule is **Modus Ponens**, which states that if we know that two formulas of the form X and $X \Rightarrow Y$ are true, then we can infer that the formula Y is true.

– Using the plain English translation of propositional calculus: if we know that

"John is an uncle" is true
and "If John is an uncle, then John is male" is true

then we can conclude that:

"John is male" is true.

– Note that the Modus Ponens rule allows us to replace the 2 entries X and $X \Rightarrow Y$ with the single statement Y, thus eliminating one occurrence of the connective $\Rightarrow$.

– The Modus Ponens rule can be expressed in the propositional calculus itself as follows:

$$(X \wedge (X \Rightarrow Y)) \Rightarrow Y$$

Proving theorems in Propositional Calculus

There are two main methods for proving theorems in propositional calculus: truth tables and formal deductions.

Truth Tables

– This method evaluates the truth table of the formula for all possible combinations of truth values assigned to proposition variables. Each combination takes one row in the table. Columns correspond to the individual variables and all sub-formulas of the formula to be proved. The truth values are represented by letters T and F.

- First, the values of the smallest sub-formulas are filled in and then the values of the larger sub-formulas. Finally, the value of the whole formula is established. If the formula gets T in all rows, then it is proved true.
- For example, the truth table shown below demonstrates the equivalence of $P \Rightarrow Q$ and $\neg P \vee Q$:

proposition variables		sub-formulas			
P	Q	$\neg P$	$\neg P \vee Q$	$P \Rightarrow Q$	$(\neg P \vee Q) \Leftrightarrow (P \Rightarrow Q)$
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

- Although this method can clearly be programmed, it is inefficient: It requires 2^N rows in the truth table for an N-variable problem. (E.g. 4 rows for 2 variables, 1024 rows for 10 variables)
- The following propositional equivalences can be proved using truth tables. These expressions represent commonly used identities in the propositional calculus.

$$\neg(\neg P) \Leftrightarrow P$$
$$(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$$
$$\neg(P \vee Q) \Leftrightarrow (\neg P \wedge \neg Q)$$
$$\neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$$

The bottom two are called **de Morgans's laws**.

Propositional Resolution

- The other general approach attempts to prove theorems by following a set of rules of inference. Different methods of inference are possible. The most popular is that known as the **resolution** method.
- Resolution is a syntactic method of proof (deduction).
- To express the resolution method a few additional concepts need to be defined.
- Two formulas are said to be **equivalent** if whenever their corresponding variables are given the same values, they always have the same truth values. E.g.

$$\neg(P \wedge Q) \qquad \text{and} \qquad \neg P \vee \neg Q$$

are equivalent expressions as can be seen by generating the truth tables.
- A **clause** is a propositional formula of a special kind: it has to be a sequence of proposition variables with or without negation ($\neg$) joined by *OR* ($\vee$). For example, the following three are clauses:

$$P \vee Q \vee \neg R \qquad S \qquad \neg P \vee S$$

In addition, the proposition *FALSE* is also a clause.

- *FALSE* could be viewed as an *empty sequence* of proposition variables joined by *OR*. This empty clause is sometimes written Π. The clause Π is never true, in the same way as "white equals black" is a proposition which is never true.
- A formula is said to be in **conjunctive normal form** (**CNF**) if it is a sequence of clauses joined by *AND* ($\wedge$). For example, the following is a formula in CNF:

$$(P \vee Q \vee \neg R) \wedge S \wedge (\neg P \vee S)$$

  and $\neg(P \wedge Q)$ is not in CNF.
- An alternative description of a formula in CNF:
  1. It is composed solely of variables and the four symbols: $\neg, \vee, \wedge$ and *FALSE*, i.e. there are no $\Rightarrow$, $\Leftrightarrow$ and *TRUE*.
  2. Negation symbols ($\neg$) apply only to variables, never to bracketed expressions.
  3. "Or" symbols ($\vee$) apply only to variables or negated variables, never to bracketed expressions containing "and" symbols ($\wedge$).
- Why do we define CNF? CNF formulas are in a special, much simpler, form. Therefore, it is easier to work with formulas in CNF than with arbitrary propositional formulas. *Most importantly, any propositional formula can be simplified to an equivalent formula which is in CNF.*
- In propositional resolution we prove a theorem T as follows:
  - Assume that the theorem T is false, and therefore its negation $\neg T$ is true.
  - Show that the assumption together with the premises leads to an impossible situation.
  - Therefore, the negation of the theorem cannot be true. Thus, the theorem must be true.
- The "impossible situation" is called a **contradiction** and is expressed by the empty clause Π.
- In more detail, the proof by resolution works as follows:
  1. Replace all the premises and the negation of the theorem by their equivalent CNF formulas. All clauses are placed in a single group.
  2. Two clauses are selected in such a way that one contains some variable V and the other its negation $\neg V$.
  3. A new clause, the **resolvent**, is formed from all the $\vee$-ed elements of the two clauses, except for V and $\neg V$. and any other variables present both in their positive and negated forms.
  4. The resolvent is added to the group.
  5. The process in steps 2–4 is repeated until either
     - a contradiction (i.e. the empty clause Π) is deduced thus proving the theorem
     - no more clauses can be added by resolution, thus the theorem is not proved (and is not true!)

Example 1

– Consider the following situation:

Premises:	If I am in an AI lecture, then I feel sleepy.
	I don't feel sleepy.
Theorem:	I am not in an AI lecture.

– If we represent 'If I am in an AI lecture' by L
'I feel sleepy' by S
then the two premises are $(L \Rightarrow S)$, $\neg S$ and the theorem is $\neg L$.

– Applying resolution, we translate the two premises to CNF getting $(\neg L \vee S)$ and $\neg S$ which consist of one clause each. The negation of the theorem in CNF is simply L. Put these together we get the following group of three clauses:

$$\neg L \vee S \qquad \neg S \qquad L$$

Now, we follow step 2 of the resolution algorithm above. We can choose, for example, the first two clauses because one contains S and the other $\neg S$. Then we add the resolvent which is $\neg L$. The group of clauses now consists of:

$$\neg L \vee S \qquad \neg S \qquad L \qquad \neg L$$

Going back to step 2, we pick these two clauses, resolve them and add the resolvent, i.e. the empty clause Π—the desired contradiction. We proved that L contradicts the premises $(L \Rightarrow S)$ and $\neg S$. Therefore, the theorem $\neg L$ follows from the premises.

Example 2

– Consider the following problem:

If the butler was in the drawing room then he was at home,
if the butler was at home then he must have heard the murderer,
if the butler told the police the truth then he did not hear the murderer,
therefore, if the butler was in the drawing room then he lied to the police.

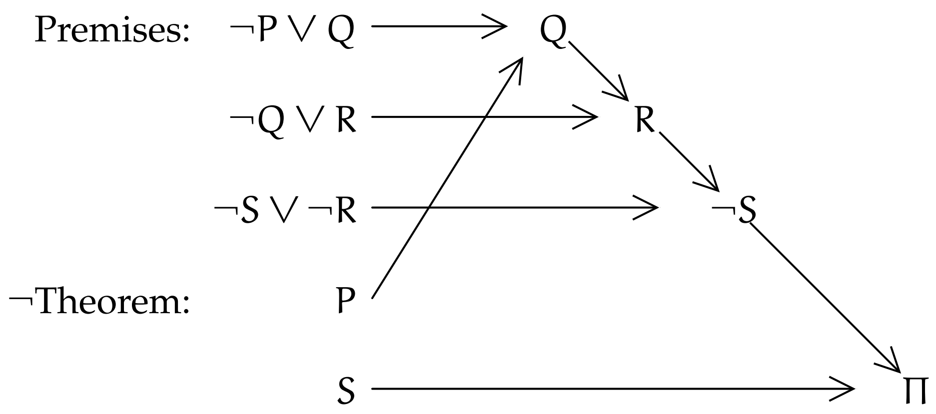
– We can define the following propositional variables with the following meanings:

P:	the butler was in the drawing room
Q:	the butler was at home
R:	the butler heard the murderer
S:	the butler told the truth to the police

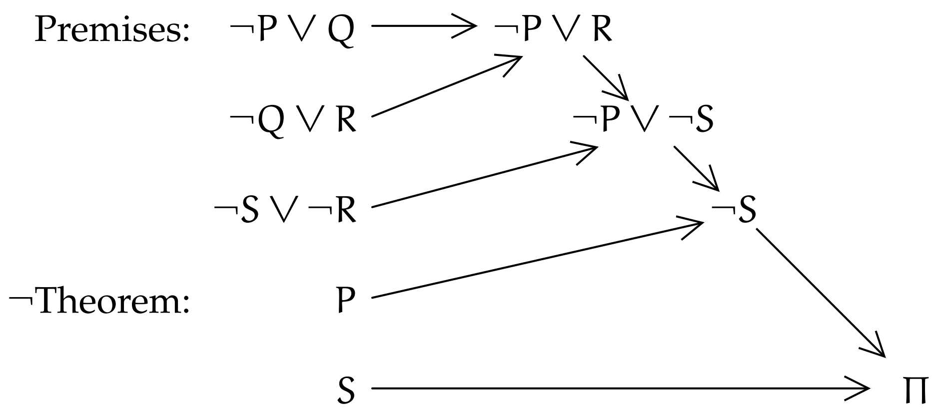
– Now we may translate each of the above statements into the following propositions and convert them to CNF:

Premises:	$(P \Rightarrow Q)$	$\neg P \vee Q$
	$(Q \Rightarrow R)$	$\neg Q \vee R$
	$(S \Rightarrow \neg R)$	$\neg S \vee \neg R$
Theorem:	$(P \Rightarrow \neg S)$	$\neg P \vee \neg S$
Negated Theorem:	$\neg(P \Rightarrow \neg S)$	$P \wedge S$

– Now we apply the resolution algorithm to disprove the negated theorem as follows:



– If we chose the pairs to be resolved differently, we obtain a different proof of the same theorem:



Unit 11. Logic 2/2

```
Learning Outcomes.  You should be able to
 - describe how propositional and predicate calculi differ
 - recognise well-formed predicate calculus formulas
 - translate predicate formulas to English and vice versa
```

Inadequacies of Propositional Calculus

– It should be possible in any logical system to refer to objects, to postulate relationships between those objects and to generalise the relationships over classes of objects.
– Propositional calculus does not allow us to do that, it is too 'coarse' to easily describe properties of objects, and it lacks the structure to express relations that exist among two or more entities — these limitations are overcome in predicate calculus.

Predicate Calculus

– is an extension of the notion of propositional calculus. The meaning of the connectives is retained but the focus of logic is changed: instead of considering sentences that are of interest merely for their truth value, predicate calculus is used to represent statements about **specific objects** or **individuals**.
– We shall first of all consider the syntax of the predicate calculus language and then discuss its semantics (= meaning).

The Syntax of Predicate Calculus

– There are the five **connectives** in predicate calculus. They are the same ones as in the propositional calculus (i.e. $\neg \wedge \vee \Rightarrow \Leftrightarrow$).
– Any formula which is syntactically correct in propositional calculus is syntactically correct in predicate calculus.
– The proposition in predicate calculus is extended to allow a predicate and its arguments to replace the simple variable. In the following example, **human** is the predicate and *socrates* is the argument. More generally, a variable X is the argument:

$$\mathbf{human}(\mathit{socrates})$$
$$\mathbf{human}(\mathrm{X})$$

– Here we adopt the scheme whereby constants will start with a lower-case letter and variables will start with an upper-case letter. Other examples:

$\mathbf{likes}(\mathit{john}, \mathit{jill})$	John likes Jill.
$\mathbf{likes}(\mathit{jill}, \mathit{john})$	Jill likes John.
$\neg\mathbf{likes}(\mathit{david}, \mathit{john})$	David does not like John.
$\mathbf{likes}(\mathit{beth}, \mathit{john}) \wedge \mathbf{likes}(\mathit{beth}, \mathit{david})$	Beth likes John and David.

- **Constants** are fixed-value terms. They name specific objects or properties in the world. Constant names begin with a lower-case letter, thus *michael*, *tree*, *tall* and *blue* are examples of well-formed constant names. The constants *true* and *false* are reserved as truth symbols.
- **Variables** are terms that can take different values, they are used to designate general classes of objects or properties in the world. They are represented by names beginning with an upper-case letter.
- **Predicates** name a relationship between zero or more objects in the world. Predicate names are written here in lower-case letters. Examples of which are:

  likes **equals** **on** **near** **part_of**

- A predicate with no arguments (e.g. **i_am_sleepy**) is equivalent to a propositional variable in propositional calculus (e.g. S).
- **Functions** denote a mapping of one or more elements in a set (called the domain of the function) into a unique element of another set (the range of the function). In addition to common arithmetic functions such as addition and multiplication, functions may define mappings between non-numeric domains. Function names (like constants) begin with a lowercase letter.
- A function expression is a function name followed by its arguments. The arguments are elements from the domain of the function. The number of arguments is the **arity** of the function and the arguments are enclosed in parentheses and separated by commas. For example,

$$f(\mathrm{X}, \mathrm{Y}) \qquad \mathit{price}(\mathit{apple})$$

Quantification

- There are two quantifier symbols, described below together with their meaning:

  $\forall \mathrm{X}$ — the **universal quantifier** indicates that the formula is true for **all** values of the variable X.

  $\exists \mathrm{X}$ — the **existential quantifier** indicates that the formula is true for **some** (i.e. at least one) value(s) of X.

- Examples of the use of the above quantifiers are shown below:

$\forall \mathrm{X}(\mathbf{person}(\mathrm{X}) \wedge \mathbf{likes}(\mathrm{X}, \mathit{chocolate}))$	Everybody likes chocolate.
$\exists \mathrm{X}(\mathbf{person}(\mathrm{X}) \wedge \mathbf{likes}(\mathrm{X}, \mathit{brussels\_sprouts}))$	Some people like brussels sprouts.
$\forall \mathrm{X}(\mathbf{student}(\mathrm{X}) \Rightarrow \mathbf{poor}(\mathrm{X}))$	All students are poor.
$\exists \mathrm{X}(\mathbf{bird}(\mathrm{X}) \wedge \neg \mathbf{flies}(\mathrm{X}))$	There exists some bird that does not fly.
$\forall \mathrm{X}(\mathbf{roses}(\mathrm{X}) \wedge \mathbf{red}(\mathrm{X}) \Rightarrow \mathbf{likes}(\mathit{beth}, \mathrm{X}))$	Beth likes red roses.
$\forall \mathrm{X}(\mathbf{roses}(\mathrm{X}) \wedge \mathbf{red}(\mathrm{X}) \Rightarrow \neg \mathbf{likes}(\mathit{beth}, \mathrm{X}))$	Beth does not like red roses.

Validity and Satisfiability

– A predicate calculus formula is **valid** if it is true for *all* possible interpretations; otherwise it is **invalid**, for example, the following is valid:

$$\mathbf{P}(A) \vee \neg \mathbf{P}(A)$$

– A predicate calculus formula is said to be **satisfied** if and only if there exists *at least one* interpretation, called *model*, for which it is true. Otherwise, the formula is said to be **unsatisfiable**.

An example of a simple proof in the predicate calculus

"All humans are mortal. Socrates is human. Therefore Socrates is mortal."

Translating to the predicate calculus, we get:

Premises: 1. $\forall X(\mathbf{human}(X) \Rightarrow \mathbf{mortal}(X))$
2. $\mathbf{human}(\mathit{socrates})$

To prove: 3. $\mathbf{mortal}(\mathit{socrates})$

By substituting *socrates* for X in 1, we get

4. $\mathbf{human}(\mathit{socrates}) \Rightarrow \mathbf{mortal}(\mathit{socrates})$

then from formulas 2 and 4 by Modus Ponens we get finally that

$\mathbf{mortal}(\mathit{socrates})$

Since these rules have been applied in a mechanistic manner, it is possible to write computer programs to work out such proofs.

Resolution Theorem Proving

– Resolution is a technique for proving theorems in the predicate calculus. (It can be used for propositional calculus as a subset of predicate calculus — we learnt about this simplified resolution in the previous unit.)

– The resolution principle, introduced by J A Robinson (1964), proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known (have been assumed) to be true. It then uses the resolution rule of inference to show that this leads to a contradiction. Once the negated goal is shown to be inconsistent with the given set of axioms then it follows that the original goal must be true — and this proves the theorem.

– The following steps are involved in resolution theorem proving:

1. Put the premises or axioms into clause form.
2. Add the negation of what is to be proved, in clause form, to the set of axioms.
3. Resolve these clauses together, producing new clauses that logically follow from them.
4. Produce a contradiction by generating the empty clause.

5. The **substitutions** used to produce the empty clause are those under which the opposite of the negated goal is true.

– The conversion of any set of predicate calculus statements to conjunctive normal form involves a number of steps. We will not consider this any further.

Example of Resolution Theorem Proving

Suppose that we are told that both of the following formulas are true:

$$\mathbf{trunk}(\mathit{dumbo}) \tag{1}$$
$$\forall X(\mathbf{trunk}(X) \Rightarrow \mathbf{elephant}(X)) \tag{2}$$

and we are asked to prove that the following formula is true:

$$\mathbf{elephant}(\mathit{dumbo}) \tag{3}$$

then by substituting *dumbo* for X in (2) and making use of the equivalence of the following formulas (see Logic 1):

$$P \Rightarrow Q \qquad \text{and} \qquad \neg P \vee Q$$

we arrive at:

$$\neg\mathbf{trunk}(\mathit{dumbo}) \vee \mathbf{elephant}(\mathit{dumbo}) \tag{4}$$

Adding the negation of the formula to be proved (i.e. (3) above), we finally arrive at the following list of formulas:

$$\mathbf{trunk}(\mathit{dumbo}) \tag{1}$$
$$\neg\mathbf{trunk}(\mathit{dumbo}) \vee \mathbf{elephant}(\mathit{dumbo}) \tag{4}$$
$$\neg\mathbf{elephant}(\mathit{dumbo}) \tag{5}$$

Clearly, these resolve to the null clause showing that formula (3) above is true.

Logic Programming

– The most widely used logic programming language is **Prolog**.

– Each sentence in the program corresponds to a formula in predicate logic although the syntax differs.

– The following shows a simple Prolog program together with the attendant statements in logic

PROLOG	LOGIC
`father(john, fred).`	$\mathbf{father}(\mathit{john},\mathit{fred})$
`father(bill,john).`	$\mathbf{father}(\mathit{bill},\mathit{john})$
`grandfather(X,Y) :-`	$\forall X\forall Y\forall Z(\mathbf{father}(X,Z) \wedge \mathbf{father}(Z,Y))$
`  father(X,Z),`	$\Rightarrow \mathbf{grandfather}(X,Y)$
`  father(Z,Y).`	

Given the above program, we could then ask Prolog to prove that `grandfather(bill,fred)` was true. It should answer `yes`.

Unit 12. Planning

Learning Outcomes. You should be able to
- define planning and list at least two application areas
- list at least 4 problems of planning
- describe and read operator schemata and Steel's notation
- describe and apply the STRIPS planning algorithm
- explain, on an example, the problem of interacting subgoals and three approaches to solving it
- define the frame problem and explain it using an example

Planning is the problem of devising a series of actions that will lead to a desired goal starting from some initial state. Planning is a large and growing area of AI.

A **plan** is a sequence of actions which, if followed, will lead to the desired goal by changing the relations among objects.

Application Domains

Planners have been applied to a variety of application domains:

Domain	Planner
Robot Control	STRIPS
Simple Program Generator	HACKER
Experiment Planning in Molecular Genetics	MOLGEN
Naval Logistics	NONLIN
Voyager Spacecraft Mission Sequencing	DEVISER

Problems

Some of the problems planners have had to address:

1. Representation — correct notation for plans
2. Anticipation and correction of problems (e.g. protection violation)
3. Managing the search through the space of possible plans — many of the issues of heuristic search (e.g. A* algorithm) are appropriate here.
4. Translating the plans into actual actions in the world.
5. Monitoring the progress of a plan — execution monitoring.
6. Re-planning when things go wrong.
7. The recognition of what is, and what is not changed, by a particular operation — the so-called **frame problem**. This will be dealt with later.

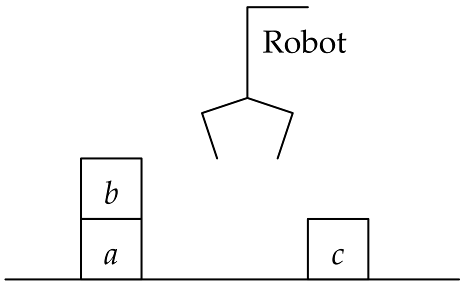
STRIPS

– Developed at Stanford University, it is an adaptation of Newell and Simon's GPS. It draws upon the following:

Means-end analysis	State-space search
Situational calculus	
Backward chaining	Use of add-lists and delete-lists

– Traditional planning systems have been introduced in terms of their performance in a *blocks world*.

– STRIPS represents a state of the "world" as a series of facts, expressed in predicate calculus e.g.



on_table(a) **on_table**(c) **clear**(b) **clear**(c)
stack(b, a) **arm_empty**

– STRIPS represents the actions it may choose to apply in its plans as operator schemata of the following general form:

name: operator identifier
preconditions: facts that must be true before an operator can be applied
effects: how the current set of facts should change on application of the operator

thus for the problem above we might have the following operators:

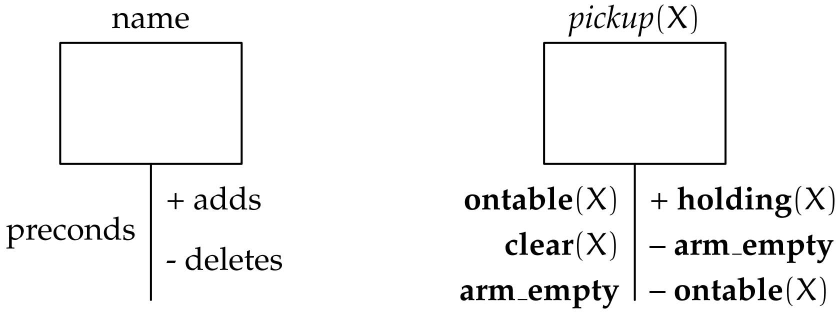
name: *pickup*(X)
preconditions: **on_table**(X), **clear**(X), **arm_empty**
deletes: **arm_empty**, **on_table**(X)
adds: **holding**(X)

name: *put_down*(X)
preconditions: **holding**(X)
deletes: **holding**(X)
adds: **arm_empty**, **on_table**(X)

name: *stack*(X, Y)
preconditions: **clear**(Y), **holding**(X)
deletes: **clear**(Y), **holding**(X)
adds: **on**(X, Y), **arm_empty**

name: *unstack*(X, Y)
preconditions: **on**(X, Y), **clear**(X), **arm_empty**
deletes: **on**(X, Y), **arm_empty**
adds: **holding**(X), **clear**(Y)

This can be represented graphically using a modified form of Steel's notation as:



– [Once the problem description is given, a possible approach to planning is to conduct a breadth-first search trying every operator for which the preconditions are true. However, this is a poor strategy because it battles against exponential tree growth.]

STRIPS planning algorithm

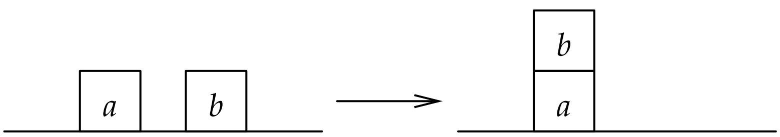
Given an initial world description S and a goal specification G_1, STRIPS proceeds to successively reduce the set of differences between S and G_1:

$S \longrightarrow G_1$	plan actions to achieve G_1
$S \longrightarrow G_2 \longrightarrow G_1$	...achieve G_2 then achieve G_1 by O_1
etc.	
$S \longrightarrow G_n \longrightarrow \cdots \longrightarrow G_3 \longrightarrow G_2 \longrightarrow G_1$	

If G_1 is not true in the current state of the world, STRIPS attempts to find an operator (O_1) which will make it true. Before this can be done, the preconditions necessary before the operator O_1 can be applied must be true. To solve (possibly) the preconditions of the operator O_1, STRIPS simply calls itself recursively.

This technique is known as **Goal Reduction**.

This search process is goal directed and proceeds by a process of backward chaining so that the operators are selected in reverse order to their intended application. To implement it, STRIPS maintains a stack of goals which it must achieve. Consider the problem:



In the beginning, on the goal stack we have only **on**(b, a). This is not true in the current state of the world:

on_table(a) **on_table**(b) **clear**(a) **clear**(b)

So STRIPS determines from its table of operators whether one of the operators, when applied, could achieve the goal. It finds that the operator *stack*(X, Y) has **on**(X, Y) in its add-list, so the planner starts to build a plan which ends with the action *stack*(*b*, *a*).

Instantiating the variables in the operator schema for *stack*(X, Y) gives the following preconditions which are pushed onto the stack, and which must be achieved prior to applying the *stack*(X, Y) operator:

clear(*a*)
holding(*b*)

The stack is popped, yielding **clear**(*a*). This is already true and so no action is necessary. The stack is popped again, yielding **holding**(*b*) as the goal. Two operators could now be applied:

pickup(X) and *unstack*(X, Y)

since both of them have the fact **holding**(X) in their add-lists. Which operator is applied depends on the ordering of the operators in the list. If we apply *pickup*(X) first then the stack appears as:

arm_empty
clear(*b*)
on_table(*b*)

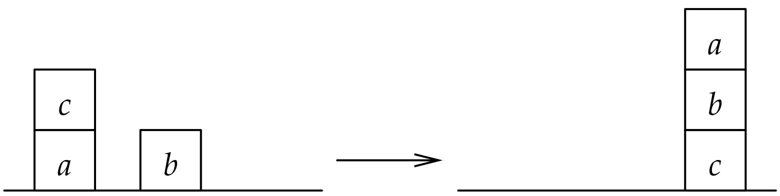
The stack can now be popped three times with STRIPS finding that each goal already holds in the current world, so no further action is required. The stack is empty and so *pickup*(*b*) can be applied and so the plan is as follows:

$$\text{START} \longrightarrow \textit{pickup}(b) \longrightarrow \textit{stack}(b, a) \longrightarrow \text{END}$$

In this example, STRIPS is able to go to the correct solution directly. This is not always the case in practice.

Problem of interacting subgoals

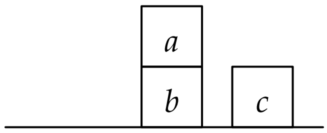
STRIPS reliance on the simple stack-based implementation means that it solves the constituent subproblems of a conjunction of goals as if they were completely decoupled. Thus it makes use of the **linearity assumption**: goals G_1, G_2 and G_3 can be achieved by concatenating the plan for G_1 and the plan for G_2 and the plan for G_3. Whilst this brings certain benefits, it can lead to inefficient plans or even total failure. Consider, for example, the following problem:



With the given initial state and the conjunctive goal **on**(*a*, *b*)∧**on**(*b*, *c*), STRIPS attempts to solve firstly **on**(*a*, *b*) completely and then solve **on**(*b*, *c*). The plan for solving the first goal is:

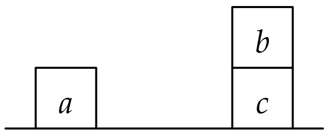
$$\text{START} \longrightarrow \textit{unstack}(c, a) \longrightarrow \textit{put\_down}(c) \longrightarrow \textit{pickup}(a) \longrightarrow \textit{stack}(a, b) \longrightarrow \text{END}$$

which results in:



STRIPS now attempts to solve the second goal, $\mathbf{on}(b, c)$, but in achieving this, it inadvertently undoes the first goal, $\mathbf{on}(a, b)$. The planning steps are as follows:

$$\text{START} \longrightarrow \mathit{unstack}(a, b) \longrightarrow \mathit{put\_down}(a) \longrightarrow \mathit{pickup}(b) \longrightarrow \mathit{stack}(b, c) \longrightarrow \text{END}$$



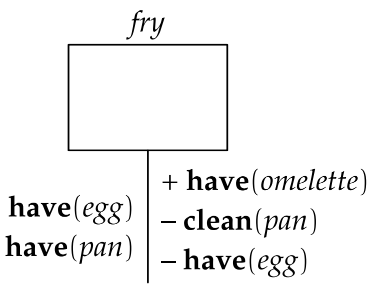
Intuitively, we can see that the correct plan for this problem is:

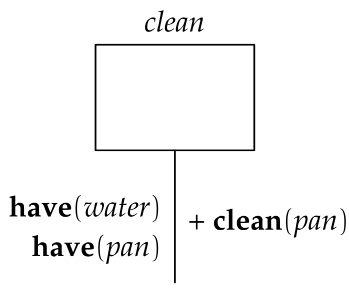
$$\text{START} \longrightarrow \underline{\mathit{unstack}(c, a)} \longrightarrow \underline{\mathit{put\_down}(c)} \longrightarrow \mathit{pickup}(b) \longrightarrow$$
$$\mathit{stack}(b, c) \longrightarrow \underline{\mathit{pickup}(a)} \longrightarrow \underline{\mathit{stack}(a, b)} \longrightarrow \text{END}$$

[Underlined operators contribute to $\mathbf{on}(b, a)$, those in plain text to $\mathbf{on}(b, c)$.]

In order to achieve the correct solution, the planning for each conjunctive goal has to be interleaved in some way.

As a further illustration of the above difficulty, consider the problem of making an omelette given the following operators:





The problem and the series of partial plans that lead to the correct omelette making plan are shown in Figure 11.

Clearly, in the above problem, if we had cleaned the pan first and then made the omelette, the final goal would not have been achieved. Thus we need a means of overcoming unfavourable interactions of this sort.

The above difficulties arise out of the fact that planning involves a series of **local** actions in order that a **global** objective may be achieved.

Handling subgoal interaction

A large part of the research effort in planning has been to deal with subgoal interaction. The research work falls into one of the following 3 broad categories:

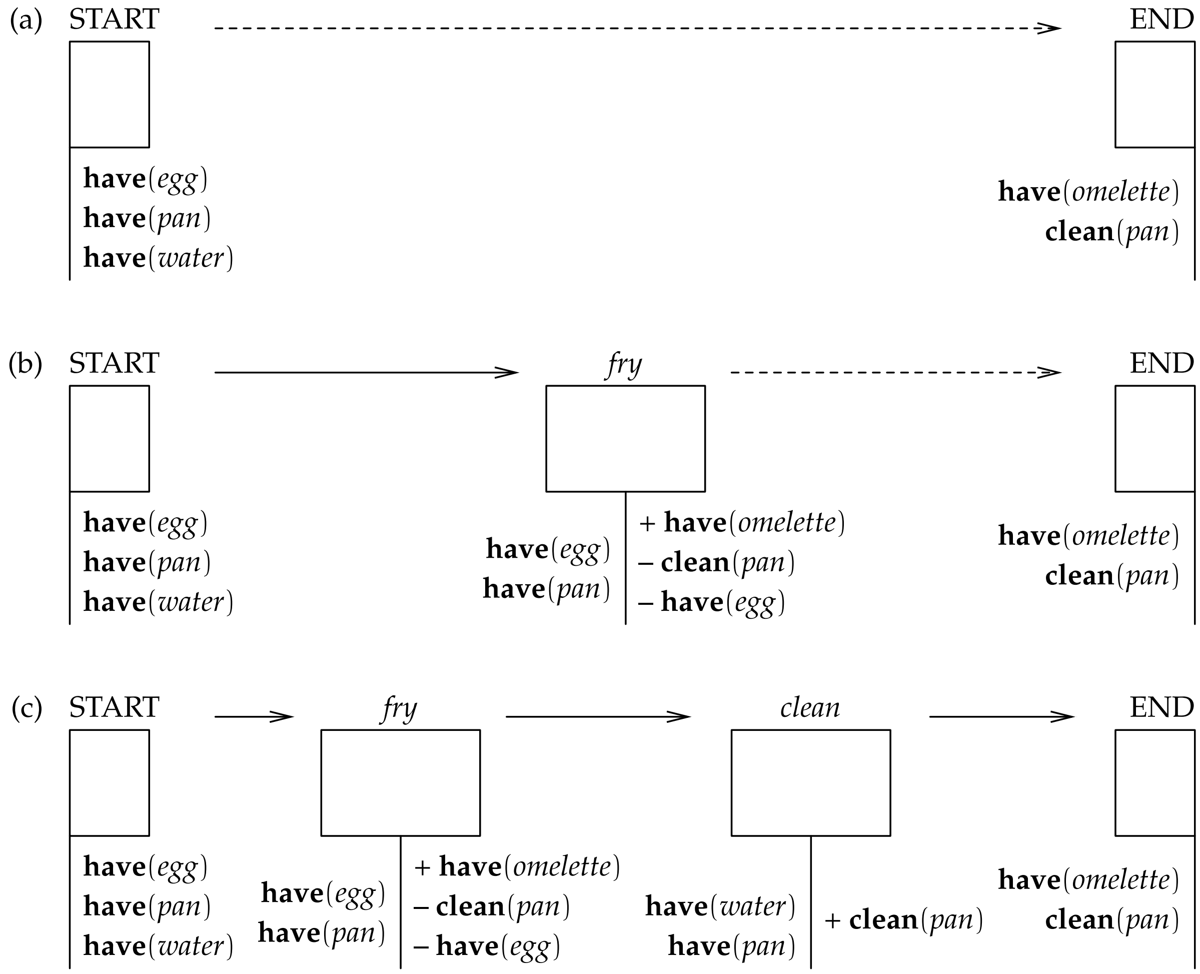


Figure 11: How to fry an omelette ? — the problem and its solution.

1. Linear planners which accept that their plans may have errors in them and attempt to debug erroneous goals by backtracking and reordering goals.
2. Linear planners which attempt to spot problems as new actions are added to the plan and correct by 'regressing' either the offending action or its parent goal.
3. Non-linear planners which do not commit themselves to any ordering of conjunctive goals except to avoid harmful interaction.

Linear planning — correction by goal reordering

The success or otherwise of planning for a simple problem may be dependent on the ordering of the conjunctive goals, e.g. in washing a car, the conjunctive ordering of (**clean**(*roof*) $\wedge$ **clean**(*doors*)) will be successful, whereas (**clean**(*doors*) $\wedge$ **clean**(*roof*)) will not be. In more complicated examples, however, it may well be impossible for us to specify the correct ordering for tackling a conjunction in advance.

One way of overcoming this problem is to allow the planner to save the points at which it makes the ordering choices, so that in the event of failure it can backtrack and try a different ordering. This technique is computationally wasteful, since useful

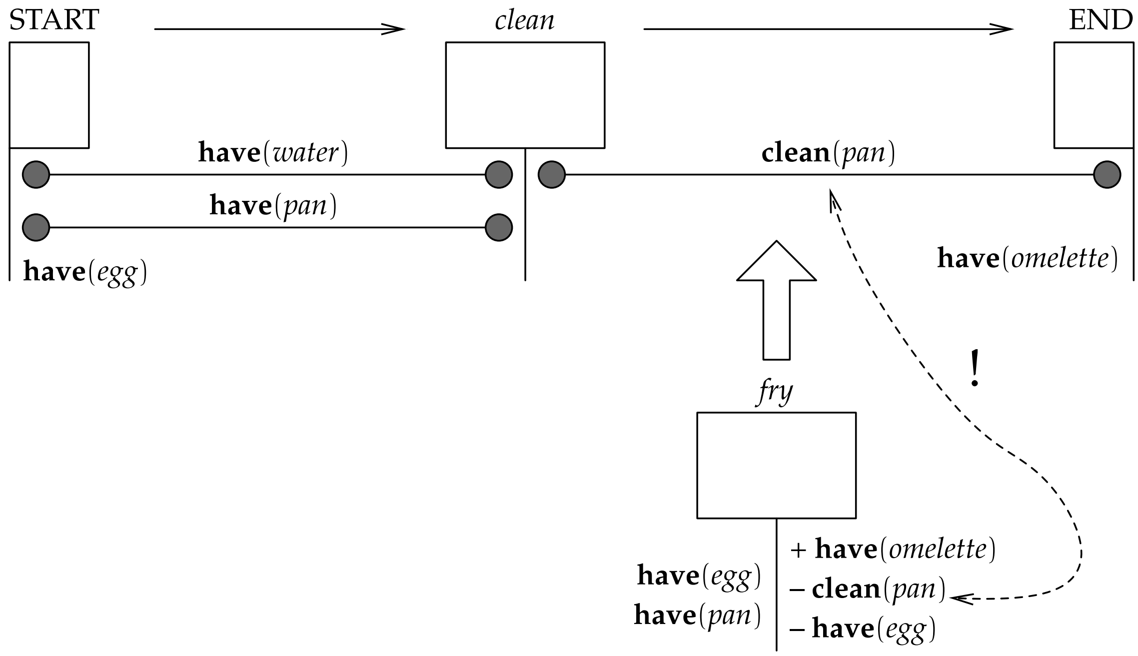


Figure 12: Linear planning with range violation — detecting subgoal interaction

planning may have been done on decoupled subgoals which then has to be undone and re-planned, perhaps several times.

Linear planning — correction by regression

These planners still employ the linearity assumption, but now attempt to anticipate interaction problems as new actions are added rather than wait until plan failure occurs. They first of all develop a plan for the first conjunct in the normal way, and then plan the subsequent goals while ensuring that the actions they introduce do not interfere with the existing partial plan. To ensure this, they need to explicitly record **ranges** (protection intervals, holding periods) over which achieved goals must remain true — typically, these will range from the achieving action to the action which has that goal as its precondition).

These planners first try to insert a new action at the end of the partial plan on which they are working. If they find that this action violates a range for some goal G (because it deletes G) then they search to find a new place in the plan for the action or goal which gave rise to that interference.

Applying such a technique to our omelette-making, we can get to a situation illustrated in Figure 12.

This clearly leads to a plan violation.

One way to proceed with the (partially) successful plan shown above is to ask: Is there some fact which, if this fact is true before *clean*, will make **have**(*omelette*) true after *clean*?

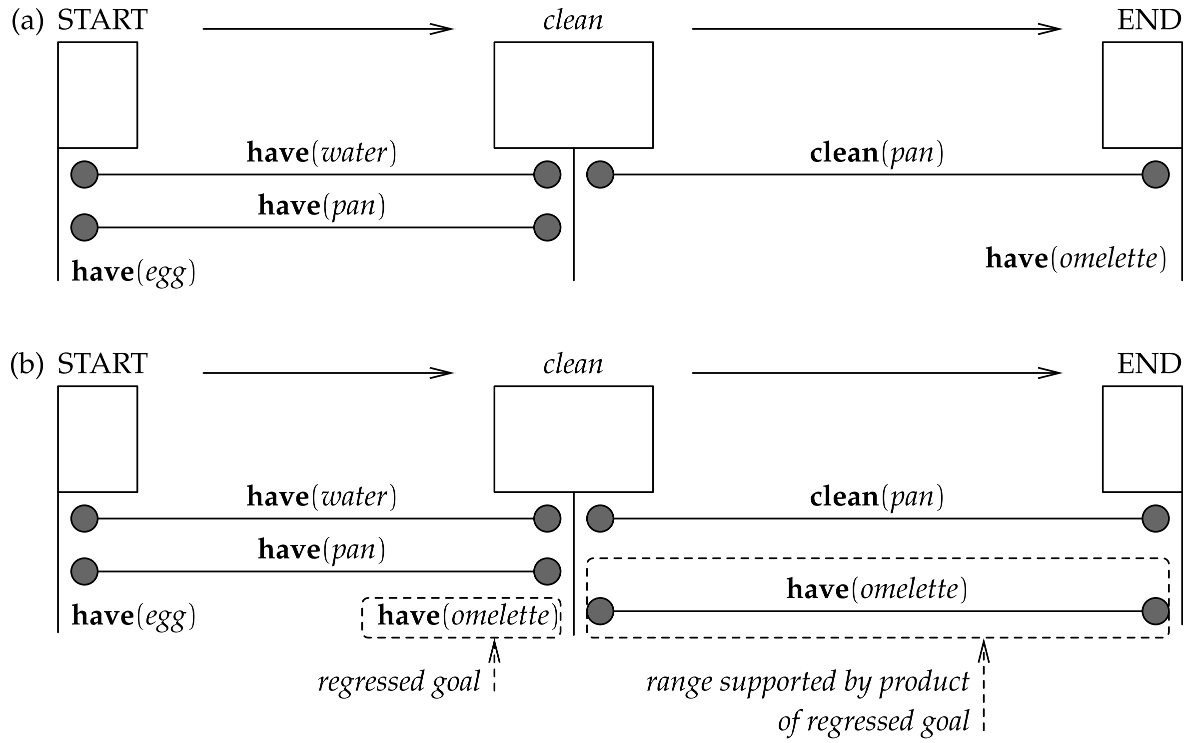


Figure 13: Example of a goal regression within a partial plan

If there is, it is the regressed version of the original goal fact. In the above example, regression of **have**(*omelette*) over *clean* yields a regressed goal which is the same as the original goal — after regression we have the situation shown in Figure 13 (b).

The *fry* action can be inserted before the *clean* action without violating any ranges.

Non-linear plans

Although the regression methods are quite successful, they have been generally superseded by non-linear planning which has become the standard approach for coping with sub-goal interactions. Such planners no longer employ the linearity assumption, but instead deliberately leave subgoals unordered unless an ordering must be imposed to avoid interaction problems.

For the omelette-making problem we can draw the non-linear plan, with ranges included shown in Figure 14

Note the intentional ambiguity relating to the two total orderings in Figure 14 (a). Clearly a range violation is possible whenever one has the situation represented in Figure 14 (b). Thus we need to rule out any interpretation of the ambiguity that would definitely involve a range violation. In Figure 14 (b), action *C* must clearly be placed either wholly after the range or wholly before it — sometimes only one choice is possible. This addition of order is called **linearising** a plan.

If we examine the partial omelette-making plan in Figure 14 (a), then we can see a possible range violation: of **clean**(*pan*) by the effect "– **clean**(*pan*)" of the fry action. The only possible linearisation is to put *fry* before *clean* giving the correct plan.

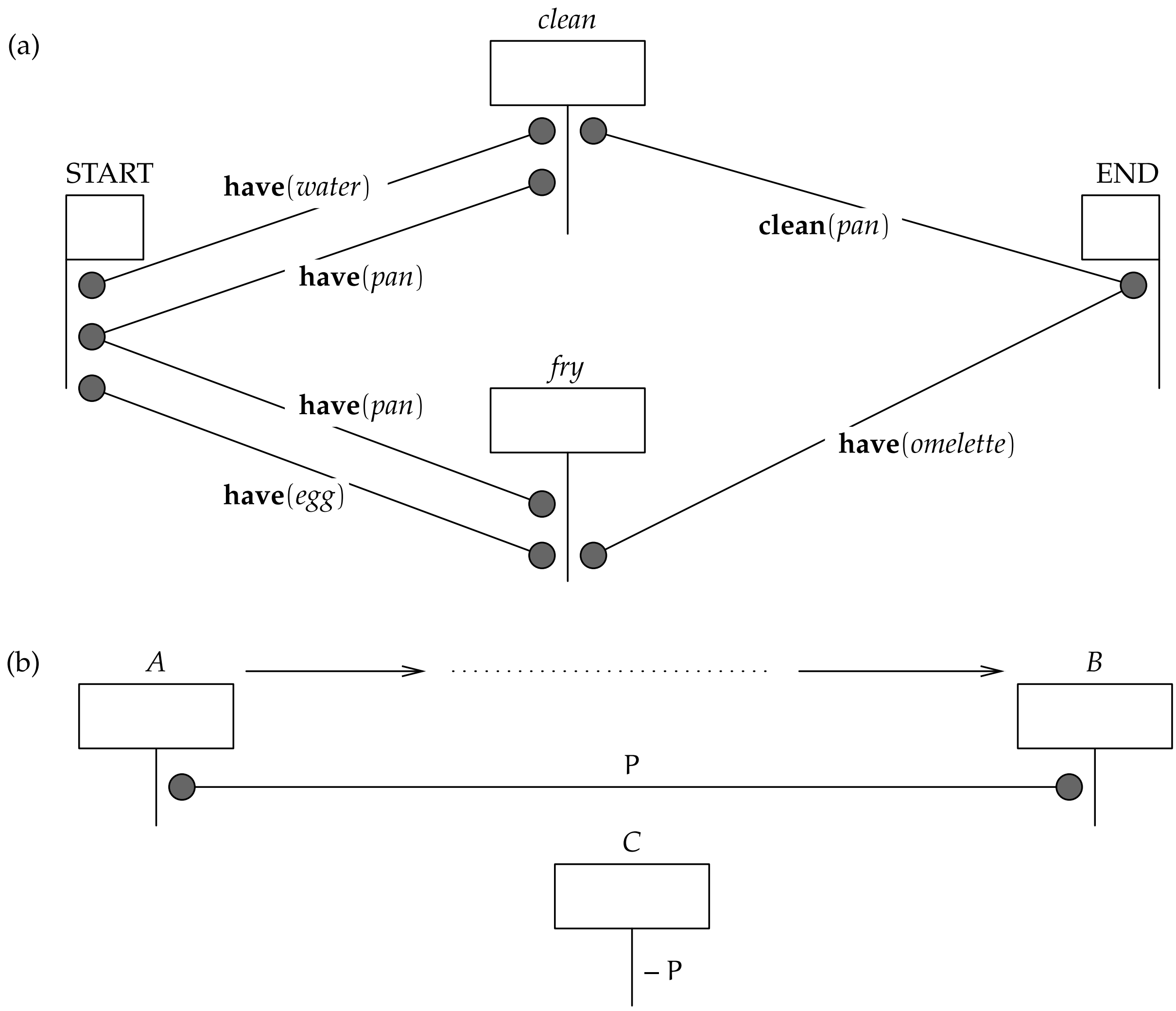


Figure 14: Non-linear planning

Frame Problem

The Frame Problem is the problem of maintaining an appropriate informational context, or frame of reference at each stage during the problem-solving process.

In a dynamic environment, there is the problem of knowing what changes have and have not taken place following some action. Some changes will be the direct result of the action. Other changes will be the result of secondary or side effects.

Some key aspects of the frame problem can be illustrated by considering the scene in Figure 15.

Facts: F1 — A table is in the centre of the room

F2 — A book is on the table

F3 — Fred is in the doorway

F4 — some facts relating to window and alcove

Actions: A1 — Fred can go from one place to another

A2 — Fred can move the table from one place to another

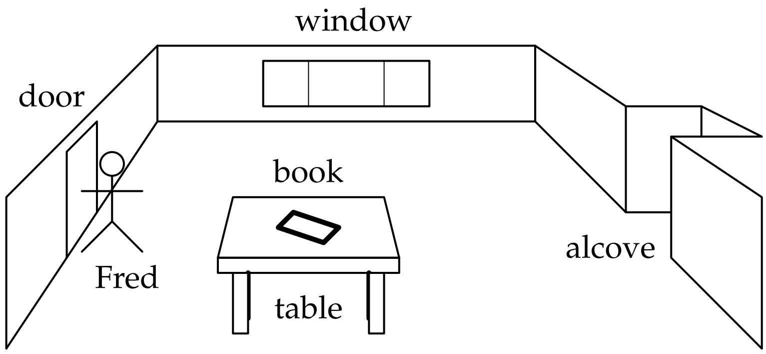


Figure 15: Scene illustrating the Frame Problem

Consider the following problems:

- **Goal 1 — Fred should be in the alcove**
  1. Accomplished by an A1-type action. Would change F3 to "Fred is in the alcove" while F1, F2, F4 would stay unchanged.
  2. Accomplished by an A2-type action. F1 would now change.
- **Goal 2 — The table should be in the alcove**

  Ambiguity removed — facts F1 and F3 change.

How can we decide which facts change?

- **Proposal 1**: Determine which facts change by comparing the goal statement with the known facts.
  - Acceptable if Goal 1 achieved through an A1 action but not A2.
  - Goal 2 makes no mention of Fred's position which must change.
- **Proposal 2**: Specify which facts are changed by each action operator. E.g.

  A1 changes Fred's position

  A2 changes position of both Fred and table.

  Occasional difficulties arise, e.g. in the initial state, it can be deduced from F1 and F2 that

  F5 — the book is in the centre of the room

  This will no longer be true when Goal 2 has been reached even though F5 has no apparent relation to A2.

If a representation is exhaustive, there will be no unspecified side effects, and the frame problem effectively disappears. The difficulty of the frame problem results from the fact that it is impossible to build a completely exhaustive knowledge base for most domains

Unit 13. Natural Language Processing 1/3

Learning Outcomes. You should be able to

- list several common application tasks of NLP
- describe four early NLP systems and their contributions
- list and describe typical levels of a language understanding system
- draw diagrams illustrating how a given simple sentence is analysed at different stages
- explain why NLP is very hard

– NLP is an active and significant area of current AI research.

– By far the largest part of human linguistic communication occurs as speech — written language is a fairly recent invention and is easier to process (computationally) than speech.

Machine Translation (MT)

– Machine translation was vigorously pursued in the 1950's and 1960's in the USA, Britain and Russia.

– Early systems had a dictionary and a few rules of grammar. Little success was achieved.

– Probably the most influential critic of the early work was a linguist, Bar-Hillel, who in 1964 published a critique which stated that sense disambiguation relies in the *deep understanding* of a sentence.

– In 1966, the Pierce Report finally brought the early MT work to an end by stating that work on MT could not be justified in terms of practical output.

– The main outcome of this research was the realisation that human language is extraordinarily complicated and that considerably more research into grammar and meaning was needed — as Winograd has stated:

> *'With years of hindsight and experience, we now understand better why the early optimism was unrealistic. Language, like many human capabilities, is far more intricate and subtle than it appears on first inspection.'*

The Problem

In the act of understanding a seemingly simple sentence, we bring to bear a considerable amount of information of widely varying sorts — not only knowledge about the language but also knowledge about the domain of discourse. The following examples illustrate:

'The city councilmen refused to give the women a permit for a demonstration because they feared violence'

'The city councilmen refused to give the women a permit for a demonstration because they advocated revolution'

Turing Test

– A game played between three people: a man, a woman, and an interrogator who may be of either sex. The interrogator stays in a room apart from the other two, and attempts to determine which of the other two is the man and which is the woman. The man tries to convince the interrogator that he is the woman. Communication between the participants is via a terminal.

– We now ask the question "What will happen when a machine takes the part of the man in this game?"

– The central idea of the Turing test is that the ability to successfully communicate with a discerning person in a free and unbounded conversation is a better indication of intelligence than any other attribute accessible to measurement.

– Vulnerable to a number of criticisms:

  – does not test abilities requiring perceptual skills or manual dexterity

  – constrains machine intelligence to fit a human mould

Early Systems

Most of the early systems that claimed to carry out some task other than just parsing (e.g. problem-solving, question-answering) were based on crude searches for keywords or patterns in the input string

Eliza

– The ELIZA system of Weizenbaum carried this approach to its extreme — mode of conversation followed the psychiatric interview ('Tell me about . . . '). [Weizenbaum CACM 1966, Jan, 36]

– The program was based on keyword/phrase matching and response. These techniques with a few enhancements produced an amazing level of fluency but the program could hardly be said to 'understand'.

– The program could pass the Turing test — this illustrates the difficulty of judging a program by its output or of defining understanding.

– ELIZA could be described as a first generation system.

– Second generation systems tackled a much larger range of language problems. This was achieved by restricting the domain of discourse to a narrow field. Examples of such second generation systems are:

Lunar

- System developed by Woods in 1967–1972. The system was designed to answer questions about the mineral samples brought by the astronauts.
- Woods developed **transition net grammars** to describe the grammatical facts about English needed for interpreting complicated structures. The system used the grammar to convert sentences into requests in a special query language which was designed to interface with an information retrieval system built for a large data base e.g.

> *'How many rocks have greater than 50ppm nickel?'*
> *'Of the type A rocks, which is the oldest?'*

SHRDLU

- The SHRDLU program of Winograd (1972) had an overwhelming effect on research into NLP.
- This was a dialogue system that could converse with a human user about a simple table-top world containing simple building blocks. The program had a crude simulation of a 'hand' and 'eye' which it could use within the simulated **'blocks' world**.
- It can be seen from the sample dialogue in Figure 16 that the program could:

  1. answer questions
  2. execute commands — robot carries out action
  3. accepts information in an interactive English dialogue

LIFER

- is a system for creating English language interfaces to other computer software. The goal was to provide a systems designer (who is not a linguist) with the ability to tailor a natural language 'front-end' to an application.
- LIFER allowed the systems designer to specify the nature of the processing to be carried out on the natural language inputs by writing pattern and response expressions. These expressions can be thought of as more complex than, but similar to, the ELIZA patterns.
- LIFER is able to handle ellipsis, e.g. the series of questions:

  "How old is John?" "How tall?" "Mary?"

  would be interpreted as:

  "How old is John?" "How tall is John?" "How tall is Mary?"

- When a given pattern is recognised by the parser, the associated expression is evaluated to produce the desired response. E.g. a specification

user: How many blocks are on the table?
SHRDLU: `FOUR OF THEM`

user: Is at least one of them narrower than the one which I told you to pick up?
SHRDLU: `YES, THE RED BLOCK`

user: Can the table pick up blocks?
SHRDLU: `NO`

user: Can a pyramid be supported by blocks?
SHRDLU: `YES`

user: Will you please stack up both of the red blocks and either a green cube or a pyramid?
SHRDLU: `OK`

user: Which cube is sitting on the table?
SHRDLU: `THE LARGE GREEN ONE WHICH SUPPORTS THE RED PYRAMID`

user: Had you touched any pyramid before you put the green one on the little cube?
SHRDLU: `YES THE GREEN ONE`

user: When did you pick it up?
SHRDLU: `WHILE I WAS STACKING UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE`

user: Why?
SHRDLU: `TO GET RID OF IT`

user: Why did you do that?
SHRDLU: `TO CLEAN OFF THE RED CUBE`

user: Why did you clear of that cube?
SHRDLU: `TO PUT IT ON A LARGE GREEN CUBE`

user: How many objects did you touch while you were doing it?
SHRDLU: `FOUR OF THEM`

user: What did the red cube support before you started to clean it off?
SHRDLU: `THE GREEN PYRAMID`

user: A "steeple" is a stack which contains two green cubes and a pyramid
SHRDLU: `I UNDERSTAND`

user: Are there any steeples now?
SHRDLU: `NO`

user: Build one
SHRDLU: `OK`

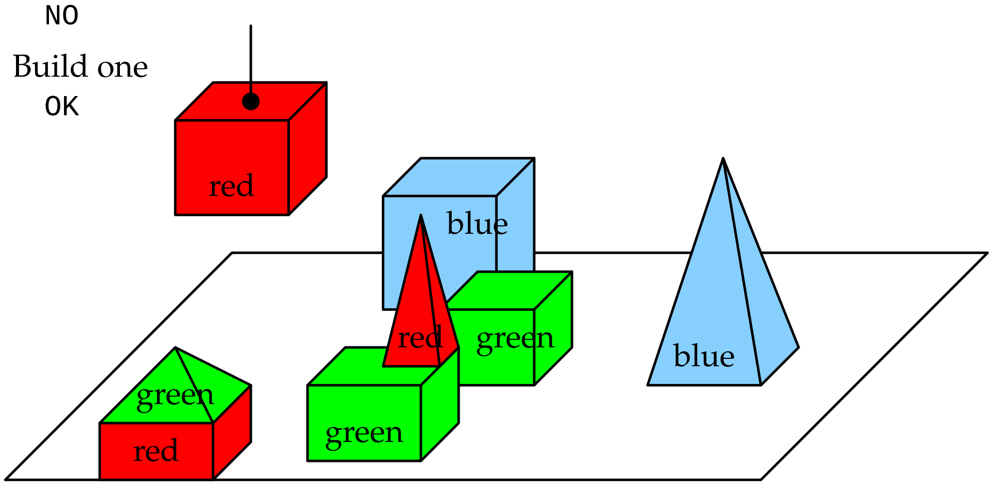


Figure 16: Sample dialogue with SHRDLU

"How ⟨ATTRIBUTE⟩ is ⟨PERSON⟩"

indicates that when an input sentence such as, "How old is John?" is entered, the system should identify "old" with ⟨ATTRIBUTE⟩ and "John" with ⟨PERSON⟩ . These interpreted words are then used in appropriate interactions with the application software.

Components of an understanding system

The process of understanding natural language may be divided into several levels:

1. Signal (acoustical) level

   This is concerned with speech input and processing — analysis at this level includes the extraction of sound units called "phones" — a phoneme is the smallest unit of sound

2. Morphological analysis

   Individual words are analysed into their components and non-word tokens (e.g. , ;) are separated from the words.

3. Syntactic analysis (e.g. parsing)

   The grammatical correctness of the sequence of words is determined. This gives rise to structures which show how the words relate to one another in abstract ways.

4. Semantic analysis

   The structures determined from the syntactic analysis stage for grammatically correct sentences are assigned meanings.

5. Pragmatic analysis

   This is concerned with the use of high-level knowledge to determine the *actual* meaning of a sentence. To achieve this, we may need to **use world knowledge**: the knowledge of the physical world, the world of human social interaction, and the role of goals and intentions in communication.

Some of these components are illustrated in Figure 17.

Levels of analysis of natural language

Linguists have defined different levels of analysis for natural language. The *particular* levels of interest to the AI researcher (and to us in this module) are the following. (They correspond to some of the system components above.)

Syntax analysis

studies the rules for combining words into legal phrases and sentences and the use of those rules to parse and generate sentences.

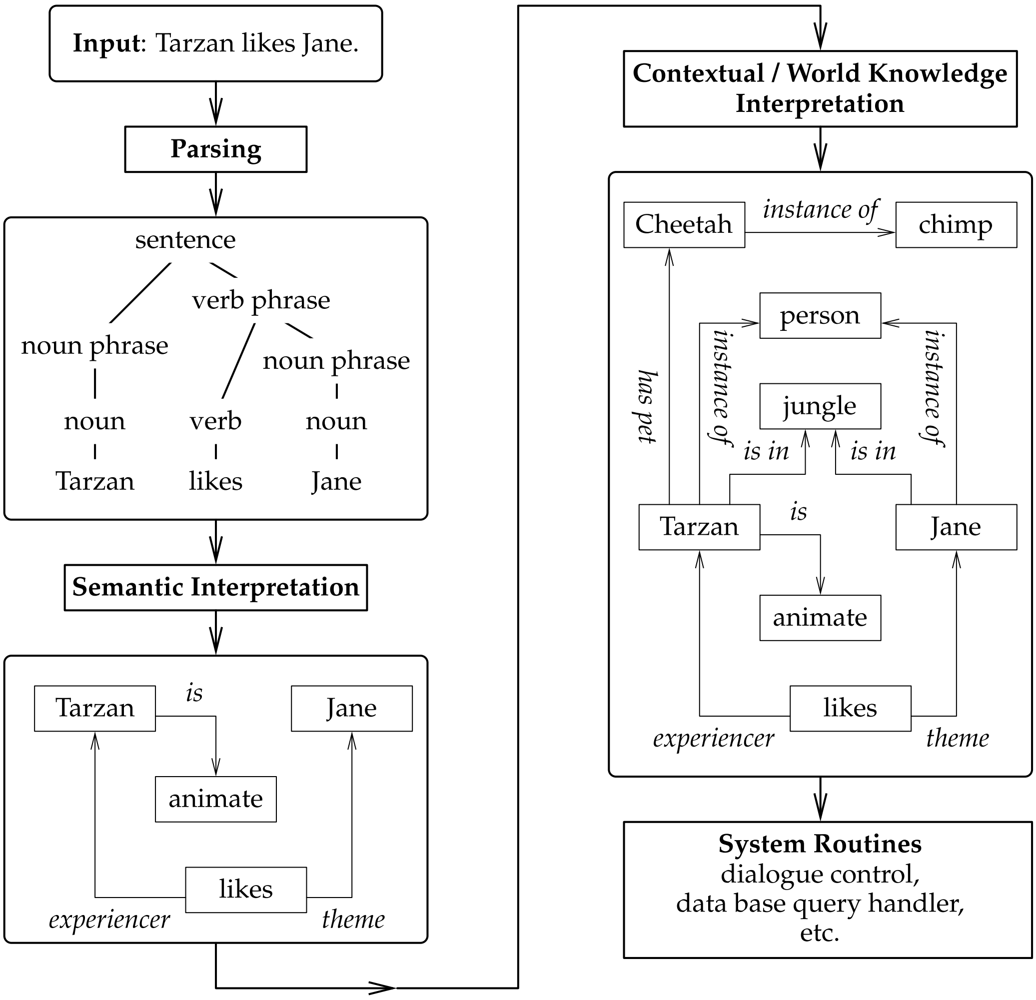


Figure 17: Stages of a Text Understanding System

Semantic analysis

concerned with the meaning of words, phrases and sentences and the way in which it is conveyed in natural language.

Pragmatics

is the study of the way in which language is used and its effects on the listener.

World Knowledge

includes knowledge of the physical world, the world of human social interaction and the role of goals and intentions in communication — is essential to understanding the full meaning of a text or conversation.

Problem of Combinatorial Explosion

- The exhaustive search through all logical possibilities becomes too time-consuming even for the fastest computer — when only a few extra dimensions of variability are added to a manageable problem.
- This problem is not unique to NLP.

Unit 14. Natural Language Processing 2/3

```
Learning Outcomes.  You should be able to
 - read and understand PSG rules
 - check whether a sequence is legal in a given PSG using
   both top-down and bottom-up derivations
 - define CFG and CSG and regular grammars
 - describe FSTNs and RTNs
 - check whether a sequence is accepted by an FSTN or RTN
 - describe the relationship between regular grammars and
   FSTNs and between CFGs and RTNs
 - explain why RTNs are not sufficient for NLP parsing
 - define the term 'semantic grammar'
```

General Approaches to NLP

There have been three general approaches to natural language processing:

(i) Pattern matching through the recognition of keywords — e.g. Eliza.
(ii) Mapping text to prescribed primitives — use of frames and scripts.
(iii) Analysis of sentences using syntax and semantics — this is the approach considered in this Unit.

Phrase Structure Grammar (PSG)

– is a tool which can represent a very large number of sentences by a small set of rules. Each rule consists of a symbol followed by $\longrightarrow$ and a string of symbols. There are two kinds of symbols:

– **terminals** = parts of sentences (usually words)
– **nonterminals** = abstract names that do not appear in sentences (e.g. **S**, **VP**)

An example grammar:

1.	**S** $\longrightarrow$ **NP VP**		
2.	**VP** $\longrightarrow$ **V NP**		
3.	**VP** $\longrightarrow$ **V NP PP**	**S**	= **sentence**
4.	**PP** $\longrightarrow$ **P NP**	**NP**	= **noun_phrase**
5.	**P** $\longrightarrow$ *to*	**VP**	= **verb_phrase**
6.	**V** $\longrightarrow$ *gave*	**V**	= **verb**
7.	**NP** $\longrightarrow$ **N**	**PP**	= **prepositional_phrase**
8.	**NP** $\longrightarrow$ *the* **N**	**P**	= **preposition**
9.	**N** $\longrightarrow$ *John*	**N**	= **noun**
10.	**N** $\longrightarrow$ *Mary*		
11.	**N** $\longrightarrow$ *book*		

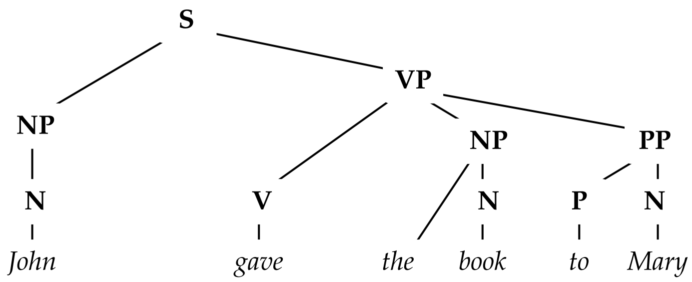


Figure 18: A parse tree.

– A **legal sentence** is any string of terminals that can be **derived** using the rules. For example, a derivation of the sentence "John gave the book to Mary" would be

String	**apply rule**
sentence	1
noun_phrase verb_phrase	7
noun verb_phrase	9
John **verb_phrase**	3
John **verb noun_phrase prepositional_phrase**	6
⋮	⋮
John gave the book to **noun**	10
John gave the book to Mary	

– This is an example of a **top-down** derivation.

– A **bottom-up** derivation starts with a string of terminals and tries to work towards the sentence symbols through a series of substitutions. For example, a bottom-up verification that "John gave the book to Mary" is a sentence may look as follows

String	**apply rule**
John gave the book to Mary	9
noun *gave the book to Mary*	7
noun_phrase *gave the book to Mary*	6
noun_phrase verb *the book to Mary*	11
noun_phrase verb *the* **noun** *to Mary*	8
noun_phrase verb noun_phrase *to Mary*	5
⋮	⋮
noun_phrase verb_phrase	1
sentence	

– A derivation may be represented as a tree structure, known as the **parse tree**, in which each node contains a symbol of the grammar. The parse tree reflects the structure of the sentence according to the specified grammar. The parse tree of the above sentence using the above grammar is in Figure 18.

– Parsing is the problem of constructing a parse tree for an input string from a formal definition of a grammar.

– Top-down parsers begin with the **sentence** symbol and attempt to derive the input string.

– Bottom-up parsers begin with the input string and attempt to yield the **sentence** symbol.

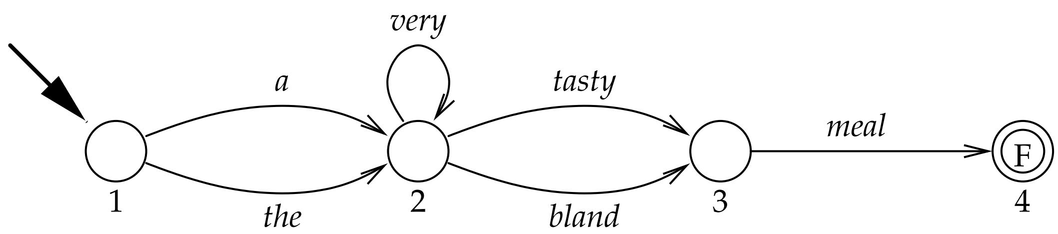
– Not only does the existence of a parse tree prove that a sentence is legal in the grammar, but it also determines the structure of the sentence. It plays an essential role in semantic interpretation by defining intermediate stages in a derivation at which semantic processing may take place.

– To determine the meaning of a word, a parser must have access to a lexicon. A **lexicon** is a dictionary of words where each word contains some syntactic, semantic and possibly pragmatic information.

– In our example grammar, every rule has one nonterminal symbol on the left-hand side. Such grammars are called **context-free grammars** (CFG). Grammars in which rules can have more than one symbol on the left-hand side are called **context-sensitive grammars** (CSG).

Finite State Transition Networks (FSTNs)

– FSTN is a simple theoretical device consisting of a set of states (drawn as nodes) with arcs leading from one state to another. It is used to represent a simple language. An example FSTN follows.



– The device can read a sequence of words and decide whether the sequence is a valid sentence in its own little language or not.

– One of the states is marked as the **initial state** (in our example, using a big arrow). This state is the focus of the device before it starts reading.

– As individual words are read, the focus (called **current state**) of the device changes, following arcs that are labelled with matching words or symbols.

For example, when reading "*a very very tasty meal*", the device will first focus on state 1, then state 2 then state 2 again, then on state 3 and finally on state 4.

– The device is said to **accept** a sequence of words if, starting from the initial state at the beginning of the sequence, it can reach a **final state** at the end of the input. (Final states are marked with F in our example.)

For example, the sequence "*a very very tasty meal*" is accepted by our FSTN but "*a very spicy meal*" is not.

– **Regular grammars** are context-free grammars in which all rules have on their right hand side either a single terminal or a sequence consisting of one terminal followed by one nonterminal. For example,

1. **A1** $\longrightarrow$ *a* **A2**
2. **A1** $\longrightarrow$ *the* **A2**
3. **A2** $\longrightarrow$ *very* **A2**
4. **A2** $\longrightarrow$ *tasty* **A3**
5. **A2** $\longrightarrow$ *bland* **A3**
6. **A3** $\longrightarrow$ *meal*

- For every FSTN there is an equivalent regular grammar and vice versa. For example, the above regular grammar is equivalent to the FSTN shown earlier.
- Regular languages (and therefore FSTNs) are inadequate for dealing with the complexity of natural languages. A necessary extension to FSTN is to provide a *recursion mechanism* that increases their recognition power.

Recursive Transition Networks (RTNs)

- RTN is a collection of finite-state transition networks in which a label of an arc may be not only a word but also a *name of the same or another network* to be given temporary control of the parsing process. See an example in Figure 19.
- Network names usually correspond to nonterminals in grammars, i.e. each network represents a syntactic construct, e.g. **prepositional_phrase**.
- RTN operates similarly to an FSTN. If the label on an arc is a word then the arc may be taken if the word being scanned is the same. Otherwise, if the arc is labelled with a network name then control is transferred to the corresponding named subnetwork, which continues to process the sentence, returning control when it successfully finishes. If the subnetwork fails, the calling network cannot take this arc.
- Any subnetwork may call any other subnetwork including itself — recursion is allowed.
- For example, the example RTN would accept the following sequences:

  S: *The little boy in the swimsuit kicked the red ball with force.*
  NP: *The little boy in the swimsuit*
  NP: *the red ball*
  PP: *with force*
  NP: *the red ball with force*

- The sentence above can be parsed by our RTN in two different ways. The **PP** *with force* may be parsed either

  - as part of the **NP** "*the red ball with force*"
  - or by the network **S**.

  This structural ambiguity cannot be resolved without some world knowledge.
- The recursive structure of our RTN can be illustrated by the following **NP** phrases:

  a friend with a friend
  a friend with a friend with a friend
  a friend with a friend with a friend with a friend

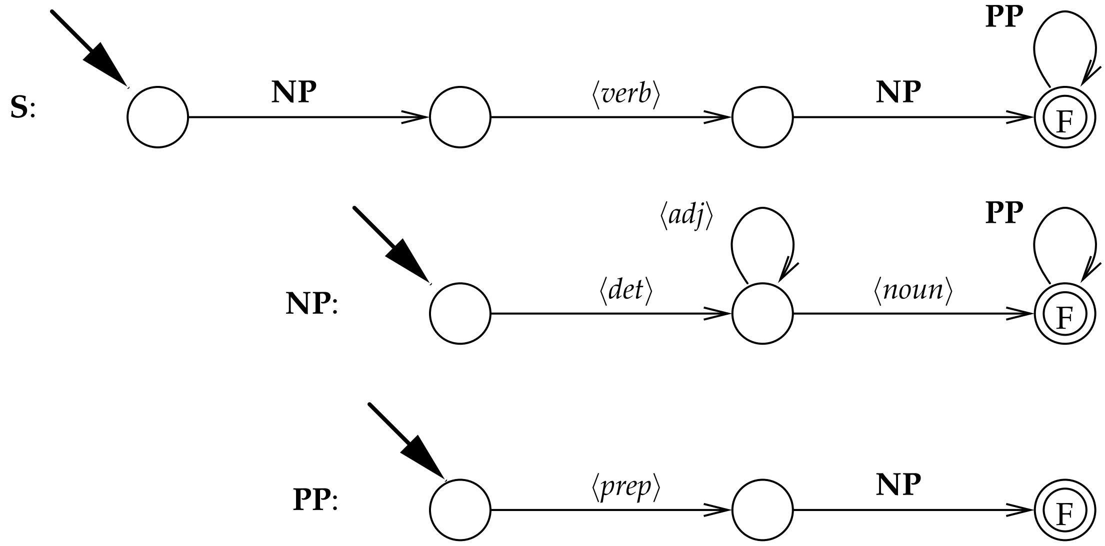


Figure 19: A recursive transition network.

- For every RTN, there is an equivalent CFG and vice versa.
- RTNs (and therefore CFGs) are still insufficient to handle some important grammatical aspects of natural languages.
- More importantly, an RTN on its own does not construct any parse tree. It is only capable of accepting or rejecting a sentence based on the grammar and syntax of the sentence. An RTN can be augmented to produce a parse tree while parsing a correct sentence.
- So-called **augmented transition networks** (ATNs) solve both problems: they allow to recognise all grammatical complexities and they can naturally produce a parse tree together with checking the correctness. The description of ATNs is beyond the scope of this introductory module.

Semantic Analysis

- Transition networks are capable of parsing sentences for different grammars — how then can we derive semantic knowledge structures?
- Experience has shown that semantic interpretation is the most difficult stage in NL understanding.
- Semantic interpretation requires that utterances be transformed into appropriate knowledge structures, semantic networks, frames, scripts etc.
- Semantic interpretation can be approached in several ways:

(i) After parsing, a semantic analyser is invoked to produce a semantic structure.

(ii) A sentence is transformed directly into a target structure with little syntactic analysis. Such an approach relies, for example, on semantic grammars.

(iii) Perform syntactic and semantic analysis concurrently using semantic information to guide the parsing process. This is the approach taken by SHRDLU.

Semantic grammars

- These are grammars in which semantic information is encoded into a grammar in which the nonterminals and the production rules are governed by semantic as well as syntactic functions.
- In addition, there is usually a **semantic action** encountered with each rule of grammar so that the result of parsing and applying all the associated semantic actions is the meaning of the sentence.
- Semantic grammars are used in a language interface to a data base management system in LIFER.
- LIFER used semantic categories like $\langle\textbf{SHIP-NAME}\rangle$, $\langle\textbf{ATTRIBUTE}\rangle$ in a grammar such as:

$$
\begin{aligned}
\textbf{S} &\longrightarrow \text{“What is } \langle\textbf{ATTRIBUTE}\rangle \text{ of } \langle\textbf{SHIP}\rangle\text{?”} \\
\langle\textbf{ATTRIBUTE}\rangle &\longrightarrow \textit{the}\ \langle\textbf{ATTRIBUTE}\rangle \mid \langle\textbf{ATTRIBUTE}\rangle \\
\langle\textbf{ATTRIBUTE}\rangle &\longrightarrow \textit{length} \mid \textit{beam} \mid \textit{commander} \mid \textit{fuel} \mid \ldots \\
\langle\textbf{SHIP}\rangle &\longrightarrow \textit{the}\ \langle\textbf{SHIP-NAME}\rangle \mid \langle\textbf{SHIP-NAME}\rangle \\
&\qquad \mid \textit{the}\ \langle\textbf{SHIP-CLASS}\rangle \mid \langle\textbf{SHIP-CLASS}\rangle \mid \ldots \\
\langle\textbf{SHIP-NAME}\rangle &\longrightarrow \textit{Kennedy} \mid \textit{Atlantis} \mid \ldots \\
\langle\textbf{SHIP-CLASS}\rangle &\longrightarrow \textit{destroyer} \mid \textit{submarine} \mid \textit{convoy} \mid \ldots
\end{aligned}
$$

  With this grammar sentences of the form:

  What is the length of the Kennedy?

  Who commands the Kennedy?

  What is the name and location of the carrier nearest to New York?

  could be posed and answered.
- The difficulty with semantic grammars is that they tend to have a great many productions for all but the smallest languages and hence the computational processing can be time consuming.

Unit 15. Natural Language Processing 3/3

Learning Outcomes. You should be able to

- describe a user interaction with SHRDLU
- understand simple examples of PLANNER expressions inside SHRDLU
- illustrate on examples how the SHRDLU parsing process is guided by semantics and world knowledge
- give examples of semantic markers and of their usage
- discuss several aspects in which SHRDLU is weak when compared with the human understanding of language

– The SHRDLU program of Winograd (1972) had an overwhelming effect on research into NLP.

– This was a dialogue system that could converse with a human user about a simple table-top world containing building blocks. The program had a crude simulation of a 'hand' and 'eye' which it could use within the simulated 'blocks' world.

– See sample dialogue in NLP 1 lecture notes.

– The program could:

1. answer questions
2. execute commands — virtual robot carries out action
3. accept information in an interactive English dialogue

Detail

– The program is organised to deal with the three different types of knowledge:

(i) syntax analysis,

(ii) semantic analysis,

(iii) the data base of assertions and the procedures (in PLANNER) which represent the knowledge of the physical world.

– *"Language should not be reduced to the separate [areas] of syntax, semantics and pragmatics in the hope that by understanding each of them separately we can understand the whole. The key to the function of language as a means of communication is in the way these areas interact."* — Winograd recognised this and introduced it into his program.

– The most important element of his program is the interaction between these components. We shall describe each of the 3 subject areas separately and then describe how they interact.

Reasoning

- The program makes use of a detailed world model, describing both the current state of the blocks world and its knowledge of procedures for changing that state and making deductions about it.
- There is a data base of simple facts describing what is true at any particular time. E.g. it can contain

```
(IS B1 BLOCK)
(IS B2 PYRAMID)
(AT B1 (LOCATION 100 100 0))
(SUPPORT B1 B2)
(CLEARTOP B2)
(MANIPULABLE B1)
(COLOUR-OF B1 RED)
(IS BLUE COLOUR)
(CAUSE EVENT 23 EVENT 25)
```

- Operating on this data, we have **procedures** written in a abstract language called PLANNER (more precisely, in its reduced version called Micro-Planner). For example, when following the procedure for 'GRASP' the set of currently active goals at a particular point is represented as the following stack.

```
(GRASP B1)
  (GET-RID-OF B2)
    (PUTON B2 TABLE 1)
      (PUT B2 (453 201 0))
      (MOVEHAND (553 301 100))
```

- Note that the subgoal structure provides the basis for asking 'WHY?' questions (e.g. 'Why did you put B2 on the table?' '...to get rid of it').
- 'HOW?' questions are answered by looking at the set of subgoals called directly in achieving a goal, and generating descriptions of the actions involved.
- The way that Winograd's system would deal with a simple description like

  "a red cube which supports a pyramid"

  is shown on the following (somewhat simplified) PLANNER procedure that finds a matching object.

```
(GOAL (IS ?X1 BLOCK))
(GOAL (COLOUR-OF ?X1 RED))
(GOAL (EQUIDIMENSIONAL ?X1))
(GOAL (IS ?X2 PYRAMID))
(GOAL (SUPPORT ?X1 ?X2))
```

  where ? indicates a variable.

– When the object had been found, then it would be incorporated into a command for doing something with the object.
– The procedure does not show loops, conditional tests and other programming details. These would be taken care of by the PLANNER programming language — the loops are implicit in PLANNER's **backtracking** mechanism:
– The description is evaluated by proceeding down the list until some goal fails at which time the system **backs up** automatically to the last point where a decision was made and tries a different possibility.

Semantic Analysis

– requires ways to interpret the meanings of individual words and of the syntactic structures in which they occur.
– Consider how simple words like 'CUBE' are defined:

```
(CUBE
 ((NOUN
   (OBJECT
    ((MANIPULABLE RECTANGULAR)
     ((IS *** BLOCK)
      (EQUIDIMENSIONAL ***)))))))
```

where '***' stands for the particular object which is talked about.
– The first part of the definition is based on the use of **semantic markers**. Thus the system can make quick checks to see whether certain combinations are ruled out by simple tests. E.g., a semantic marker PHYSICAL of an adjective might express the fact that *"this meaning of the adjective applies only to words which represent physical objects"*. The marker PHYSICAL would be applied to some words like *rectangular* and *green* but not to others like *good* and *quick*.

Such markers would give a ready elimination of Chomsky's sentence

"Colourless green ideas sleep furiously."

– and a similar example from SHRDLU:

"Can the table pick up blocks?"

– 'Pick up' demands a subject that is marked ANIMATE whereas 'table' has the incompatible marker INANIMATE.
– Although the above definition of CUBE looks like a static rule statement, such definitions are in effect *calls to procedures* which do the appropriate checks and build the semantic structure.
– Programs need to be flexible: e.g. the phrase 'pick up' has different meanings depending on whether the object is singular or plural:

"Pick up the red block." vs. "Pick up your toys."

- This flexibility is even more important once we get beyond simple words. In defining words like 'the', 'of', 'one', we can hardly make a simple list of properties and descriptions of each word similar to that given above. Rather, for instance, presence of 'one' in a noun-group must trigger a program which considers the previous discourse to see what objects have been mentioned. Various rules and heuristics are applied to determine the appropriate reference. E.g.

'a big red block and a little *one*' {one = red block}

- Words like 'the' are more complex — 'the' refers to a definite article as in:

'*the* Moon' 'Yesterday I met *the* strange man.'

- Thus a model of language must be able to account for the role this type of knowledge plays in understanding. Thus the different possibilities for the meaning of 'the' are procedures which check various facts about the context, then prescribe actions such as:

'look for a unique object in the database which fits the description'

OR

'assert that the object being described is unique as far as the speaker is concerned'

- For example, assuming a pyramid is in the scene, then the following discourse would be acceptable:

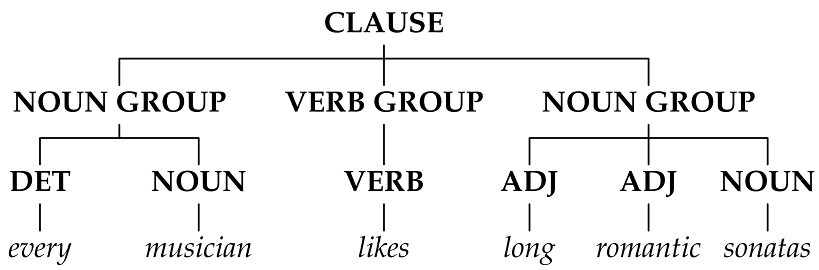
'Grasp a pyramid'
'What is the pyramid supported by?'

Syntax Analysis

- SHRDLU contains a parser and a fairly comprehensive grammar.
- The approach to syntax is based on a belief that the form of syntactic analysis must be usable by a realistic *semantic* system. The emphasis of the resulting grammar differs in several ways from traditional transformational approaches.
- The system first looks for syntactic units which play a primary role in determining meaning — a sentence such as:

'Every musician likes long romantic sonatas'

could be parsed to generate the structure shown below:



– The semantic programs are organised into groups of procedures, each of which is used for interpreting a certain type of unit.

– For each unit, there is a syntactic program (written in a language called PROGRAMMAR, specially designed for the purpose) which operates on the input string to see whether it could represent a unit of that type. In doing this, it will call on other syntactic programs including possibly itself.

– Each unit has a description of the possible ordering of words and other units, e.g. a scheme for a noun group

DETERMINER ORDINAL NUMBER ADJECTIVE* CLASSIFIER* NOUN QUALIFIER*

– The following phrase would be parsed by the above scheme

"the first three big red fire-hydrant covers without handles…"

– Additional constraints: e.g. the indefinite determiner 'a' cannot be followed by an ordinal and a number as in:

"a first three…"

Program Organisation

– Program does **not** operate by first parsing a sentence, then doing some semantic analysis and finally by using deduction produce a response — as Winograd stated:

> *'Language cannot be reduced into separate cases such as syntax, semantics and pragmatics in hopes that by understanding each of them separately, we have understood the whole. The key to the function of language as a means of communication is in the way these areas interact.'*

– The three activities proceed concurrently during the processing of a sentence. As soon as a piece of syntactic structure begins to take shape, a semantic program is called to see whether it makes sense and the answer can direct the parsing. E.g. consider the sentence

"Put the blue pyramid on the block in the box."

– Parser first comes up with 'blue pyramid on the block' as a candidate for a noun group. Semantic analysis is then invoked and since 'the' is definite, a check is made in the data base for the object being referred to. When no such *unique* object is found, the parsing is redirected to 'the block in the box' as a single phrase indicating a location.

– In other cases, the system of semantic markers may reject a possible interpretation on the basis of conflicting category information — thus there is a continuing interplay between the different sets of analysis with the results of one affecting others.

– The procedure as a whole operates in a left to right direction through the sentence. It does not carry multiple possibilities for the syntactic analysis, but instead uses the backtracking facility if the wrong possibility is chosen. In simple sentences like those of the example dialogue, very little backup is ever used, since the combination of syntactic and semantic information usually guides the parser down profitable paths.

Areas of inadequacy

– In Winograd's program, syntax analysis is first invoked and has the responsibility of coming up with possibilities. The semantic routines determine the line of parsing.

– This approach would fail if trying to understand phrases like "Global warming threat". Although the phrase is not syntactically well-formed, it invokes a meaning. So another approach to simulate human understanding of language would be to swap the roles of syntax and semantics: focus on constructing semantic structure from words and sometimes use grammar as a supplementary guide.

– Nevertheless, it seems clear that no single approach is really correct. In contract to the previous example, we are able to interpret sentences syntactically even when we do not know the meaning of individual words. Indeed, most of our vocabulary is acquired in this manner.

– In addition, everyday conversation may be of the form

"Then the other one did the same thing to it."

– Understanding seems to be a coordinated process in which a variety of syntactic and semantic information can be relevant, and in which whatever is more useful in understanding a given part of the sentence is taken advantage of.

– Much remains to be done in understanding how to write programs in which a number of concurrent processes are working in a coordinated fashion without being under the primary hierarchical control of one of them.

– Language *learning* is scarcely touched.

– SHRDLU makes only a *primitive sort of deduction*, whereas humans are continually constructing models and hypotheses in the process of language understanding. We try to understand what the speaker is 'getting at'.

Unit 16. Learning

Learning Outcomes. You should be able to
- define supervised and unsupervised learning, discuss their weaknesses
- list and explain the typical steps of concept learning
- understand examples of geometrical knowledge derivations in Winston's learning approach
- define the term 'inductive learning'
- describe the input, output and the steps of the ID3 algorithm (NO NEED to remember formulas)
- list and explain the advantages and disadvantages of ID3 and inductive learning

The ability to learn must be part of any intelligent system — Simon (1983) has defined learning as:

> *'any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population'*

Approaches to Learning

There are two general approaches to learning :

1. **Supervised learning**
   This approach assumes that the system has a co-operative teacher, and that the teacher will present a succession of samples from which the system can improve its knowledge. The samples will normally include **examples** and **near misses**.
2. **Unsupervised learning**
   The opposite approach is to present all the samples in an unordered fashion through which the system works to produce some order. Large amounts of memory are required for this approach since the samples have to be stored and searched. This does also require large processing times.

The first approach is limited in that the system is only as good as the teacher used. If the teacher fails to communicate, then the system will not learn.

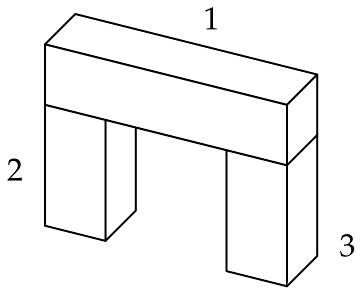
The second approach is only as good as the searching algorithm used, although the system can prepare its own examples and has the ability to learn independently.

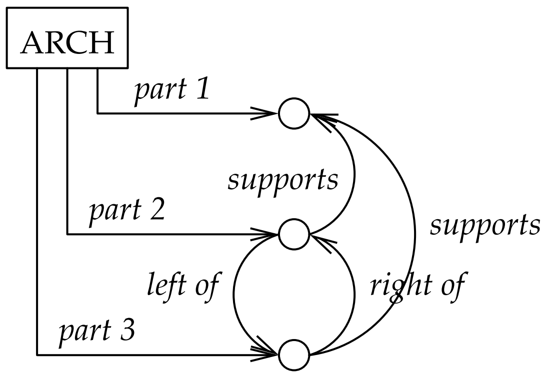
Concept Learning

- Concerned with learning class definitions. The classic example here is the structural concept learning program of P. H. Winston which operated in a simple blocks world.
- Program started with a line drawing of a blocks world structure. It constructed a semantic network representation of the structural description of the objects.
- The basic approach to concept learning can be described as follows:
  1. Begin with an example of the class being examined.
  2. Examine other samples which are also included in the class. **Generalise** the classification rules to include them.
  3. Examine other samples which are near misses. **Restrict** the classification to exclude them.

Winston's approach will now be illustrated on an example:

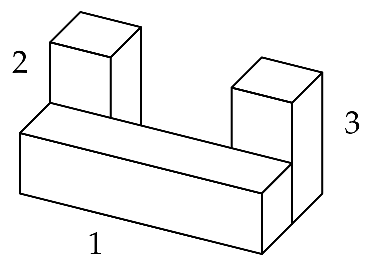
1. First sample (must be a **positive** example)

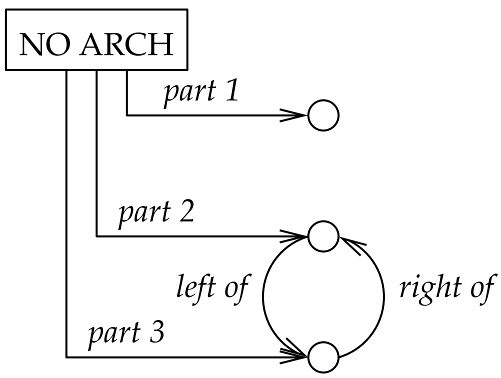




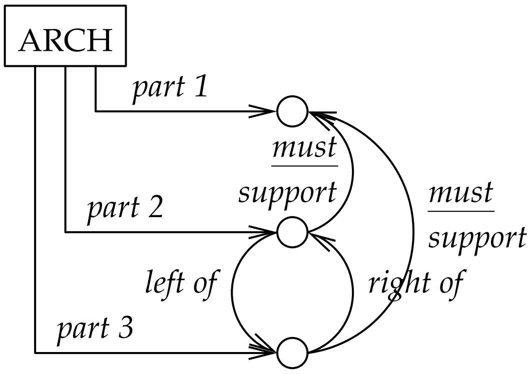
   The derived semantic network (on the right above) is the first attempt at a definition of the 'Arch' concept.

2. Second sample (**near miss**)

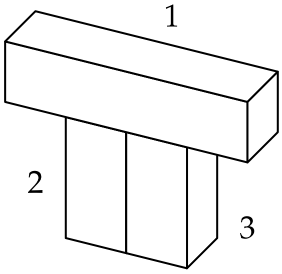


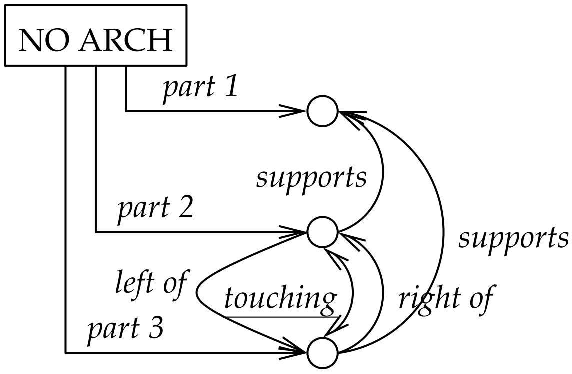


   The concept of support is missing, so assume that this omission caused the failure of the structure to be an example. Hence, adjust the definition of arch to:

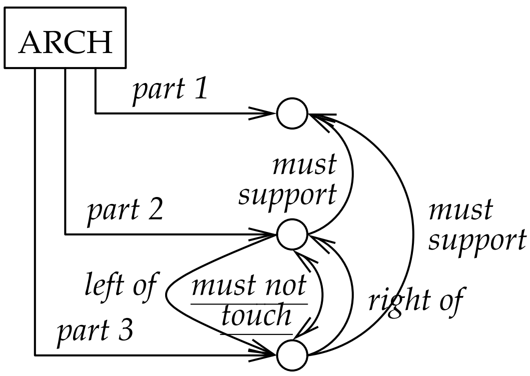


3. Third sample (another **near miss**)

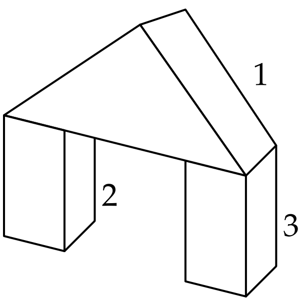




A new concept here is that there must be space between the columns (i.e. they should not touch). Assume, therefore, that no space between parts 2 and 3 has caused the failure. Hence, adjust the definition of arch to:



4. Fourth sample (**positive**)



The derived network is shown in Figure 20 on the left. On the right is a part of the network derived from the first example. At the bottom is a network combining the

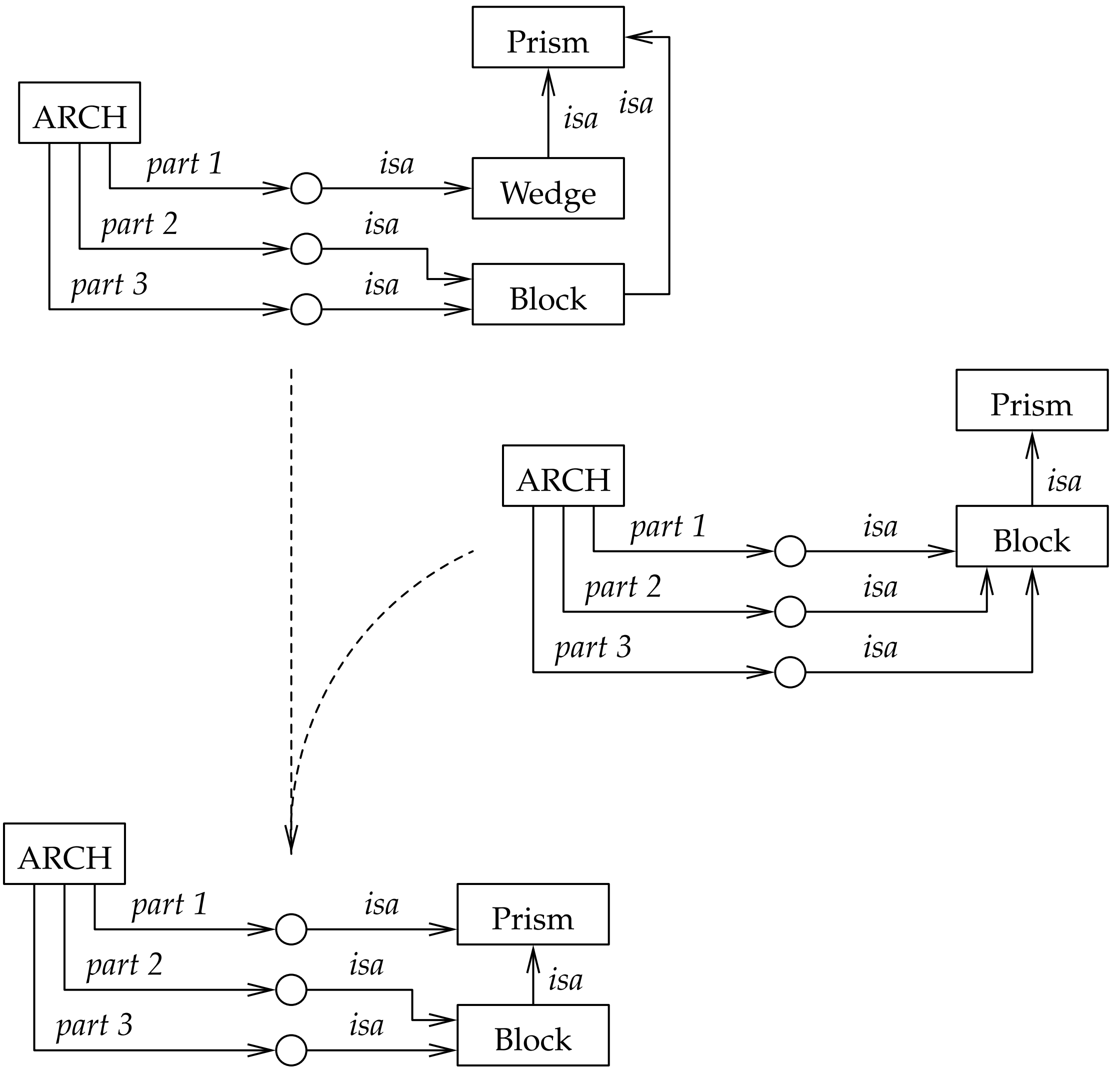
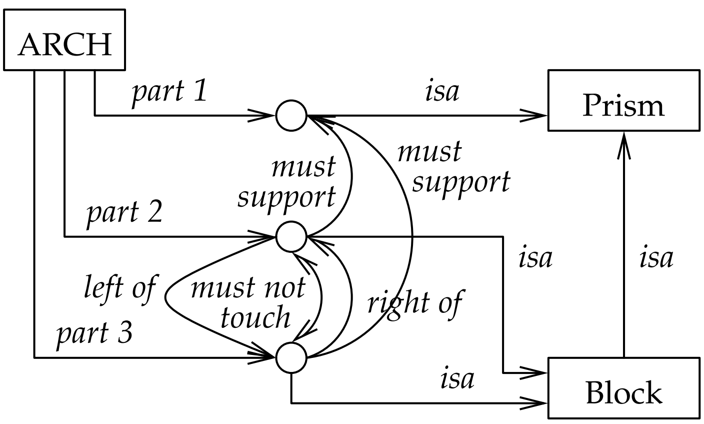


Figure 20: Generalising the shape of the top piece of an arch.

properties part 1 from the networks above it. (The positioning links are not shown, they are as before. The shape information (block) was omitted in the previous drawings.)

We finally arrive at:



Rules of learning approach

1. You cannot learn if you cannot know, i.e. the appropriate representation must be given at the outset.
   (This follows from the way we use positive examples.)
2. You cannot learn if you cannot distinguish the important from the incidental.
   (This follows from the way we use near misses.)

Summary of the above supervised training procedure

– It uses teacher supplied samples (and is therefore no better than the teacher). The samples are either examples (correct) or near misses (incorrect). The procedure is:

  1. The first sample must be an example of the class being investigated, and is used to build an initial description in a frame format.
  2. The second stage is a repeated loop of subsequent samples. These samples will either be examples which tend to expand the set of rules by generalisation, or near misses which tend to constrict the set of rules.

– The supervised learning procedure is an incremental learning procedure, working on a principle that it is easier, hence more reliable, to learn **in small steps**.

Martin's law — you cannot learn anything unless you almost know it already.

Induction Systems

– The most essential component of an Expert System is knowledge. Knowledge acquisition has proved to be the bottleneck of ES development. An expert may be unable to articulate a rule but usually would say "I'll give you an example".

– Given these problems, techniques that can automate the process of knowledge acquisition are appealing. One of these methods is **induction**. It is the process of *reasoning from a given set of specific facts to conclude general principles or rules*.

– Several induction algorithms have been developed — one such is the ID3.

ID3

– It is a general purpose rule induction algorithm developed by Quinlan.

– In the beginning, there are a number of observations, each being either positive or negative. Observations are classified according to a collection of attributes.

– The aim of the algorithm is to **build a decision tree** that can **predict** whether a future observation will be positive or negative.

The ID3 algorithm has the following steps.

1. Select any single attribute A.
2. Categorise the data according to attribute A, setting up a group for each value of the attribute.
3. For each group, compute a number B_v which expresses how much *information about the result* is contained *within the group* (where v stands for the value of attribute A corresponding to this group).

   B_v is **never negative**. The smallest value $B_v = 0$ means that the group contains **perfect information**, i.e. all observations in this group share the same result. The *higher* B_v is the more **variety** of results there is within the group, i.e. the *less* information it contains about the result.

   The number B_v is computed by the following formula:

   $$B_v = -p^+ \cdot \log_2(p^+) - p^- \cdot \log_2(p^-) \tag{1}$$

   where p^+ is the proportion of '**yes**' decisions under the group with value v and p^- is the proportion of '**no**' decisions under the same group.
4. Then, compute the measure M_A of the *information about the result that we gain focusing only on the value of the attribute* A.

   Again, this number expresses the **variety of results** depending on the value of attribute A. Thus $M_A = 0$ means perfect information — by knowing the value of A, we know the result of the observation. The higher M_A it is, the less information about the result we get by knowing the value of the attribute A.

   Number M_A is computed as a weighted sum of numbers B_v for all groups:

   $$M_A = r_{v_1} B_{v_1} + \cdots + r_{v_n} B_{v_n} \tag{2}$$

   where

   – $v_1, \ldots, v_n$ are all of the possible values of attribute A and
   – for every $i = 1, \ldots, n$, the coefficient r_{v_i} is the proportion of the observations made with the value v_i with respect to the total number of observations.
5. Repeat steps 1–4 for all other decision attributes B, C, etc. to get M_B, M_C, etc.
6. Choose the attribute with the **smallest** M and build a level of the decision tree: make a branch for each value of the chosen attribute.

   If all M_A, M_B, etc. are equal to 0, the decision is trivial — it is always 'yes' or always 'no' independently of the values of any of the attributes.
7. If we have made new branches in the previous step, repeat steps 1–6 for each of the branches to select one of the remaining attributes to be used for the next level of branching.

Problem Example

Consider a weather prediction problem in which a small set of observations has been collected which describe the chance of rain given the attributes 'sky', 'barometer' and 'wind'. The following Table summarises the results.

Sky	Barometer	Wind	Rain
clear	rising	north	-
cloudy	rising	south	+
cloudy	steady	north	+
clear	falling	north	-
cloudy	falling	north	+
cloudy	rising	north	+
cloudy	falling	south	-
clear	rising	south	-

The collection of examples above contains four in class + and four in class -. We can use equation 2 to measure the information uncertainty of the entire collection:

$$M_C = -4/8 \cdot \log_2(4/8) - 4/8 \cdot \log_2(4/8) = 1.0$$

Consider firstly the attribute 'wind'. From equation 1 we have

$$B_{\text{north}} = -3/5 \cdot \log_2(3/5) - 2/5 \cdot \log_2(2/5) = 0.971$$
$$B_{\text{south}} = -1/3 \cdot \log_2(1/3) - 2/3 \cdot \log_2(2/3) = 0.918$$

From equation 2, the expected information uncertainty is

$$M_{\text{wind}} = 5/8 \cdot .971 + 3/8 \cdot .918 = 0.951$$

and so the information gained by using this attribute is

$$M_C - M_{\text{wind}} = 1 - 0.951 = 0.049$$

Considering now the attribute 'sky', we have

$$B_{\text{cloudy}} = -4/5 \cdot \log_2(4/5) - 1/5 \cdot \log_2(1/5) = 0.772$$

The group for 'clear' provides no further information uncertainty, and so:

$$M_{\text{sky}} = 3/8 \cdot 0 + 5/8 \cdot .722 = 0.45$$

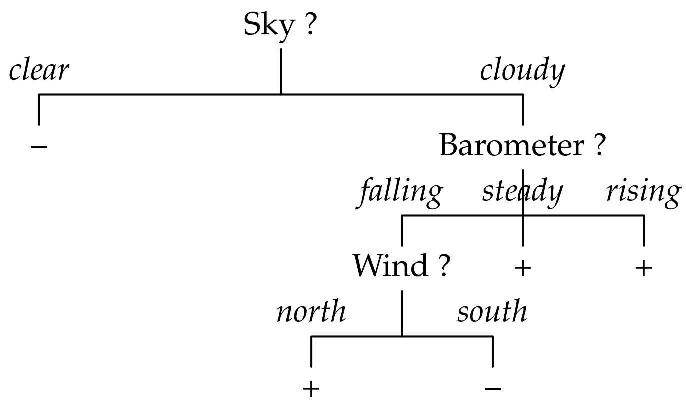
and the information gained by using the attribute 'sky' is

$$M_C - M_{\text{sky}} = 1 - 0.45 = 0.55$$

In a similar manner to the above, we can deduce that the expected information gained by testing the 'barometer' attribute is 0.156.

Therefore, ID3 indicates that the attribute 'sky' should be considered first since it provides the maximum gain in expected information.

The final computed decision tree is the following:



Advantages of ID3

- Selects the most discriminatory attribute first — this enhances system efficiency since it reduces the combinatorial explosion of the decision tree.

Limitations of ID3

- Rules are not probabilistic, therefore:
  - Its decision tree does not inform us how strong the evidence is for its results, in particular:
    - Several identical examples have not much more effect than one example.
  - Cannot deal with contradictory examples.
  - The results are not overly sensitive to small alterations in the training set.

Advantages of induction

- Discovers rules from examples.
- Avoids knowledge elicitation problems.
- Can produce new knowledge.
- Can uncover critical decision factors.
- Can eliminate irrelevant decision factors.
- Can uncover contradictions.

Disadvantages of induction

- Often difficult to choose good decision factors.
- Difficult to understand rules.
- Applicable only for classification problems.

Unit 17. Neural Networks

Learning Outcomes. You should be able to

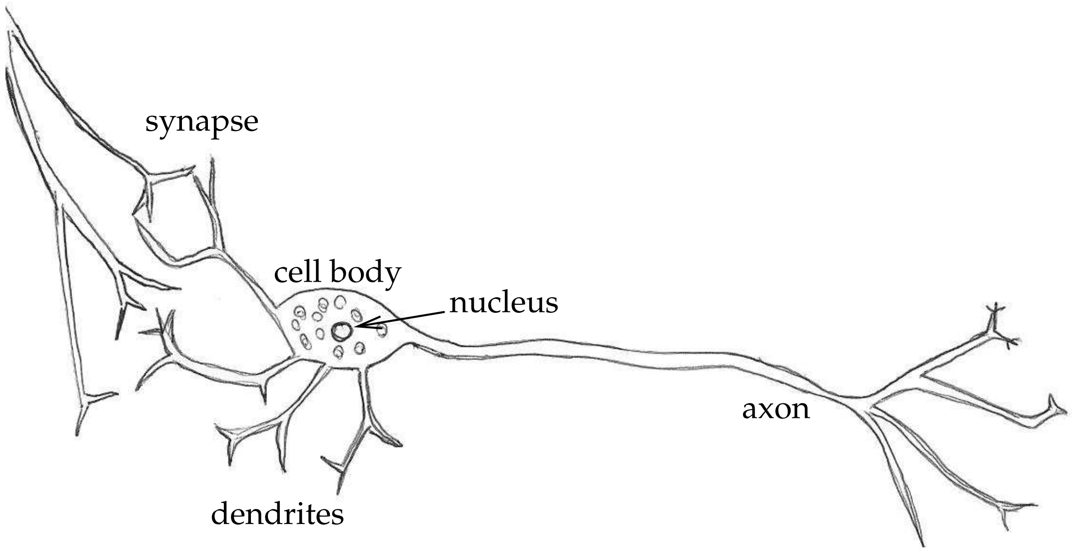
- draw a diagram illustrating the operation of a neuron in Perceptron, explain its function, show an example calculation
- describe the Perceptron training algorithm and apply it on an example
- state what problem was found with Perceptron and how it can be solved
- describe a multi-layer network and its training algorithm, explain the roles of layers, individual neurons, activation function, forward pass and back-propagation (NO NEED to remember formulas)

– Artificial neural networks are biologically inspired.

– Human brain consists of 10^{10}–10^{11} nerve cells, called **neurons**.

– Each neuron is connected to many other neurons which it can influence. It is estimated that there are 10^{15} interconnections over transmission paths that may range up to a metre or more.

– A simplified diagram of a neuron is shown below.



– Inputs from other neurons are received at the synapse where the signals are conducted to the cell body. There they are summed, some inputs tending to excite the cell, others tending to inhibit it — when the cumulative excitation in the cell body exceeds a threshold, the cell **fires**, sending a signal down the axon to the dendrites of other neurons where they receive signals at the synaptic region.

– McCulloch and Pitts (1943, 1947) suggested a simple model, an artificial neuron, which accounted for most of the main properties of the natural neuron.

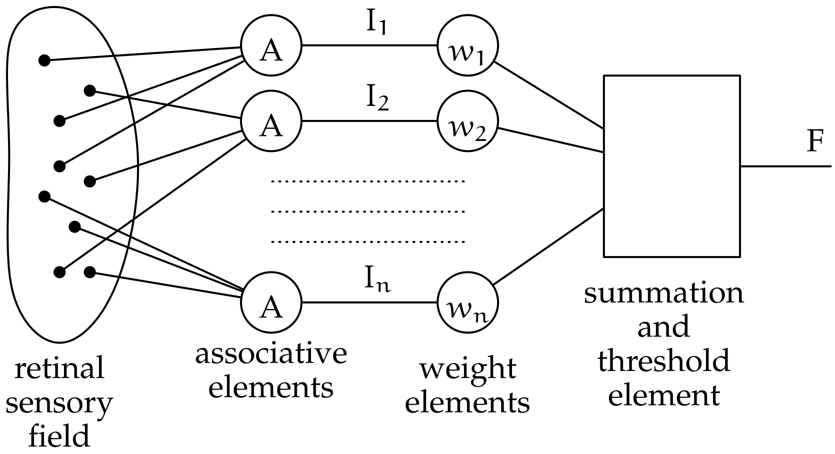


Figure 21: Schematic operation of a neuron connected to retina.

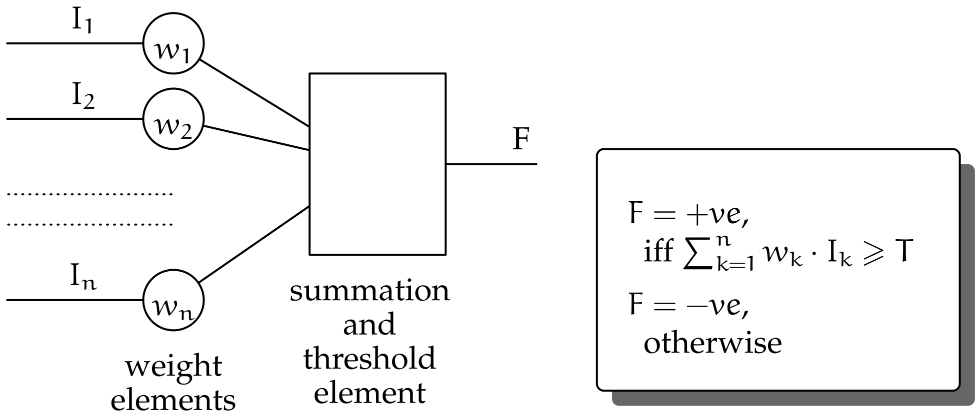


Figure 22: Schematic operation of an artificial neuron in Rosenblatt's Perceptron.

- It was left to Rosenblatt (1962) to actually construct a machine called the **Perceptron** which was based on artificial neurons. He also described a training algorithm through which the network could learn.
- Rosenblatt's work provided a great impetus to research in the field.
- The 'learning' ability is a neural network's most intriguing property. The network can modify itself as a result of experience to produce a more desirable pattern.
- Learning can be either supervised or unsupervised. (See Unit 16)
- **supervised**: the teacher evaluates the behaviour of the system and directs the subsequent modification. In practice, both the input layer and the output layer of the network have their states **clamped** (fixed) during the learning phase, and the network instructed on the mapping to be performed.
- **unsupervised**: the network has no knowledge of the correct answer and thus cannot know exactly what the correct response should be. The network acts as a regularity detector as it tries to determine the underlying structure of the input vector set.
- Perceptron learning is of the supervised type.

- Rosenblatt provided a Perceptron training algorithm.
- A Perceptron is trained by presenting a set of patterns, the **training set**, to its inputs one at a time, and adjusting the weights until the desired output occurs for each member of the training set.
- A 'trained' Perceptron can then be used to predict likely outcomes from similar sets of data.

Training algorithm

1. Data collected and classified into 2 groups — data coded.
2. Initialise all weight vectors (usually, set to zero).
3. Take next line of data, form the sum

$$\sum_{k=1}^{n} I_k \cdot w_k = I_1 \cdot w_1 + I_2 \cdot w_2 + I_3 \cdot w_3 + \cdots + I_n \cdot w_n$$

4. If the sign of the computed response (that is, $\sum_{k=1}^{n} I_k \cdot w_k$) equals the sign of the desired response, then go to step 5.

   If there is a discrepancy in the responses then the weight vectors must be changed.

   If the desired response is *negative*, then all the weight vectors w_k (for $k = 1, \ldots, n$ for which $I_k \neq 0$) are *decremented* by 1.

   If the desired response is *positive*, then the all the weight vectors w_k (for $k = 1, \ldots, n$ for which $I_k \neq 0$) are *incremented* by 1.
5. If all the data points have been classified correctly, the training is complete; otherwise go to step 3.

An example run of the Perceptron training algorithm is shown in Table 1.

Problems with the Perceptron approach

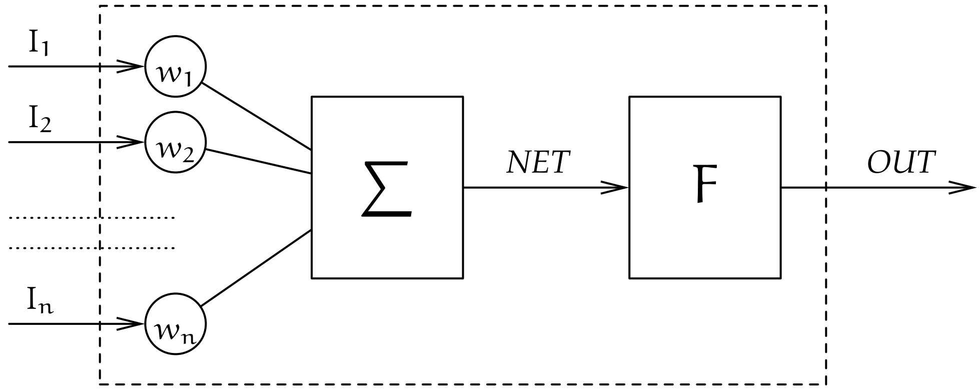
- The Perceptron approach was questioned by two prominent AI researchers, Minsky and Papert in their book 'Perceptrons'.
- In a rigorous theoretical treatment of Perceptrons, they proved that such networks could not solve such simple problems as the exclusive or (XOR). I.e. a network cannot be trained to accept samples that contain either one sub-pattern *or* another sub-pattern but *not both* of them. For example, it is impossible to train a network to accept inputs that have $I_1 = 1$ or $I_2 = 1$ and, at the same time, reject patterns that have $I_1 = I_2$.
- This serious limitation can be overcome by adding *more layers* of neurons to the single-layer network of the Perceptron.
- A number of training algorithms have been suggested for such networks.

I_1	I_2	I_3	I_4	w_1	w_2	w_3	w_4	$\sum w_k I_k$	Perc	Des	Adj	w'_1	w'_2	w'_3	w'_4
0	1	1	1	0	0	0	0	0	*	−	YES	0	−1	−1	−1
0	0	0	1	0	−1	−1	−1	−1	−	+	YES	0	−1	−1	0
1	0	0	1	0	−1	−1	0	0	*	+	YES	1	−1	−1	1
1	0	1	1	1	−1	−1	1	1	+	+	NO	1	−1	−1	1
0	0	1	1	1	−1	−1	1	0	*	−	YES	1	−1	−2	0
1	1	0	1	1	−1	−2	0	0	*	+	YES	2	0	−2	1
1	1	1	1	2	0	−2	1	1	+	−	YES	1	−1	−3	0
0	1	0	1	1	−1	−3	0	−1	−	−	NO	1	−1	−3	0
Iteration 2															
0	1	1	1	1	−1	−3	0	−4	−	−	NO	1	−1	−3	0
0	0	0	1	1	−1	−3	0	0	*	+	YES	1	−1	−3	1
1	0	0	1	1	−1	−3	1	2	+	+	NO	1	−1	−3	1
1	0	1	1	1	−1	−3	1	−1	−	+	YES	2	−1	−2	2
0	0	1	1	2	−1	−2	2	0	*	−	YES	2	−1	−3	1
1	1	0	1	2	−1	−3	1	2	+	+	NO	2	−1	−3	1
1	1	1	1	2	−1	−3	1	−1	−	−	NO	2	−1	−3	1
0	1	0	1	2	−1	−3	1	0	*	−	YES	2	−2	−3	0
Iteration 3															
0	1	1	1	2	−2	−3	0	−5	−	−	NO	2	−2	−3	0
0	0	0	1	2	−2	−3	0	0	*	+	YES	2	−2	−3	1
1	0	0	1	2	−2	−3	1	3	+	+	NO	2	−2	−3	1
1	0	1	1	2	−2	−3	1	0	*	+	YES	3	−2	−2	2
0	0	1	1	3	−2	−2	2	0	*	−	YES	3	−2	−3	1
1	1	0	1	3	−2	−3	1	2	+	+	NO	3	−2	−3	1
1	1	1	1	3	−2	−3	1	−1	−	−	NO	3	−2	−3	1
0	1	0	1	3	−2	−3	1	−1	−	−	NO	3	−2	−3	1
Iteration 4															
0	1	1	1	3	−2	−3	1	−4	−	−	NO	3	−2	−3	1
0	0	0	1	3	−2	−3	1	1	+	+	NO	3	−2	−3	1
1	0	0	1	3	−2	−3	1	4	+	+	NO	3	−2	−3	1
1	0	1	1	3	−2	−3	1	1	+	+	NO	3	−2	−3	1
0	0	1	1	3	−2	−3	1	−2	−	−	NO	3	−2	−3	1

Table 1: Example Perceptron training.
Legend: Perc = (perceived) network's result (sign), Des = desired result (sign),
Adj = weights adjusted, w'_1 w'_2 w'_3 w'_4 = new weights

Multilayer networks and back-propagation

- Invention of the back-propagation algorithm has played a large part in the resurgence of interest in artificial neural networks. Back-propagation allows multilayer networks to be trained in a similar way to Perceptron networks.
- The model of the neuron used as a fundamental building block for multilayer networks is shown below.

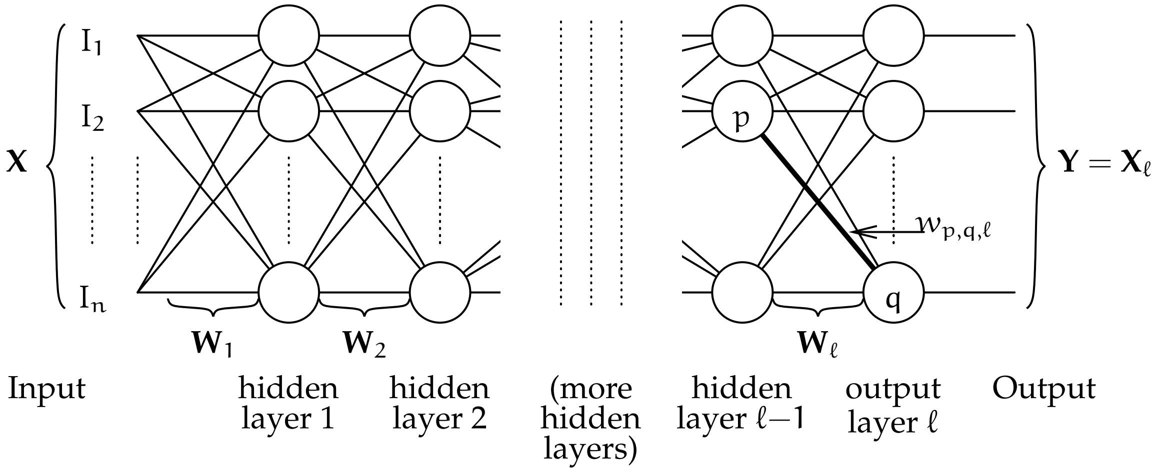


$$NET = \sum_{k=1}^{n} I_k \cdot w_k = I_1 \cdot w_1 + I_2 \cdot w_2 + I_3 \cdot w_3 + \cdots + I_n \cdot w_n$$

- after *NET* is calculated, an **activation function** F is applied to modify it, thereby producing the signal *OUT*. The purpose of the activation function is to "squash" the *NET* values so that *OUT* would always fall within some numerical range, usually between 0 and 1.
- The activation function usually used is

$$OUT = \frac{1}{1 + e^{-NET}}$$

Its graph in Figure 23 illustrates its "squashing" effect.
- Back-propagation can be applied to networks with any number of layers.
- In training a network, the weights are adjusted so that application of a set of inputs produces the desired set of outputs.
- A multilayer network suitable for training with back-propagation is shown below.



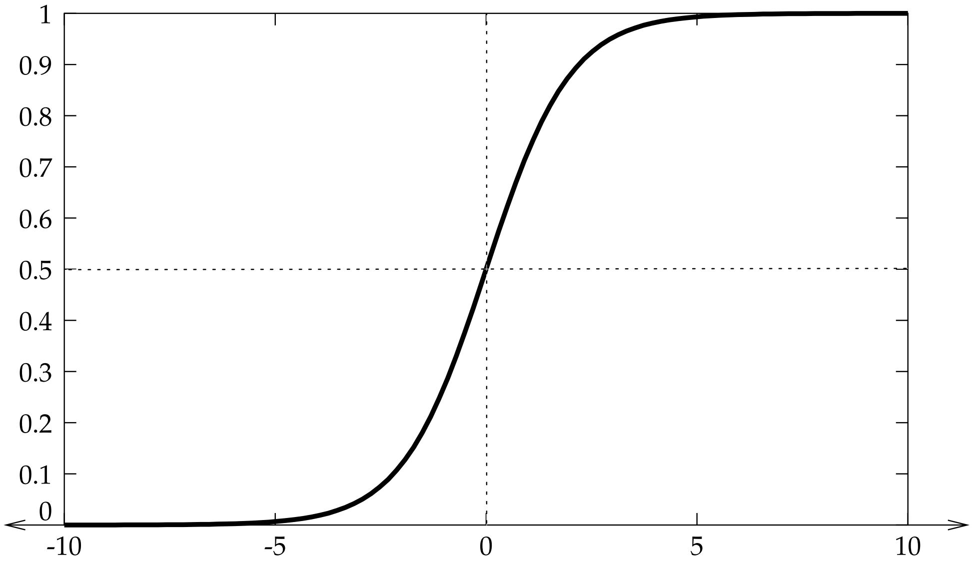


Figure 23: A typical activation function.

An overview of Training

1. *Initialise* the weights to small *random* numbers.
2. Select the next training pair (comprising input and output vectors) from the training set; apply the input vector to the network input.
3. Calculate the output of the network.
4. Calculate the error between the network output and the desired output.
5. Adjust the weights of the network in a way that minimises the error.
6. Repeat steps 2–5 for each vector in the training set until the error for the entire set is acceptably low.
7. When an acceptable value for the error is reached, the network is said to be trained. The weight values now remain constant — the network can then be used for recognition.

Steps 2 and 3 above constitute a **forward** pass through the network. An input vector $\mathbf{X} = I_1, \ldots, I_n$ is applied and some output vector $\mathbf{Y}$ is produced. The training set provides the input vector $\mathbf{X}$ and the desired output $\overline{\mathbf{Y}}$.

Calculation in multilayer networks is performed layer by layer, starting at the layer nearest to the inputs. The first layer transforms the input vector $\mathbf{X}$ into another vector of outputs $\mathbf{X}_1$. The second layer takes $\mathbf{X}_1$ as input and produces $\mathbf{X}_2$, etc.

The *NET* value of each neuron p is calculated on the weighted sum of the inputs of p. The activation function F then "squashes" *NET* to produce the *OUT* value of the neuron p. The "squashing" needs to be performed so that *OUT* is within a certain numerical range (usually between 0 and 1).

This calculation is done for all neurons in the current layer before proceeding to the next layer. It can be expressed using vectors and matrices. The weights between two neighbouring layers j and k can be considered to be a matrix $\mathbf{W}_k$. E.g. the weight from neuron 5 in layer 2 to neuron 7 in layer 3 is $w_{5,7,3} = (\mathbf{W}_3)_{5,7}$. Now we can express the calculation that happens across a layer k as follows:

$$\mathbf{X}_k = F(\mathbf{X}_{\mathbf{k-1}} \times \mathbf{W}_k) =$$
$$F\big(w_{1,1,k} \cdot x_{1,k-1} + \cdots + w_{n,1,k} \cdot x_{n,k-1}\big), \ldots, F\big(w_{1,n,k} \cdot x_{1,k-1} + \cdots + w_{n,n,k} \cdot x_{n,k-1}\big)$$

The output vector for one layer is the input vector for the next, so calculating the outputs of the final layer requires the application of the above equation to each layer, from the first hidden layer to the output layer.

Adjusting the weights of the output layer

Steps 4 and 5 in the training process constitute a reverse pass through the network.

For each neuron in the output layer, we have a target value for the response. Consequently, adjusting the weights in the output layer is easily accomplished. However, training the interior layers, the **hidden layers**, is more complicated as their outputs have no target values for comparison.

The difference ∂, the adjustment to be made to each output neuron, is calculated as follows

$$\partial = OUT \cdot (1 - OUT) \cdot ERROR \tag{3}$$

where *ERROR* is the difference between the calculated output and the desired output, thus

$$\partial = OUT \cdot (1 - OUT) \cdot (TARGET - OUT) \tag{4}$$

The modification to be made to the weight $w_{p,q,\ell}$ between a neuron p in the last hidden layer $\ell - 1$ and neuron q in the output layer ℓ is

$$\Delta w_{p,q,\ell} = \eta \cdot \partial_{q,\ell} \cdot OUT_{p,\ell-1} \tag{5}$$

and the new weight vector is

$$w^{\text{NEW}}_{p,q,\ell} = w^{\text{OLD}}_{p,q,\ell} + \Delta w_{p,q,\ell} \tag{6}$$

Note that subscripts p and q refer to a specific neuron, whereas subscripts $\ell - 1$ and ℓ refer to a layer. The other symbols used above are explained below:

$\partial_{q,\ell} =$	the value of ∂ for neuron q in the output layer ℓ.
$OUT_{p,\ell-1} =$	the value of *OUT* for neuron p in the hidden layer $\ell - 1$.
$\eta =$	constant **training rate coefficient** (typically between 0.01 and 1.0)
$w^{\text{OLD}}_{p,q,\ell} =$	value of the weight from neuron p in the hidden layer to neuron q in the output layer before adjustment
$w^{\text{NEW}}_{p,q,\ell} =$	the value of the weight after adjustment

Adjusting the weights of the hidden layers

– Hidden layers have no target vector, so the training process described above cannot be used.

– Equations 5 and 6 are used for all layers, both output and hidden. However, for hidden layers, ∂ must be generated without the benefit of a target vector.

– To accomplish this, ∂ is first of all calculated for each neuron in the output layer, as in equation 4. It is then used to adjust the weights feeding into the output layer. These values of δ are then *propagated back* through *the same weights* to generate a value for ∂ for each neuron in the preceding layer.

– These values of ∂ are used, in turn, to adjust the weights of the hidden layer and, in a similar way, are propagated back to all preceding layers.

– Let us consider this process in a little more detail.

– Consider a single neuron in the hidden layer just before the output layer.

– In the forward pass, this neuron propagates its output value to neurons in the output layer through the interconnecting weights.

– During training, these weights operate in reverse, passing the value of ∂ from the output layer back to the hidden layer — each of these weights is multiplied by the ∂ value of the neuron to which it connects in the output layer.

– The value of ∂ needed for the hidden-layer neuron is produced by the following expression:

$$\partial_{p,j} = OUT_{p,j} \cdot (1 - OUT_{p,j}) \cdot \left(\sum_q \partial_{q,k} \cdot w_{p,q,k} \right) \tag{7}$$

– With ∂ having been calculated, the weights feeding the first hidden layer can be adjusted using equations 5 and 6, modifying indexes to indicate the correct layers.

– For each neuron in a given hidden layer, ∂'s must be calculated, and all weights associated with that layer must be adjusted. This is repeated moving back toward the input layer by layer, until all weights are adjusted.

Applications

1. Optical character recognition
2. Machine recognition of hand-written characters (e.g. post codes)
3. Text-to-speech systems — NetTalk (1987)
4. Visual perception
5. Financial market predictions

Unit 18. Genetic Algorithms (GAs)

Learning Outcomes. You should be able to

- describe the initialisation, main loop and termination of a genetic algorithm
- describe and follow the steps of a simple generation cycle
- understand and explain a given simple example of the generation cycle
- list four main parameters of a genetic algorithm and describe how their value affects the run of the algorithm
- list a few typical application areas of genetic algorithms

- GAs are search algorithms based on the mechanism of natural selection — a Darwinian survival of the fittest among "string creatures".
- Developed by John Holland and co-workers by 1975 at the University of Michigan.
- GAs provide a robust search in complex spaces.
- They have generated a great deal of research interest in recent years as the scope of application and performance of GAs has come to be recognised.

A simple genetic algorithm

This can be surprisingly simple, involving nothing more complex than copying strings and swapping string portions. Despite this simplicity, the algorithm is quite powerful.

A simple genetic algorithm that yields good results for many practical problems contains the following operators:

(i) fitness evaluation,
(ii) selection of mates,
(iii) crossover,
(iv) mutation.

Modelling assumptions

- Research has defined a fairly general framework for the specification of genetic algorithms.
- Each point in the problem space can be considered as an individual, represented uniquely within the system by a fixed length *string of symbols*. Usually the binary symbols 0 and 1 are used, although other representations (e.g. floating point) have been investigated.

– This string serves as the 'genetic material' with specific positions (**loci**) on the string (**chromosome**) containing unique symbols or string features (**genes**) taking on values (**alleles**). (The parenthesised words in the previous sentence are notions used in genetics that inspired GAs.)
– The system always maintains a **population** of strings representing the current set of possible solutions to the problem.
– The process begins either by random generation or designer specification of an **initial population**.
– Time is measured in discrete steps called **generations**. During every generation step, a new population is produced from the previous population.
– Each string can be assigned a **fitness value** which is indicative of its worth in the given environment and also indicative of its likely survival in future generations.

Basic Execution Cycle

The basic (simple version of the) execution cycle is as follows (one run of the cycle corresponds to one generation step):

1. In the current population, **evaluate the fitness** of every solution (string).
2. Randomly **select candidates** for mating. Solutions with a higher fitness have a greater probability of being selected than solutions with lower fitness.
3. Randomly **select pairs** from the candidate solutions.
4. Produce offspring for each pair using the **crossover** operator. All offspring produced in this step form the next population, replacing their parents.

   Crossover is an exchange of a portion of genetic material between the two mates. It involves **selecting a site** within the chromosome string where the portion of genetic material is located that should be exchanged. The selection of site may be random too.
5. Apply the **mutation** operator, i.e. make small changes to the new population at random.
6. **Iterate** the process by returning to step 1 above or **terminate** if success has been achieved or a specified number of generations has been exceeded.

A simple example problem

– Consider the toy 'problem' of maximising the function $f(x) = x^2$ in the integer range 0 to 31, i.e. searching for an integer between 0 and 31, inclusive, which has the largest square.
– We first decide how to represent x as a finite-length string. Clearly, one possible representation is as a string of 5 bits. (A bit is either 0 or 1.) E.g. the string 10101 represents the number

$$\underline{1} \times 16 + \underline{0} \times 8 + \underline{1} \times 4 + \underline{0} \times 2 + \underline{1} \times 1 = 16 + 4 + 1 = 21.$$

Number i	String	Value ν_i	Fitness $f_i = \nu_i^2$	Select % $f_i/\sum_k f_k$
1	01101	13	169	14%
2	11000	24	576	49%
3	01000	8	64	6%
4	10011	19	361	31%
		Sum	1170	100%
		Average	293	25%
		Maximum	576	49%

Table 2: An initial population with fitness values and population statistics.

- A genetic algorithm starts with a population of strings picked at random and then generates successive populations through fitness evaluation, mate selection, crossover and mutation.
- Assume that the initial population of four strings shown in Table 2 has been derived by random means.

  The table shows also the fitness of each string. In this case, the fitness is the value of the function which we want to maximise, i.e. the square of the number that the string represents.

  The fitness of each individual is also shown as a percentage of the sum of fitness over the whole population.

  Finally, a statistics of the whole population is included.
- The initial population is now subjected to the process of selection. We select as many candidates as is the size of the new population. One and the same string may be selected *more than once.* Strings with a *higher fitness* value have a *higher probability* of being selected.

  In our scenario, we select four strings to maintain the same population size. We assume the most likely pick: string number 2 is picked twice, strings number 1 and 4 are picked once and string number 3 is not picked at all.
- Next, the pairs are formed. Here, we mate all candidates, so we select two pairs. In this case, they happen to be: 2 with 1 and 2 with 4.
- In this example, we shall consider **one-point crossover** in which a position k along the string is selected at random. Thus the position k is an integer between 1 and $(L-1)$, inclusive, where L is the string length. Two new strings are created by swapping all characters from position $k+1$ till the end of the string. For example, from the pair of strings below on the left with $k=2$, the pair of offspring strings on the right would be produced:

$$01\underline{101} \text{ and } 11\underline{000} \longrightarrow 01\underline{000} \text{ and } 11\underline{101}$$

  In our scenario, $L=5$ and k may be any number between 1 and 4. For $k=2$ (as above) we swap the string portions starting from position $2+1=3$ until the end, i.e. the portion at positions $3,4,5$.

number	original string	chosen mate	crossover site	offspring string	value v_i	fitness $f_i = v_i^2$
1	$0110\underline{1}$	2a	4	01100	12	144
2a	$1100\underline{0}$	1	4	11001	25	625
2b	$11\underline{000}$	4	2	11011	27	729
4	$10\underline{011}$	2b	2	$100\underline{1}0$	18	324
					Sum	1822
					Average	455
					Maximum	729

Table 3: Selected pairs and their offspring forming a new population.

- The last operator, mutation, is performed on a bit-by-bit basis. This simply changes a bit in some randomly chosen string at some random position.
- We assume that the probability of mutation for this problem is 0.01 — e.g. with 20 transferred bit positions we would expect the average of $20 \times 0.01 = 0.2$ bits to undergo mutation during one generation.

  In our example of a generation step, we assume that one bit underwent mutation. In Table 3, the mutated bit is underlined.

  With the given probability, we would expect no mutation to take place in most generations. Typically, every one in a few generations a mutation takes place in one or more bits within the population.
- Table 3 shows the selected strings, the pairs, as well as their offspring created by crossover and mutation. At the bottom, the fitness statistics of the new population is stated showing an improvement over the previous one.
- This process will then continue until a satisfactory result is obtained or until a specified number of iterations has been exceeded.

The first two generations of the GA cycle for the problem of maximising $f(x) = x^2$ are summarised in Figure 24.

Genetic Parameters

Let us consider the general effect of varying the genetic parameters:

Population Size (N)

- affects both the global performance and the efficiency of the genetic algorithm.
- With small populations, the algorithm usually performs poorly because the population provides an insufficient coverage of the problem space.
- A large population is more likely to be representative of the entire problem domain and will also tend to converge to global instead of local solutions.

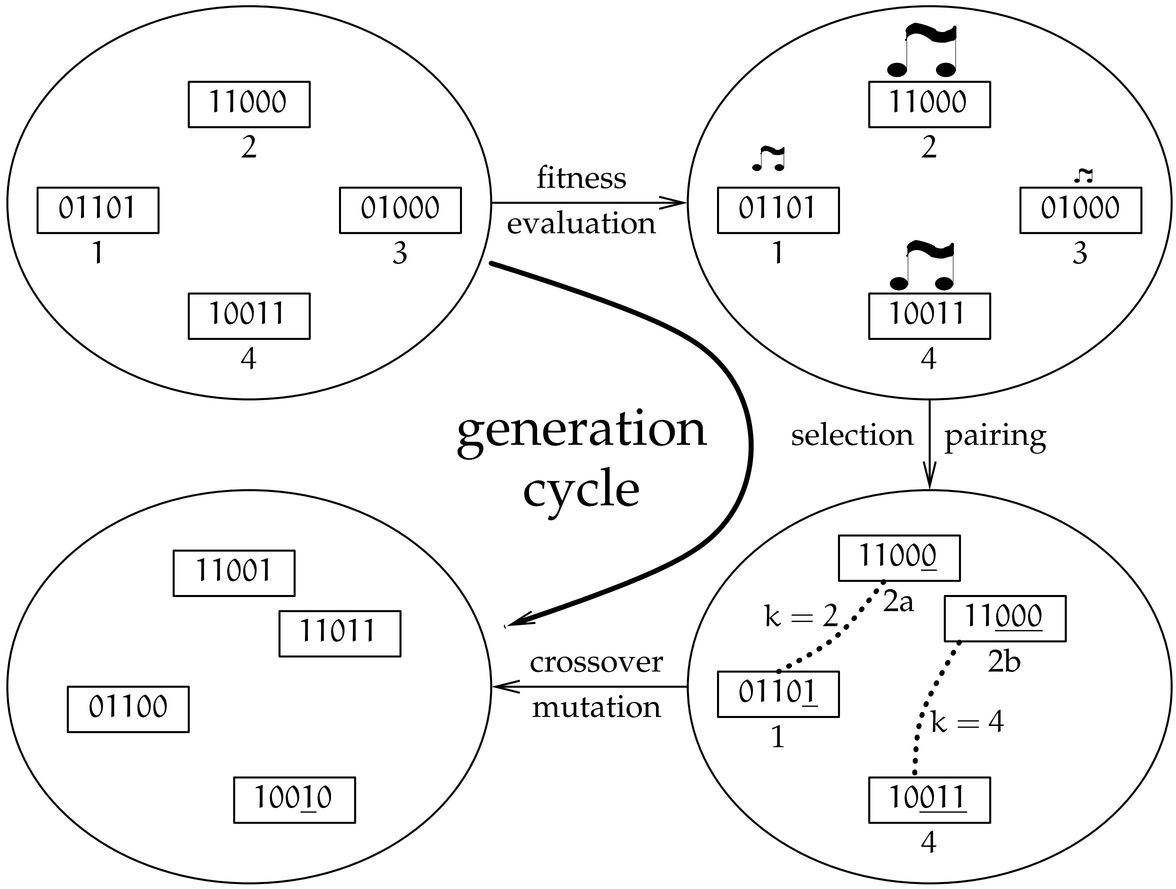


Figure 24: Example of a generation cycle

Crossover rate (C)

- The frequency with which the crossover operator is applied is controlled by the crossover rate. In each new population, the average number of $N \times C$ structures undergo crossover.
- The higher the crossover rate the more quickly new structures are introduced to the population.
- If the crossover rate is too high, then good performance structures are removed faster than selection can produce improvements.
- A low crossover on the other hand may stagnate the search.

Mutation Rate (M)

- Mutation is random — each fundamental unit (bit, position, or token) in a structure has a certain (usually small) probability of changing.
- A large mutation rate results in essentially random search.
- Approximately $M \times N \times L$ mutations occur per generation where L is the structure length.

Generation Gap (G)

- The generation gap controls the percentage of the population to be replaced during each generation.
- $N \times G$ structures of a population are chosen to be replaced in the next population. A value of $G = 1.0$ means that the entire population is replaced during each generation.

Note

There are many variations on the basic genetic algorithm. For example, several crossover operators have been investigated such as two-point crossover, uniform crossover.

Maintaining population diversity during evolution is important and the parameters should be chosen so as to accomplish this, attaining so-called "**genetic diversity**."

A **super-performer** is an individual with relatively high fitness that dominates the population until no further improvement is possible. Such a super-performer may appear early in the process. This so-called "**genetic plateau**" should be avoided so that the problem space is sufficiently explored.

Applications

- Optimisation of complicated functions, scheduling
- Computer vision — object recognition in brightness arrays
- Problem-solving
- Neural networks — topology design, training
- Concept learning
- Genetic programming — computer writing programs