1. What is Artificial Intelligence?

Data: Raw facts, unformatted information.

Information: It is the result of processing, manipulating and organizing data in response to a specific need. Information relates to the understanding of the problem domain.

Knowledge: It relates to the understanding of the solution domain – what to do?

Intelligence: It is the knowledge in operation towards the solution – how to do? How to apply the solution?

Artificial Intelligence: Artificial intelligence is the study of how make computers to do things which people do better at the moment. It refers to the intelligence controlled by a computer machine.

One View of AI is

- About designing systems that are as intelligent as humans
- Computers can be acquired with abilities nearly equal to human intelligence
- How system arrives at a conclusion or reasoning behind selection of actions
- How system acts and performs not so much on reasoning process.

Why Artificial Intelligence?

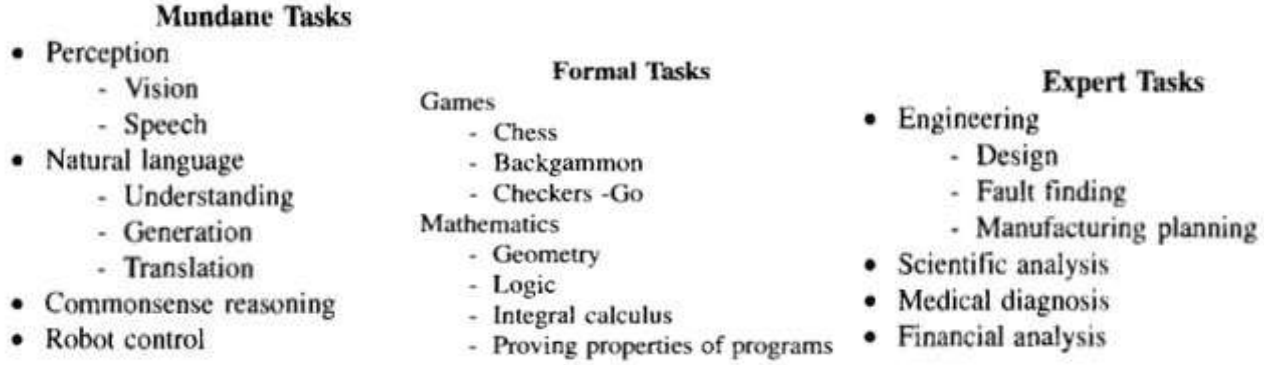
- ❑ Making mistakes on real-time can be costly and dangerous.
- ❑ Time-constraints may limit the extent of learning in real world.

The AI Problem

There are some of the problems contained within AI.

1. *Game Playing and theorem proving* share the property that people who do them well are considered to be displaying intelligence.
2. Another important foray into AI is focused on *Commonsense Reasoning*. It includes reasoning about physical objects and their relationships to each other, as well as reasoning about actions and other consequences.
3. To investigate this sort of reasoning Nowell Shaw and Simon built the *General Problem Solver (GPS)* which they applied to several common sense tasks as well as the problem of performing symbolic manipulations of logical expressions. But no attempt was made to create a program with a large amount of knowledge about a particular problem domain.

4. The following are the figures showing some of the tasks that are the targets of work in AI:



Only quite simple tasks were selected.

Perception of the world around us is crucial to our survival. Animals with much less intelligence than people are capable of more sophisticated visual perception. Perception tasks are difficult because they involve analog signals. A person who knows how to perform tasks from several of the categories shown in figure learns the necessary skills in standard order.

First perceptual, linguistic and commonsense skills are learned. Later expert skills such as engineering, medicine or finance are acquired.

Physical Symbol System Hypothesis

At the heart of research in artificial intelligence, the underlying assumptions about intelligence lie in what Newell and Simon (1976) call the physical symbol system hypothesis. They define a physical symbol system as follows:

1. Symbols
2. Expressions
3. Symbol Structure
4. System

A physical symbol system consists of a set of entities called symbols, which are physically patters that can occur as components of another type of entity called an expression (or symbol structure). A symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way. At any instance of the time the system will contain a collection of these symbol structures. The system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction.

They state hypothesis as:

"***A physical symbol system has the necessary and sufficient means for general 'intelligent actions'."***

This hypothesis is only a hypothesis there appears to be no way to prove or disprove it on logical ground so, it must be subjected to empirical validation we find that it is false. We may find the bulk of the evidence says that it is true but only way to determine its truth is by experimentation"

Computers provide the perfect medium for this experimentation since they can be programmed to simulate physical symbol system we like. The importance of the physical symbol system hypothesis is twofold. It is a significant theory of the nature of human intelligence and so is of great interest to psychologists.

What is an AI Technique?

Artificial Intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard. There are techniques that are appropriate for the solution of a variety of these problems. The results of AI research tells that

Intelligence requires Knowledge. Knowledge possesses some less desirable properties including:

- *It is voluminous*
- *It is hard to characterize accurately*
- *It is constantly changing*
- *It differs from data by being organized in a way that corresponds to the ways it will be used.*

AI technique is a method that exploits knowledge that should be represented in such a way that:

- The knowledge captures generalizations. In other words, it is not necessary to represent each individual situation. Instead situations that share important properties are grouped together.
- It can be understood by people who must provide it. Most of the knowledge a program has must ultimately be provided by people in terms they understand.
- It can be easily be modified to correct errors and to reflect changes in the world and in our world view.
- It can be used in a great many situations even if it is not totally accurate or complete.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

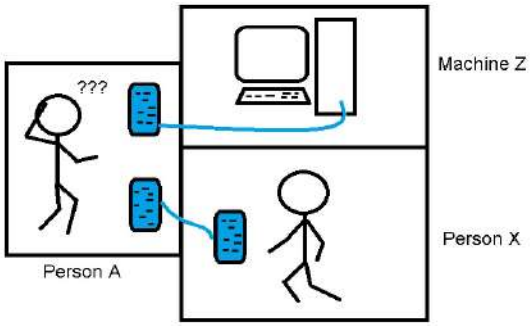
It is possible to solve AI problems without using AI techniques. It is possible to apply AI techniques to solutions of non-AI problems.

Important AI Techniques:

- Search: Provides a way of solving problems for which no more direct approach is available as well as a framework into which any direct techniques that are available can be embedded.
- Use of Knowledge: Provides a way of solving complex problems by exploiting the structures of the objects that are involved.
- Abstraction: Provides a way of separating important features and variations from the many unimportant ones that would otherwise overwhelm any process.

Criteria for Success (Turing Test)

In 1950, Alan Turing proposed the method for determining whether a machine can think. His method has since become known as the "Turing Test". To conduct this test, we need two people and the machine to be evaluated. Turing Test provides a definition of intelligence in a machine and compares the intelligent behavior of human being with that of a computer.



One person A plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask set of questions to both the computer Z and person X by typing questions and receiving typed responses. The interrogator knows them only as Z and X and aims to determine who the person is and who the machine is.

The goal of machine is to fool the interrogator into believing that it is the person. If the machine succeeds we conclude that the machine can think. The machine is allowed to do whatever it can do to fool the interrogator.

> *For example, if asked the question "How much is 12,324 times 73,981?" The machine could wait several minutes and then respond with wrong answer.*

The interrogator receives two sets of responses, but does not know which set comes from human and which from computer. After careful examination of responses, if interrogator cannot definitely tell which set has come from the computer and which from human, then *the computer has passed the Turing Test*. The more serious issue is the amount of knowledge that a machine would need to pass the Turing test.

Overview of Artificial Intelligence

MODULE-1

It was the ability of electronic machines to store large amounts of information and process it at very high speeds that gave researchers the vision of building systems which could emulate (imitate) some human abilities.

We will see the introduction of the systems which equal or exceed human abilities and see them because an important part of most business and government operations as well as our daily activities.

Definition of AI: Artificial Intelligence is a branch of computer science concerned with the study and creation of computer systems that exhibit some form of intelligence such as systems that learn new concepts and tasks, systems that can understand a natural language or perceive and comprehend a visual scene, or systems that perform other types of feats that require human types of intelligence.

To understand AI, we should understand

- Intelligence
- Knowledge
- Reasoning
- Thought
- Cognition: gaining knowledge by thought or perception learning

The definitions of AI vary along two main dimensions: thought process and reasoning and behavior.

AI is not the study and creation of conventional computer systems. The study of the mind, the body, and the languages as customarily found in the fields of psychology, physiology, cognitive science, or linguistics.

In AI, the goal is to develop working computer systems that are truly capable of performing tasks that require high levels of intelligence.

2. Problems, Problem Spaces and Search

Problem:

A problem, which can be caused for different reasons, and, if solvable, can usually be solved in a number of different ways, is defined in a number of different ways.

To build a system or to solve a particular problem we need to do four things.

1. Define the problem precisely. This definition must include precise specification of what the initial situation will be as well as what final situations constitute acceptable solutions to the problem
2. Analyze the problem
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best solving technique and apply it to the particular problem.

Defining the Problem as a State Space Search

Problem solving = Searching for a goal state

It is a structured method for solving an unstructured problem. This approach consists of number of states. The starting of the problem is "Initial State" of the problem. The last point in the problem is called a "Goal State" or "Final State" of the problem.

State space is a set of legal positions, starting at the initial state, using the set of rules to move from one state to another and attempting to end up in a goal state.

Methodology of State Space Approach

1. To represent a problem in structured form using different states
2. Identify the initial state
3. Identify the goal state
4. Determine the operator for the changing state
5. Represent the knowledge present in the problem in a convenient form
6. Start from the initial state and search a path to goal state

To build a program that could "Play Chess"

- we have to first specify the starting position of the chess board

  Each position can be described by an 8-by-8 array.

  Initial position is the game opening position.

- rules that define the legal moves

  Legal moves can be described by a set of rules:

  - Left sides are matched against the current state.
  - Right sides describe the new resulting state.

- board positions that represent a win for one side or the other

  Goal position is any position in which the opponent does not have a legal move and his or her king is under attack.

- We must make explicit the preciously implicit goal of not only playing a legal game of chess but also winning the game, if possible.

Production System

The entire procedure for getting a solution for AI problem can be viewed as "Production System". It provides the desired goal. It is a basic building block which describes the AI problem and also describes the method of searching the goal. Its main components are:

- A Set of Rules, each consisting of a left side (a pattern) that determines the applicability of the rule and right side that describes the operation to be performed if the rule is applied.
- Knowledge Base – It contains whatever information is appropriate for a particular task. Some parts of the database may be permanent, while the parts of it may pertain only to the solution of the current problem.
- Control Strategy – It specifies the order in which the rules will be compared to the database and the way of resolving the conflicts that arise when several rules match at one.
  - The first requirement of a goal control strategy is that it is cause motion; a control strategy that does not cause motion will never lead to a solution.
  - The second requirement of a good control strategy is that it should be systematic.
- A rule applier: Production rule is like below `if(condition) then consequence or action`

Algorithm for Production System:

1. Represent the initial state of the problem
2. If the present state is the goal state then go to step 5 else go to step 3
3. Choose one of the rules that satisfy the present state, apply it and change the state to new state.
4. Go to Step 2
5. Print "Goal is reached " and indicate the search path from initial state to goal state 6. Stop

Classification of Production System:

Based on the direction they can be

1. Forward Production System
    - Moving from Initial State to Goal State
    - When there are number of goal states and only one initial state, it is advantage to use forward production system.
2. Backward Production System
    - Moving from Goal State to Initial State
    - If there is only one goal state and many initial states, it is advantage to use backward production system.

Production System Characteristics

Production system is a good way to describe the operations that can be performed in a search for solution of the problem.

Two questions we might reasonably ask at this point are:

- *Can production systems, like problems, be described by a set of characteristics that shed some light on how they can easily be implemented?*
- *If so, what relationships are there between problem types and the types of production systems best suited to solving the problems?*

The answer for the first question can be considered with the following *definitions of classes of production systems*:

A monotonic production system is a production system in which the applications of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected.

A non-monotonic production system is one which this is not true.

A partially commutative production system is a production system with the property that if the application of a particular sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state X into state Y.

A commutative production system is a production system that is both monotonic and partially commutative.

In a formal sense, there is no relationship between kinds of problems and kinds of production of systems, since all problems can be solved by all kinds of systems. But in practical sense, there definitely is such a relationship between kinds of problems and the kinds of systems that led themselves naturally to describing those problems.

The following figure shows the four categories of production systems produced by the two dichotomies, monotonic versus non-monotonic and partially commutative versus non-partially commutative along with some problems that can be naturally be solved by each type of system.

	Monotonic	Non-monotonic
Partially commutative	Theorem proving	Robot Navigation
Not Partially commutative	Chemical Synthesis	Bridge

The four categories of Production Systems

- Partially commutative, monotonic production systems are useful for solving ignorable problems that involves creating new things rather than changing old ones generally ignorable. Theorem proving is one example of such a creative process partially commutative, monotonic production system are important for a implementation stand point because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed.

- Non-monotonic, partially commutative production systems are useful for problems in which changes occur but can be reversed and in which order of operations is not critical. This is usually the case in physical manipulation problems such as “Robot navigation on a flat plane”. The 8-puzzle and blocks world problem can be considered partially commutative production systems are significant from an implementation point of view because they tend to read too much duplication of individual states during the search process.

- Production systems that are not partially commutative are useful for many problems in which changes occur. For example “Chemical Synthesis”

- Non-partially commutative production system less likely to produce the same node many times in the search process.

Problem Characteristics

In order to choose the most appropriate method (or a combination of methods) for a particular problem, it is necessary to analyze the problem along several key dimensions:

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the universe predictable?
- Is a good solution absolute or relative?
- Is the solution a state or a path?

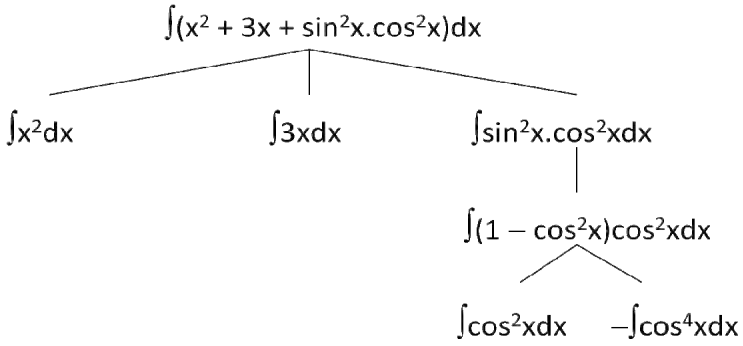
- What is the role of knowledge?
- Does the task require human-interaction?
- Problem Classification

Is the problem decomposable?

Decomposable problem can be solved easily. Suppose we want to solve the problem of computing the expression.

$$\int (x^2 + 3x + sin^2\ x.cos^2 x)\ dx$$

We can solve this problem by breaking it down into these smaller problems, each of which we can then solve by using a small collection of specific rules the following figure shows problem tree that as it can be exploited by a simple recursive integration program that works as follows.

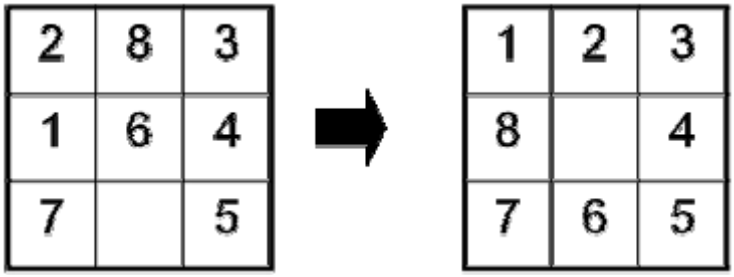


At each step it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly. If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. It can create those problems and calls itself recursively on using this technique of problem decomposition we can often solve very large problem easily.

Can solution steps be ignored or undone?

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. A lemma that has been proved can be ignored for next steps as eventually we realize the lemma is no help at all.

Now consider the 8-puzzle game. A sample game using the 8-puzzle is shown below:



In attempting to solve the 8 puzzle, we might make a stupid move for example; we slide the tile 5 into an empty space. We actually want to slide the tile 6 into empty space but we can back track and undo the first move, sliding tile 5 back to where it was then we can know tile 6 so mistake and still recovered from but not quit as easy as in the theorem moving problem. An additional step must be performed to undo each incorrect step.

Now consider the problem of playing chess. Suppose a chess playing problem makes a stupid move and realize a couple of moves later. But here solutions steps cannot be undone.

The above three problems illustrate difference between three important classes of problems:

1) Ignorable: in which solution steps can be ignored.
   Example: Theorem Proving
2) Recoverable: in which solution steps can be undone.
   Example: 8-Puzzle
3) Irrecoverable: in which solution steps cannot be undone.
   Example: Chess

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for problem solution.

Ignorable problems can be solved using a simple control structure that never backtracks. *Recoverable problems* can be solved by slightly complicated control strategy that does sometimes make mistakes using backtracking. *Irrecoverable problems* can be solved by recoverable style methods via planning that expands a great deal of effort making each decision since the decision is final.

Is the universe predictable?

There are certain outcomes every time we make a move we will know what exactly happen. This means it is possible to plan entire sequence of moves and be confident that we know what the resulting state will be. Example is 8-Puzzle.

In the uncertain problems, this planning process may not be possible. Example: Bridge Game – Playing Bridge. We cannot know exactly where all the cards are or what the other players will do on their turns.

We can do fairly well since we have available accurate estimates of a probabilities of each of the possible outcomes. A few examples of such problems are

- Controlling a robot arm: The outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the arm might stick.
- Helping a lawyer decide how to defend his client against a murder charge. Here we probably cannot even list all the possible outcomes, which leads outcome to be uncertain.

- *For certain-outcome problems, planning can used to generate a sequence of operators that is guaranteed to lead to a solution.*
- *For uncertain-outcome problems, a sequence of generated operators can only have a good probability of leading to a solution.*
- *Plan revision is made as the plan is carried out and the necessary feedback is provided.*

Is a Good Solution Absolute or Relative?

Consider the problem of answering questions based on a database of simple facts, such as the following:

1) Marcus was a man.
2) Marcus was a Pompeian.
3) Marcus was born in 40 A.D.
4) All men are mortal.
5) All Pompeian's died when the volcano erupted in 79 A.D.
6) No mortal lives longer than 150 years. 7) It is now 1991 A.D.

Suppose we ask a question "Is Marcus alive?" By representing each of these facts in a formal language such as predicate logic, and then using formal inference methods we can fairly easily derive an answer to the question.

		Justification
1.	Marcus was a man.	axiom 1
4.	All men are mortal.	axiom 4
8.	Marcus is mortal.	1, 4
3.	Marcus was born in 40 A.D.	axiom 3
7.	It is now 1991 A.D.	axiom 7
9.	Marcus' age is 1951 years.	3, 7
6.	No mortal lives longer than 150 years.	axiom 6
10.	Marcus is dead.	8, 6, 9
	OR	
7.	It is now 1991 A.D.	axiom 7
5.	All Pompeians died in 79 A.D.	axiom 5
11.	All Pompeians are dead now.	7, 5
2.	Marcus was a Pompeian.	axiom 2
12.	Marcus is dead.	11, 2

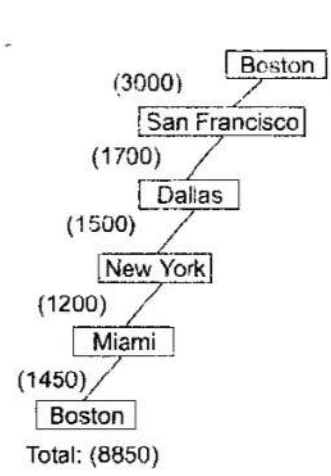
Two Ways of Deciding That Marcus Is Dead

Since we are interested in the answer to the question, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see if some other path might also lead to a solution. These types of problems are called as “Any path Problems”.

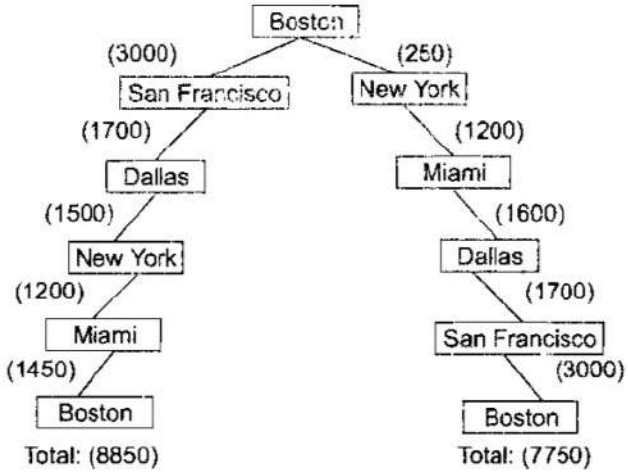
Now consider the Travelling Salesman Problem. Our goal is to find the shortest path route that visits each city exactly once

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

An Instance of the Traveling Salesman Problem



One Path among the Cities



Two Paths Among the Cities

Suppose we find a path it may not be a solution to the problem. We also try all other paths. The shortest path (best path) is called as a solution to the problem. These types of problems are known as "Best path" problems. But path problems are computationally harder than any path problems.

Is the solution a state or a path?

Consider the problem of finding a consistent interpretation for the sentence

The bank president ate a dish of pasta salad with the fork

There are several components of this sentence, each of which may have more than one interpretation. Some of the sources of ambiguity in this sentence are the following:

- The word "Bank" may refer either to a financed institution or to a side of river. But only one of these may have a President.
- The word "dish" is the object of the word "eat". It is possible that a dish was eaten.
- But it is more likely that the pasta salad in the dish was eaten.

Because of the interaction among the interpretations of the constituents of the sentence some search may be required to find a complete interpreter for the sentence. But to solve the problem of finding the interpretation we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary. But with the "water-jug" problem it is not sufficient to report the final state we have to show the "path" also.

So the solution of natural language understanding problem is a state of the world. And the solution of “Water jug” problem is a path to a state.

What is the role of knowledge?

Consider the problem of playing chess. The knowledge required for this problem is the rules for determining legal move and some simple control mechanism that implements an appropriate search procedure.

Now consider the problem of scanning daily newspapers to decide which are supporting ‘n’ party and which are supporting ‘y’ party. For this problems are required lot of knowledge.

The above two problems illustrate the difference between the problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

Does a task require interaction with the person?

Suppose that we are trying to prove some new very difficult theorem. We might demand a prove that follows traditional patterns so that mathematician each read the prove and check to make sure it is correct. Alternatively, finding a proof of the theorem might be sufficiently difficult that the program does not know where to start. At the moment people are still better at doing the highest level strategies required for a proof. So that the computer might like to be able to ask for advice.

For Example:

- Solitary problem, in which there is no intermediate communication and no demand for an explanation of the reasoning process.
- Conversational problem, in which intermediate communication is to provide either additional assistance to the computer or additional information to the user.

Problem Classification

When actual problems are examined from the point of view all of these questions it becomes apparent that there are several broad classes into which the problem fall. The classes can be each associated with a generic control strategy that is approached for solving the problem. There is a variety of problem-solving methods, but there is no one single way of solving all problems. Not all new problems should be considered as totally new. Solutions of similar problems can be exploited.

PROBLEMS

Water-Jug Problem

Problem is "You are given two jugs, a 4-litre one and a 3-litre one. One neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into 4-litre jug?"

Solution:

The state space for the problem can be described as a set of states, where each state represents the number of gallons in each state. The game start with the initial state described as a set of ordered pairs of integers:

- State: (x, y)
  - x = number of lts in 4 lts jug
  - y = number of lts in 3 lts jug

  x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3
- Start state: (0, 0) i.e., 4-litre and 3-litre jugs is empty initially.
- Goal state: (2, n) for any n that is 4-litre jug has 2 litres of water and 3-litre jug has any value from 0-3 since it is not specified.
- Attempting to end up in a goal state.

Production Rules: These rules are used as operators to solve the problem. They are represented as rules whose left sides are used to describe new state that result from approaching the rule.

1	(x, y) if $x < 4$	→	$(4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	→	$(x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	→	$(x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	→	$(x, y - d)$	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	→	$(0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	→	$(x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	→	$(4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	→	$(x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	→	$(x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	→	$(0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	→	$(2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	→	$(0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Fig. 2.3 *Production Rules for the Water Jug Problem*

The solution to the water-jug problem is:

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

One Solution to the Water Jug Problem

Chess Problem

Problem of playing chess can be defined as a problem of moving around in a state space where each state represents a legal position of the chess board.

The game start with an initial state described as an 8x8 of each position contains symbol standing for the appropriate place in the official chess opening position. A set of rules is used to move from one state to another and attempting to end up on one of a set of final states which is described as any board position in which the opponent does not have a legal move as his/her king is under attacks.

The state space representation is natural for chess. Since each state corresponds to a board position i.e. artificial well organized.

Initial State: Legal chess opening position

Goal State: Opponent does not have any legal move/king under attack

Production Rules:

These rules are used to move around the state space. They can be described easily as a set of rules consisting of two parts:

1. Left side serves as a pattern to be matching against the current board position.
2. Right side that serves decides the chess to be made to the board position to reflect the move.

To describe these rules it is convenient to introduce a notation for pattern and substitutions

E.g.:

1. White pawn at square (file1,rank2)

Move pawn from square (file i, rank2) AND square (file i, rank2)

AND

Square (file i,rank3) is empty → To square (file i,rank4)

AND

Square (file i,rank4) is empty

2. White knight at square (file i,rank1) move Square(1,1) to → Square(i-1,3)

AND

Empty Square(i-1,3)

3. White knight at square (1,1) move Square(1,1) to → Square(i-1,3)

AND

Empty Square(i-1,3)

8-Puzzle Problem

The Problem is 8-Puzzle is a square tray in which 8 square tiles are placed. The remaining 9th square is uncovered. Each tile has a number on it. A file that is adjacent to the blank space can be slide into that space. The goal is to transform the starting position into the goal position by sliding the tiles around.

Solution:

State Space: The state space for the problem can be written as a set of states where each state is position of the tiles on the tray.

Initial State: Square tray having 3x3 cells and 8 tiles number on it that are shuffled

2	8	3
1	6	4
7		5

Goal State

1	2	3
8		4
7	6	5

Production Rules: These rules are used to move from initial state to goal state. These are also defined as two parts left side pattern should match with current position and left side will be resulting position after applying the rule.

1. Tile in square (1,1)
AND *Move tile from square (1,1) to (2,1)*
Empty square (2,1)

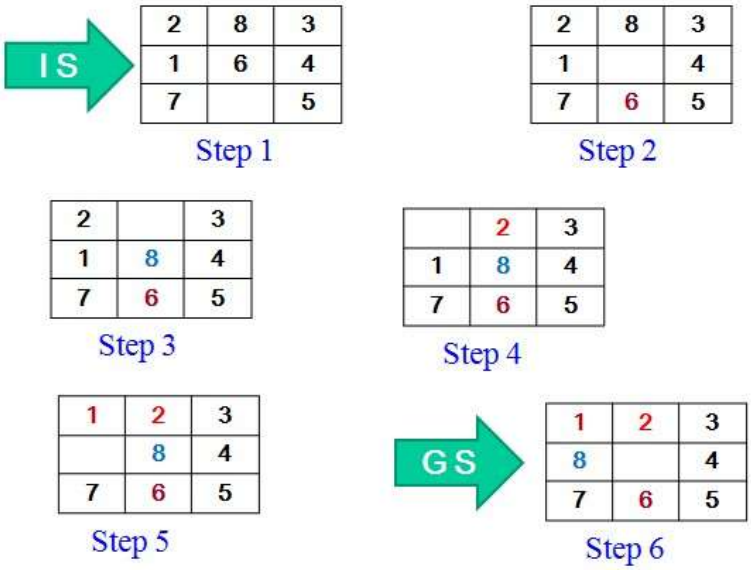
2. Tile in square (1,1)
AND *Move tile from square (1,1) to (1,2)* Empty square (1,2)

3. Tile in square (2,1)
AND *Move tile from square (2,1) to (1,1)*
Empty square (1,1)

1,1 2	1,2 3	1,3 2
2,1 3	2,2 4	2,3 3
3,1 2	3,2 3	3,3 2

No. of Production Rules: 2 + 3 + 2 + 3 + 4 + 3 + 2 + 3 + 2 = 24

Solution:



Travelling Salesman Problem

The Problem is the salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both states and finishes at any one of the cities.

Solution:

State Space: The state space for this problem represents states in which the cities traversed by salesman and state described as salesman starting at any city in the given list of cities. A set of rules is applied such that the salesman will not traverse a city traversed once. These rules are resulted to be states with the salesman will complex the round trip and return to his starting position.

Initial State

- Salesman starting at any arbitrary city in the given list of cities

Goal State

- Visiting all cities once and only and reaching his starting state

Production rules:

These rules are used as operators to move from one state to another. Since there is a path between any pair of cities in the city list, we write the production rules for this problem as

- `Visited(city[i]) AND Not Visited(city[j])`
  - `Traverse(city[i],city[j])`
- `Visited(city[i],city[j]) AND Not Visited(city[k])`
  - `Traverse(city[j],city[k])`
- `Visited(city[j],city[i]) AND Not Visited(city[k])`
  - `Traverse(city[i],city[k])`
- `Visited(city[i],city[j],city[k]) AND Not Visited(Nil)`
  - `Traverse(city[k],city[i])`

Towers of Hanoi Problem

Problem is the state space for the problem can be described as each state representing position of the disk on each pole the position can be treated as a stack the length of the stack will be equal to maximum number of disks each post can handle. The initial state of the problem will be any one of the posts will the certain the number of disks and the other two will be empty.

Initial State:

- Full(T1) | Empty(T2) | Empty(T3) Goal State:
- Empty(T1) | Full(T2) | Empty (T3)

Production Rules:

These are rules used to reach the Goal State. These rules use the following operations:

- POP(x) → Remove top element x from the stack and update top
- PUSH(x,y) → Push an element x into the stack and update top. [Push an element x on to the y]

Now to solve the problem the production rules can be described as follows:

1. Top(T1)<Top(T2) → PUSH(POP(T1),T2)
2. Top(T2)<Top(T1) → PUSH(POP(T2),T1)
3. Top(T1)<Top(T3) → PUSH(POP(T1),T3)
4. Top(T3)<Top(T1) → PUSH(POP(T3),T1)
5. Top(T2)<Top(T3) → PUSH(POP(T2),T3)
6. Top(T3)<Top(T2) → PUSH(POP(T3),T2)
7. Empty(T1) → PUSH(POP(T2),T1)
8. Empty(T1) → PUSH(POP(T3),T1)
9. Empty(T2) → PUSH(POP(T1),T3)
10. Empty(T3) → PUSH(POP(T1),T3)
11. Empty(T2) → PUSH(POP(T3),T2)
12. Empty(T3) → PUSH(POP(T2),T3)

Solution: Example: 3 Disks, 3 Towers

1) T1 → T2
2) T1 → T3
3) T2 → T3
4) T1 → T2
5) T3 → T1
6) T3 → T2
7) T1 → T2

Monkey and Bananas Problem

Problem: A hungry monkey finds himself in a room in which a branch of bananas is hanging from the ceiling. The monkey unfortunately cannot reach the bananas however in the room there are also a chair and a stick. The ceiling is just right high so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move round,

carry other things around reach for the bananas and wave the stick in the air. What is the best sequence of actions for the monkey to acquire lunch?

Solution: The state space for this problem is a set of states representing the position of the monkey, position of chair, position of the stick and two flags whether monkey on the chair & whether monkey holds the stick so there is a 5-tuple representation.

(M, C, S, F1, F2)

- M: position of the monkey
- C: position of the chair
- S: position of the stick
- F1: 0 or 1 depends on the monkey on the chair or not
- F2: 0 or 1 depends on the monkey holding the stick

Initial State (M, C, S, 0, 0)

- The objects are at different places and obviously monkey is not on the chair and not holding the stick

Goal State (G, G, G, 1, 1)

- G is the position under bananas and all objects are under it, monkey is on the chair and holding stick

Production Rules:

These are the rules which have a path for searching the goal state here we assume that when monkey hold a stick then it will swing it this assumption is necessary to simplify the representation.

Some of the production rules are:

1) (M,C,S,0,0) → (A,C,S,0,0) {An arbitrary position A}
2) (M,C,S,0,0) → (C,C,S,0,0) {monkey moves to chair position}
3) (M,C,S,0,0) → (S,S,S,0,0) {monkey brings chair to stick position}
4) (C,C,S,0,0) → (A,A,S,0,0) {push the chair to arbitrary position A}
5) (S,C,S,0,0) → (A,C,A,0,1) {Taking the stick to arbitrary position}
6) (S,C,S,0,0) → (C,C,S,0,0) {monkey moves from stick position to chair position}
7) (C,C,C,0,1) → (C,C,C,1,1)
    - {monkey and stick at the chair position, monkey on the chair and holding stick} 8) (S,C,S,0,1) → (C,C,C,0,1)

Solution:

1) (M,C,S,0,0)

2) (C,C,S,0,0)
3) (G,G,S,0,0)
4) (S,G,S,0,0)
5) (G,G,G,0,0)
6) (G,G,G,0,1)
7) (G,G,G,1,1)

Missionaries and Cannibals Problem

Problem is 3 missionaries and 3 cannibals find themselves one side of the river. They have agreed that they would like to get the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across without missionaries risking hang eager?

Solution:

The state space for the problem contains a set of states which represent the present number of cannibals and missionaries on the either side of the bank of the river. (C,M,C1,M1,B)

- C and M are number of cannibals and missionaries on the starting bank
- C1 and M1 are number of cannibals and missionaries on the destination bank
- B is the position of the boat wither left bank (L) or right bank (R)

Initial State → C=3,M=3,B=L so (3,3,0,0,L)

Goal State → C1=3, M1=3, B=R so (0,0,3,3,R)

Production System: These are the operations used to move from one state to other state. Since at any bank the number of cannibals must less than or equal to missionaries we can write two production rules for this problem as follows:

- (C,M,C1,M1,L / C=3, M=3) → (C-2,M,C1+2,M1,R)
- (C,M,C1,M1,L / C=3, M=3) → (C-1,M-1,C1+1,M1+1,R)
- (C,M,C1,M1,L / C=3, M=3) → (C-1,M,C1+1,M1,R)
- (C,M,C1,M1,R / C=1, M=3) → (C+1,M,C1-1,M1,L)
- (C,M,C1,M1,R / C=0, M=3,C1=3,M1=0) → (C+1,M,C1-1,M1,L)

The solution path is

LEFT BANK			RIGHT BANK	
C	M	BOAT POSITION	C1	M1

3	3		0	0
1	3	→	2	0
2	3	←	1	0
0	3	→	3	0
1	3	←	2	0
1	1	→	2	2
2	2	←	1	1
2	0	→	1	3
3	0	←	0	3
1	0	→	2	3
2	0	←	1	3
0	0	→	3	3

3. Heuristic Search Techniques

Control Strategy

The question arises

> "***How to decide which rule to apply next during the process of searching for a solution to a problem?***"

Requirements of a good search strategy:

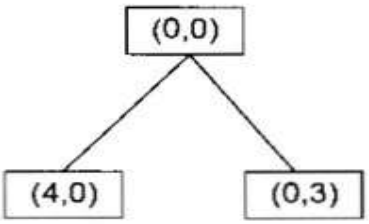
1. It causes motion. It must reduce the difference between current state and goal state. Otherwise, it will never lead to a solution.
2. It is systematic. Otherwise, it may use more steps than necessary.
3. It is efficient. Find a good, but not necessarily the best, answer.

Breadth First Search

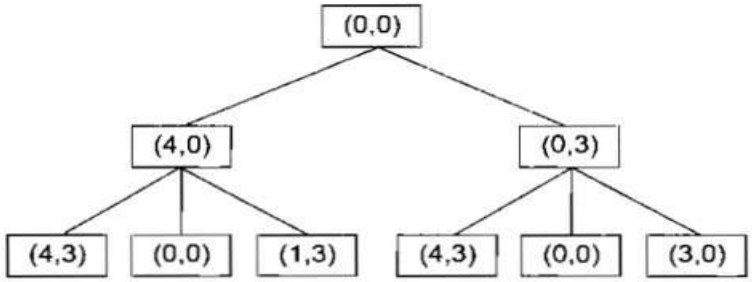
To solve the water jug problem systemically construct a tree with limited states as its root. Generate all the offspring and their successors from the root according to the rules until some rule produces a goal state. This process is called Breadth-First Search.

Algorithm:

1) Create a variable called NODE_LIST and set it to the initial state.
2) Until a goal state is found or NODE_LIST is empty do:
    a. Remove the first element from NODE_LIST and call it E. If NODE_LIST was empty quit.
    b. For each way that each rule can match the state described in E do:
        i. Apply the rule to generate a new state ii. If the new state is goal state, quit and return this state
        iii. Otherwise add the new state to the end of NODE_LIST



One Level of a Breadth-First Search Tree



Two Levels of a Breadth-First Search Tree

The data structure used in this algorithm is QUEUE.
Explanation of Algorithm:

- Initially put (0,0) state in the queue
- Apply the production rules and generate new state
- If the new states are not the goal state, (not generated before and not expanded) then only add these states to queue.

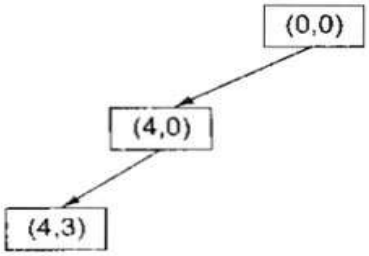
Depth First Search

There is another way of dealing the Water Jug Problem. One should construct a single branched tree utility yields a solution or until a decision terminate when the path is reaching a dead end to the previous state. If the branch is larger than the pre-specified unit then backtracking occurs to the previous state so as to create another path. This is called Chronological Backtracking because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. This procedure is called Depth-First Search.

Algorithm:

1) If the initial state is the goal state, quit return success.
2) Otherwise, do the following until success or failure is signaled
    a. Generate a successor E of the initial state, if there are no more successors, signal failure

b. Call Depth-First Search with E as the initial state
c. If success is returned, signal success. Otherwise continue in this loop.



A Depth-First Search Tree

The data structure used in this algorithm is STACK.

Explanation of Algorithm:

- Initially put the (0,0) state in the stack.
- Apply production rules and generate the ne
- If the new states are not a goal state, (not g
  then only add the state to top of the Stack.
- If already generated state is encountered then POP the top of stack elements and search in another direction.

Advantages of Breadth-First Search

- BFS will not get trapped exploring a blind alley.
  - In case of DFS, it may follow a single path for a very long time until it has no successor.
- If there is a solution for particular problem, the BFS is generated to find it. We can find minimal path if there are multiple solutions for the problem.

Advantages of Depth –First Search

- DFS requires less memory since only the nodes on the current path are stored.
- Sometimes we may find the solution without examining much.

Example: Travelling Salesman Problem

To solve the TSM problem we should construct a tree which is simple, motion causing and systematic. It would explore all possible paths in the tree and return the one with the shortest length. If there are N cities, then the number of different paths among them is 1.2...(N-1) or (N-1)!

The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to N!

Another strategy is, begin generating complete paths, keeping track of the shorter path so far and neglecting the paths where partial length is greater than the shortest found. This method is better than the first but it is inadequate.

To solve this efficiently we have a search called HEURISTIC SEARCH.

HEURISTIC SEARCH

Heuristic:

- It is a "rule of thumb" used to help guide search
- It is a technique that improves the efficiency of search process, possibly by sacrificing claims of completeness.
- It is involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods.

Heuristic Function:

- It is a function applied to a state in a search space to indicate a likelihood of success if that state is selected
- It is a function that maps from problem state descriptions to measures of desirability usually represented by numbers – Heuristic function is problem specific.

The purpose of heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available (best promising way).

We can find the TSM problem in less exponential items. On the average Heuristic improve the quality of the paths that are explored. Following procedure is to solve TRS problem

- Select a Arbitrary City as a starting city
- To select the next city, look at all cities not yet visited, and select one closest to the current city
- Repeat steps until all cities have been visited

Heuristic search methods which are the general purpose control strategies for controlling search is often known as "weak methods" because of their generality and because they do not apply a great deal of knowledge.

Weak Methods

a) Generate and Test
b) Hill Climbing
c) Best First Search
d) Problem Reduction
e) Constraint Satisfaction
f) Means-ends analysis

Generate and Test

The generate-and-test strategy is the simplest of all the approaches. It consists of the following steps:

Algorithm:

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.

3. If a solution has been found, quit. Otherwise return to step 1.

If there exists a solution for one problem then this strategy definitely finds the solution. Because the complete solution must be generated before they can be tested. So, we can say that Generate-and-test algorithm is a Depth-First Search procedure. It will take if the problem space is very large. In the strategy we can operate by generating solution randomly instead of systematically. Then we cannot give the surety that we will set the solution.

To implement this generate and test usually, we will use depth-first tree. If there are cycles then we use graphs rather than a tree. This is not an efficient (mechanism) technique when the problem is much harder. It is acceptable for simple problems. When it is combined with the other techniques it will restrict the space.

For example, one of the most successful AI program is DENDRAL, which informs the structure of organ i.e. components using mass spectrum and nuclear magnetic resonance data. It uses the strategy called *plan-generate-test*, in which a planning process that uses constraint satisfaction techniques, which creates lists of recommended structures. The generate-and-test procedure then uses those lists so that it can explain only a limited set of structures, which is proved highly effective.

Examples:

- Searching a ball in a bowl (Pick a green ball) - State
- Water Jug Problem – State and Path

Hill Climbing

A GENERATE and TEST procedure, if not only generates the alternative path but also the direction of the path in the alternatives which be near, than all the paths in Generate and Test procedures the heuristic function responds only yes or no but this heuristic function responds only yes will generate an estimate of how close a given state is to a goal state.

> Searching for a goal state = Climbing to the top of a hill Hill Climbing is Generate-and-test + direction to move.

Simplest Hill Climbing

- Apply only one particular rule at a time.

Algorithm:

1. Evaluate the initial state. If it is also goal state then return it, otherwise continue with the initial states as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state
    a) Select an operator that has not yet been applied to the current state and apply it to produce new state
    b) Evaluate the new state
        i. If it is a goal state then return it and quit
        ii. If it is not a goal state but it is better than the current state, then make it as current state iii. If it is not better than the current state, then continue in loop.

The key difference between this algorithm and generate and test algorithm is the use of an evaluation function as a way to inject task-specific knowledge into the control process.

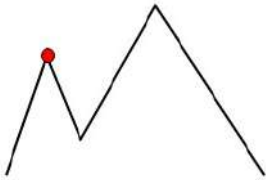
Steepest Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called *steepest-ascent hill climbing* or *gradient search*.

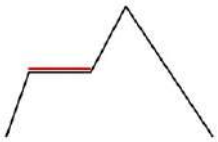
Algorithm:

1. Evaluate the initial state. If it is also a goal state then return it and quit. Otherwise continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
    a. Let SUCC be a state such that any possible successor of the current state will be better than SUCC.
    b. For each operator that applies to the current state do:
        i. Apply the operator and generate a new state.
        ii. Evaluate the new state. If it is a goal state, then return it and quit. If not compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
    c. IF the SUCC is better than current state, then set current state to SUCC.

Bothe basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting a state from which no better states can be generated. This will happen if the program has reached a local maximum, a plateau or a ridge.



A *local maximum* is a state that is better than all its neighbors but it not better than some other states farther away. At the local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills*.



A *plateau* is a flat area of the search space in which a whole set of has the same value. In this, it is not possible to determine the best di move by making local comparisons.

A *ridge* is a special kind of maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope.

There are some ways of dealing with these problems, although these methods are by no means guaranteed:

- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. This is a fairly good way to deal with local maxima.
- Make a big jump in some direction to try to get to a new section of the search space. This is a good way of dealing with plateaus.
- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a good strategy for dealing with ridges.

Simulated Annealing:

A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

In simulated annealing at the beginning of the process some hill moves may be made. The idea is to do enough exploration of the whole space early on. So that the final solution in relatively insensitive to the starting state. By doing so we can lower the chances of getting caught at local maximum, plateau or a ridge.

In this we attempt to minimize rather than maximize the value of the objective function. Thus this process is one of valley descending in which the object function is the energy level.

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy state.
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state: $e^{-E/kT}$.

The probability that a transaction to a higher energy state will occur and so given by a function:

$$P = e^{-\Delta E/kT}$$

- ❑ E is the +ve level in the energy level
- ❑ T is the temperature
- ❑ k is Boltzmann's constant

The rate at which the system is cooled is called annealing schedule in an analogous process. The units for both E and T are artificial. It makes sense to incorporate k into T.

Algorithm:

1. Evaluate the initial state. If it is also a goal state then return and quit. Otherwise continue with the initial state as a current state.
2. Initialize Best-So-Far to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state:
    a. Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    b. Evaluate the new state. Compute

$$\Delta E = (value\ of\ current) - (value\ of\ new\ state)$$

        (i) If the new state is goal state then return it and quit
        (ii) If it is not a goal state but is better than the current state then make it the current state. Also set BEST-SO-FAR to this new state.
        (iii) If it is not better than the current state, then make it the current state with probability ′ as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than ′ then the move is accepted. Otherwise do nothing.

    c. Revise T as necessary according to the annealing schedule.
5. Return BEST-SO-FAR as the answer.

Note:

For each step we check the probability of the successor with the current state. If it is greater than the current state the move is accepted. Otherwise move is rejected and search in other direction.

$$Current\ State\ p = 0.45\ and\ New\ State\ p' = 0.36 \rightarrow (p > p') - Move\ is\ Rejected$$
$$Current\ State\ p = 0.45\ and\ New\ State\ p' = 0.66 \rightarrow (p < p') - Move\ is\ Accepted$$

Best-First Search

Best-First Search (BFS) is a way of combining the advantages of both depth-first search and breadth first search into a single method, i.e., is to follow a single path at a time but switch paths whenever completing path looks more promising than the current one does.

The process is to select the most promising of the new nodes we have generated so far. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, then we can quit, else repeat the process until we search goal.

In BFS, one move is selected, but others are kept around so that they can be revisited later if the selected path becomes less promising. This is not the case steepest ascent climbing.

OR Graphs

A graph is called OR graph, since each of its branches represents alternative problems solving path.

To implement such a graph procedure, we will need to use lists of nodes:

1) *OPEN*: nodes that have been generated and have had the heuristic function applied to them which have not yet been examined. It is a priority queue in which the elements with highest priority are those with the most promising value of the heuristic function.
2) *CLOSED*: nodes that have already been examined whenever a new node is generated we need to check whether it has been generated before.
3) A heuristic function f which will estimate the merits of each node we generate.

Algorithm:

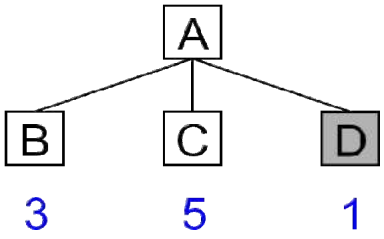
1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do:
    a. Pick the best node on OPEN
    b. Generate its successors
    c. For each successor do:
        i. If it is not been generated before, evaluate it, add it to OPEN and record its parent.
        ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case update the cost of getting to this node and to any successors that this node may already have.

Step 1:

A → NIL

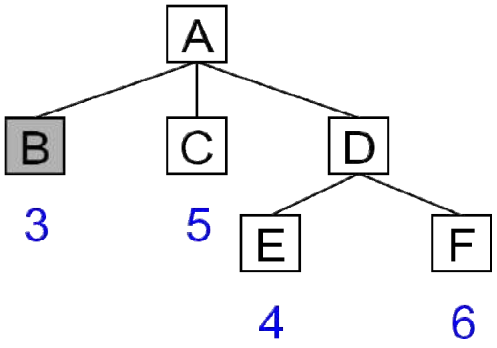
Step 2:

A
B
C
D



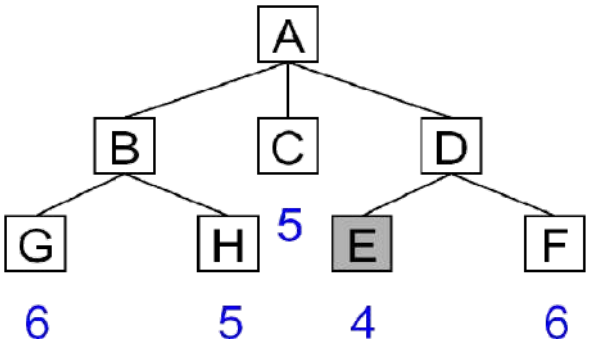
→ NIL
→ A
→ A
→ A

Step 3:



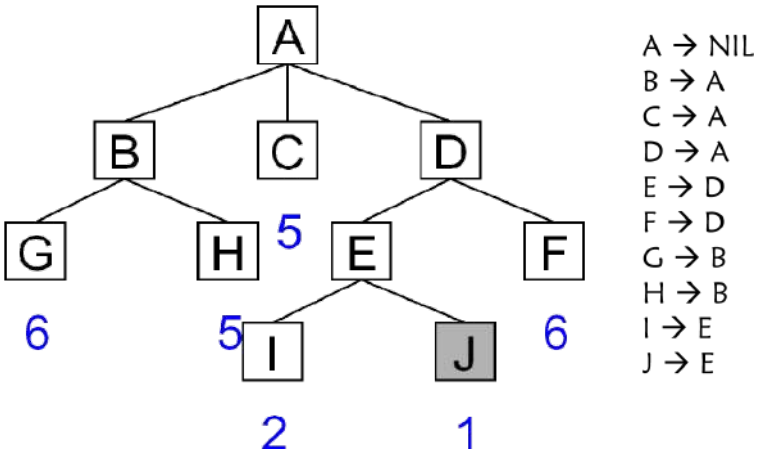
A → NIL
B → A
C → A
D → A
E → D
F → D

Step 4:



A → NIL
B → A
C → A
D → A
E → D
F → D
G → B
H → B

Step 5:



The Element with the low cost is the first element. The new states are added according to the cost value.

A* Algorithm:

A* algorithm is a best first graph search algorithm that finds a least cost path from a given initial node to one goal node. The simplification of Best First Search is called A* algorithm. This algorithm uses , functions as well as the lists OPEN and CLOSED.

For many applications, it is convenient to define function as the sum of two components that we call g and h'.

$$f' = g + h'$$

- g :
  - Measures of the cost of getting from the initial state to the current node.
  - It is not the estimate; it is known to be exact sum of the costs.
- **h' :**
  - is an estimate of the additional cost of getting from current node to goal state.

Algorithm:

1) Start with OPEN containing only the initial state (node) set that node g value 0 its h' value to whatever it is and its ' value '+ 0 or '. Set CLOSED to the empty list.

2) Until a goal node is found repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise pick the node on OPEN with lowest ' value. CALL it BESTNODE. Remove from OPEN. Place it on CLOSED. If BESTNODE is the goal node, exit and report a solution. Otherwise, generate the successors of BESTNODE. For each successor, do the following
   a) Set successors to point back to BESTNODE this backwards links will make possible to recover the path once a solution is found.

b) Compute

$$g(successor) = g(BESTNODE) + cost\ of\ getting\ from\ BESTNODE\ to\ successor$$

c) If successor is already exist in OPEN call that node as OLD and we must decide whether OLD' s parent link should reset to point to BESTNODE (graphs exist in this case)

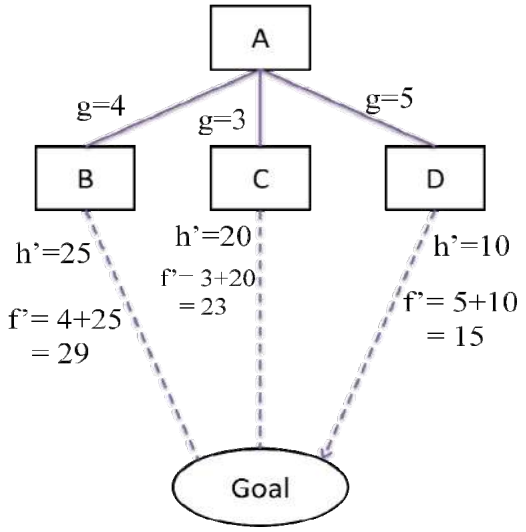
If OLD is cheaper then we need do nothing. If successor is cheaper then reset OLD's parent link to point to BESTNODE. Record the new cheaper path in ( ) and update ′( ).

d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call node on CLOSED OLD and add OLD to the list of BESTNODE successors. Calculate all the g, f' and h' values for successors of that node which is better then move that.

So to propagate the new cost downward, do a depth first traversal of the tree starting at OLD, changing each nodes value (and thus also its ′ value), terminating each branch when you reach either a node with no successor or a node which an equivalent or better path has already been found.

e) If successor was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE successors. Compute

$$f'(successor) = g(successor) + h'(successor)$$



A* algorithm is often used to search for the lowest cost path from the start to the goal location in a graph of visibility/quad tree. The algorithm solves problems like 8-puzzle problem and missionaries & Cannibals problem.

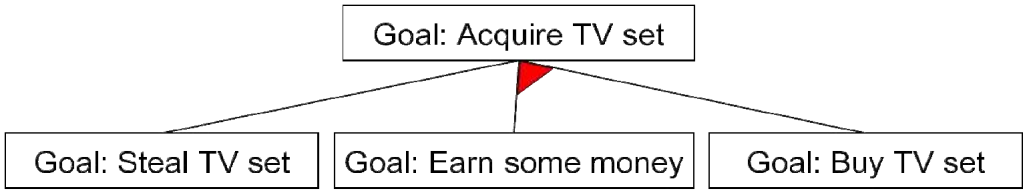
Problem Reduction:

- Planning how best to solve a problem that can be recursively decomposed into subproblems in multiple ways.

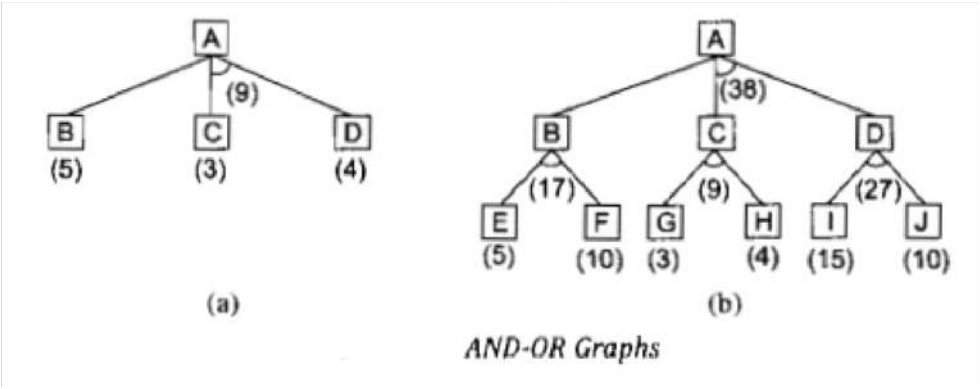
- There can be more than one decompositions of the same problem. We have to decide which is the best way to decompose the problem so that the total solution or cost of the solution is good.
- Examples:
    - Matrix Multiplication
    - Towers of Hanoi
    - Blocks World Problem
    - Theorem Proving
- Formulations: (AND/OR Graphs)
    - An OR node represents a choice between possible decompositions.
    - An AND node represents a given decomposition.

The AND-OR graph (or tree) is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition or reduction generate arcs that we call AND arcs.

One AND arc may point to any number of successors nodes all of which must be solved in order for the arc to point to a solution. Just as in OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved.



AND-OR Graphs



AND-OR Graphs

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately.

To see why our Best-First search is not adequate for searching AND-OR graphs, consider Fig (a).

- The top node A has been expanded, producing 2 arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of f' at that node.
- We assume for simplicity that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with multiple successors has a cost of 1 for each of its components.
- If we look just at the nodes and choose for expansion the one with the lowest f' value, we must select C. It would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C+D+2) compared to the cost of 6 that we get through B.
- The choice of which node to expand next must depend not only on the f' value of that node but also on whether that node is part of the current best path from the initial node.

The tree shown in Fig (b)

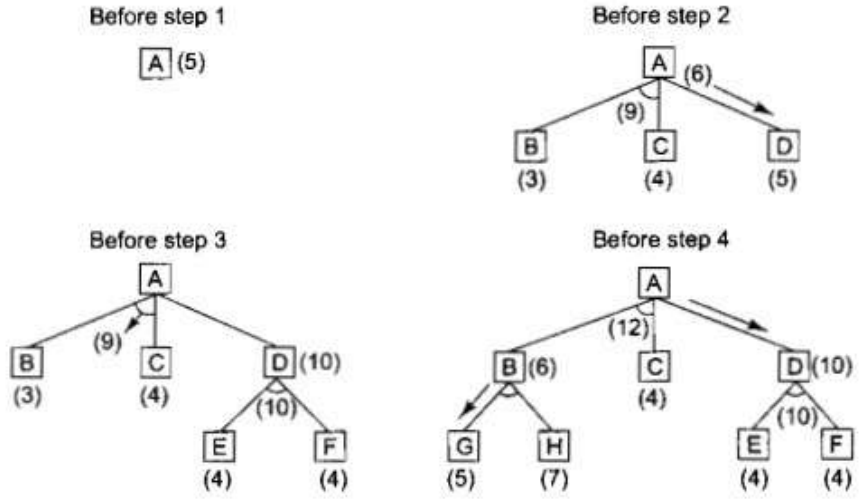
- The most promising single node is G with an f' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27.
- The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph we need to exploit a value that we call FUTILITY. If the estimated cost of a solution becomes greater than the value of FUTILITY, then we abandon the search. FUTILITY should be chosen to correspond to a threshold such any solution with a cost above it is too expensive to be practical even if it could ever be found.

Algorithm:

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:
    a. Traverse the graph, starting at the initial node following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded or labeled solved.
    b. Pick up one of those unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise add the successors to the graph and each of this compute f' (use only h' and ignore g). If f' of any node is "0", mark the node as SOLVED.
    c. Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor whose descendants are all solved, label the node itself as SOLVED. At each node that is visible while going up the graph, decide which of its successors arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. The

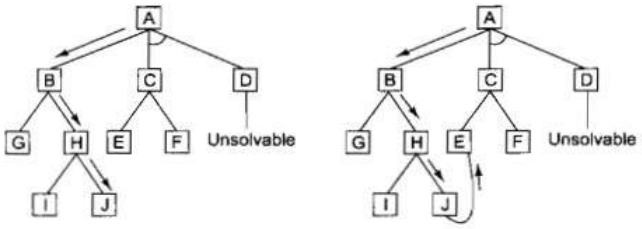
propagation of revised cost estimates backup the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.



The Operation of Problem Reduction .

- At Step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9.
- In Step 2, node D is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f' value of D to 10.
- We see that the AND arc B-C is better than the arc to D, so it is labeled as the current best path. At Step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually. SO explore B first.
- This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' to B to 6. This requires updating the cost of AND arc B-C to 12 (6+4+2). Now the arc to D is again the better path from A, so we record that as the current best path and either node E or F will be chosen for the expansion at Step 4.

This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.



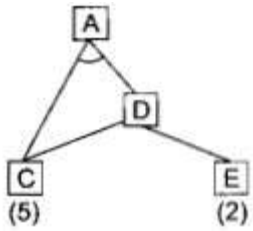
Limitations

1. A longer path may be better In Fig (a), the nodes were generated. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Fig (b). The new path to E is longer than the previous path to E going through C. Since the path through C will

only lead to a solution if there is also a solution to D, which there is not. The path through J is better.

While solving any problem please don't try to travel the nodes which are already labeled as solved because while implementing it may be struck in loop.

2. Interactive Sub-goals



Another limitation of the algorithm fails to take into account any interaction between sub-goals. Assume in figure that both node C and node E ultimately lead to a solution; our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must be solved. But the algorithm considers the solution of D as a completely separate process from the solution of C.

While moving to the goal state, keep track of all the sub-goals we try to move which one is giving an optimal cost.

AO* Algorithm:

AO* Algorithm is a generalized algorithm, which will always find minimum cost solution. It is used for solving cyclic AND-OR graphs The AO* will use a single structure GRAPH representing the part of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to immediate predecessors. The top down traversing of the best-known path which guarantees that only nodes that are on the best path will ever be considered for expansion. So h' will serve as the estimate of goodness of a node.

Algorithm (1):

1) Initialize: Set G* = {s}, f(s) = h(s).
   If , label s as SOLVED, where T is terminal node.

2) Terminate: If s is SOLVED then Terminate

3) Select: Select a non-terminal leaf node n from the marked sub tree

4) Expand: Make explicit the successors of n.
   For each new successor, m: Set f(m) = h(m)
   If m is Terminal, label m as SOLVED.

5) Cost Revision: Call cost-revise(n)

6) Loop: Goto Step 2.

Cost Revision

1. Create Z = { n }
2. Z = { } return
3. Otherwise: Select a node m from z such that m has no descendants in Z
4. If m is an AND node with successors r1, r2, …, rk

   Set (m) = ∑ [( ) + (,)]

   Mark the edge to each successor of m. If each successor is labeled SOLVED then label m as SOLVED.
5. If m is an OR node with successors r1, r2, …, rk

   Set (m) = min{ ( ) + (,) }

   Mark the edge to each successor of m. If each successor is labeled SOLVED then label m as SOLVED.
6. If the cost or label of m has changed, then insert those parents of m into Z for which m is marked successor.

Algorithm (2):

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.*) Compute $h'(INIT)$
2. Until *INIT* is labeled *SOLVED* or until *INIT's* h' value becomes greater than *FUTILITY,* repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE.*
   (b) Generate the successors of *NODE.* If there are none, then assign *FUTILITY* as the h' value of *NODE.* This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH.*
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its h' value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let *S* be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE.* Until *S* is empty, repeat the, following procedure:
      (i) If possible, select from *S* a node none of whose descendants in *GRAPH* occurs in *S.* If there is no such node, select any node from *S.* Call this node *CURRENT,* and remove it from *S.*
      (ii) Compute the cost of each of the arcs emerging from *CURRENT.* The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new h' value the minimum of the costs just computed for the arcs emerging from it.
      (iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.
      (iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED.*
      (v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to *S.*

Means-Ends Analysis:

One general-purpose technique used in AI is means-end analysis, a step-by-step, or incremental, reduction of the difference between the current state and the final goal. The program selects actions from a list of means–in the case of a simple robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT–until the goal is reached. This means we could solve major parts of a problem first and then return to smaller problems when assembling the final solution.

Usually, we search strategies that can reason either forward or backward. Often, however a mixture of the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems arise when combining them together. Such a technique is called "Means - Ends Analysis".

This process centers on the detection of difference between the current state and goal state. After the difference had been found, we should find an operator which reduces the difference. But this operator cannot be applicable to the current state. Then we have to set up a sub-problem of getting to the state in which it can be applied if the operator does not produce the goal state which we want. Then we should set up a sub-program of getting from state it does produce the goal. If the chosen inference is correct, the operator is effective, then the two sub-problems should be easier to solve than the original problem.

The means-ends analysis process can be applied recursively to them. In order to focus system attention on the big problems first, the difference can be assigned priority levels, in which high priority can be considered before lower priority.

Like the other problems, it also relies on a set of rules rather than can transform one state to another these rules are not represented with complete state description. The rules are represented as a left side that describes the conditions that must be met for the rule applicable and right side which describe those aspects of the problem state that will be changed by the application of the rule.

Consider the simple HOLD ROBOT DOMAIN. The available operators are as follows:

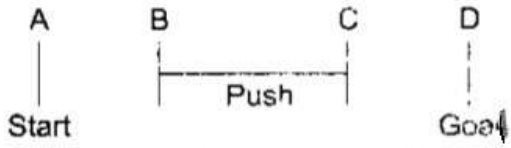
OPERATOR	PRECONDITIONS	RESULTS
PUSH(obj,loc)	At(robot,obj)^large(obj)^clear(obj)^ armempty	At(obj,loc)^at(robot,loc)
CARRY(obj,loc)	At(robot,obj)^small(obj)	At(obj,loc)^at(robot,loc)
WALK(loc)	NONE	At(robot,loc)
PICKUP(obj)	At(robot,obj)	Holding(obj)
PUTDOWN(obj)	Holding(obj)	7 Holding(obj)
PLACE(obj1,obj2)	At(robot,obj2)^Holding (obj1)	On(obj1,obj2)

Difference Table

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

A Difference Table

The difference table describes where each of the operators is appropriate table:

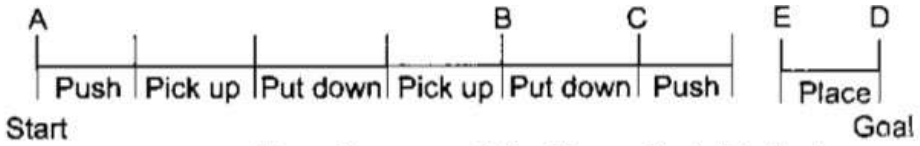


The Progress of the Means-Ends Analysis Method

Suppose that the robot were given the problems of moving desk with two things on it from one room to another room. The objects on top must also be moved the difference between start and goal is the location of the desk.

To reduce the difference either PUSH or CARRY can be chosen. If the CARRY is chosen first its precondition must be met. These results in two more differences that must be reduced; the location of the robot and the size of the desk. The location of the robot can be handled by applying WALK, but there are no operators that can change the size of the objects. So their path problem solve program will be shown above AND here also the thing does not get it quit to the goal state. So now the difference between A, B and between C, D must be reduced.

PUSH has 4-preconditions. Two of which produce difference between start and goal states since the desks is already large. One precondition creates no difference. The ROBOT can be brought to the location by using WALK, the surface can be cleared by two uses of pickup but after one pickup the second results in another difference – the arm must be empty. PUTDOWN can be used to reduce the difference.



More Progress of the Means-Ends Method

One PUSH is performed; the problem state is close to the goal state, but not quite. The objects must be placed back on the desk. PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot is holding the objects. Then we will find the progress as shown above. The final difference between C and E can be reduced by using WALK to get the ROBOT back to the objects followed by PICKUP and CARRY.

Algorithm:

1. Until the goal is reached or no more procedures are available:
   - Describe the current state, the goal state and the differences between the two.

- Use the difference the describe a procedure that will hopefully get nearer to goal.
- Use the procedure and update current state.

2. If goal is reached then success otherwise fail.

Algorithm:

Algorithm: Means-Ends Analysis (CURRENT, GOAL)

1. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:
   (a) Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.
   (b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.
   (c) If
   (FIRST-PART ← MEA(CURRENT, O-START))
   and
   (LAST-PART ← MEMO-RESULT, GOAL))
   are successful, *then* signal success and return the result of concatenating *FIRST-PART*, *O*, and *LAST-PART*.

Constraint Satisfaction

- Search procedure operates in a space of constraint sets. Initial state contains the original constraints given in the problem description.
- A goal state is any state that has been constrained enough – *Cryptarithmetic:* "enough" means that each letter has been assigned a unique numeric value.
- Constraint satisfaction is a 2-step process:
  - Constraints are discovered and propagated as far as possible.
  - If there is still not a solution, then search begins. A guess about is made and added as a new constraint.
- To apply the constraint satisfaction in a particular problem domain requires the use of 2 kinds of rules:
  - Rules that define valid constraint propagation
  - Rules that suggest guesses when necessary

Problem:

```
   SEND
+ MORE
.....
  MONEY
```

Initial State:

No two letters have the same value.
The sums of the digits must be as shown in the problem.

Fig. 3.13 *A Cryptarithmetic Problem*

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set *OPEN* to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until *OPEN* is empty:
   (a) Select an object *OB* from *OPEN*. Strengthen as much as possible the set of constraints that apply to *OB*.
   (b) If this set is different from the set that was assigned the last time *OB* was examined or if this is the first time *OB* has been examined, then add to *OPEN* all objects that share any constraints with *OB*.
   (c) Remove *OB* from *OPEN*.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
   (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   (b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

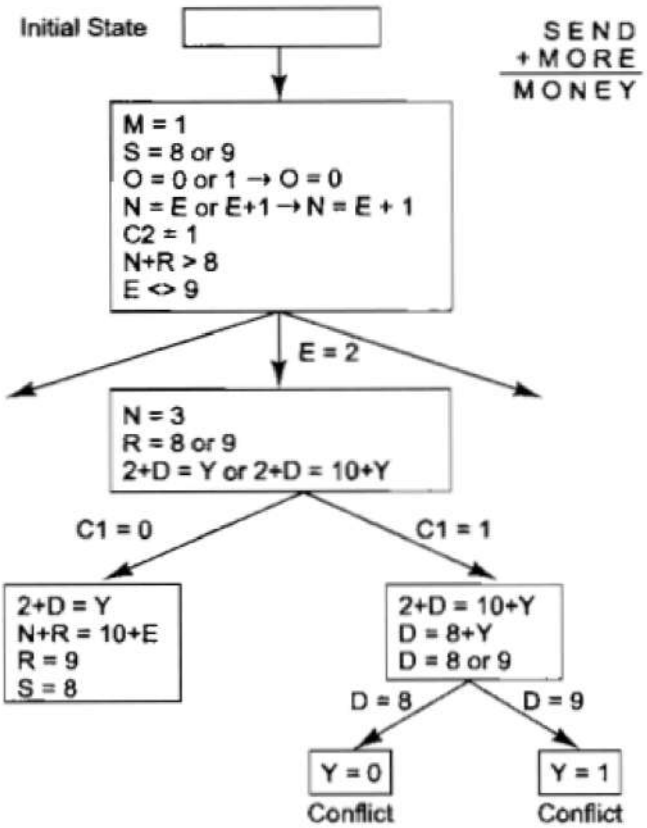


Fig. 3.14 *Solving a Cryptarithmetic Problem*

Goal State:

- We have to assign unique digit for the above specified alphabets.

→ 1. SEND
MORE
MONEY

0 1 2 3 4 5 6 7 8 9.

Sol:-

$C_3\ C_2\ C_1$
S E N D
M O R E
M O N E Y

0 1 2 3 4 5 6 7 8 9
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
O M Y E N O R S

1) Two single digit numbers sum is maximum 18 or 19.
(one is carry from previous addition)

So we conclude that $m=1$

2) $S+m+C_3 \geq 10$

$S+1+C_3 \geq 10$

The carry may be $C_3 = 0$ or 1.

$S+1+0 \geq 10 \Rightarrow S \geq 9 \Rightarrow S=9$.

$S+1+1 \geq 10 \Rightarrow S \geq 8 \Rightarrow S=8$ or 9

If $S=9 \Rightarrow 9+1+C_3 \Rightarrow C_3=0 \Rightarrow \frac{1}{m}\ \frac{0}{0}$ ✓

$C_3=1 \Rightarrow \frac{1}{m}\ \frac{1}{0}$

$S=8 \Rightarrow 8+1+C_3 \Rightarrow C_3=1 \Rightarrow \frac{1}{m}\ \frac{0}{0}$

3) let us consider $S=9$ ✓

4) $E+0+C_2 = N$

$\Downarrow$

$E+0+C_2 = N$

$E+1=N$

it implies C_2 must be 1

$C_2=1$

5) Guessing :-

$E \to 2$: $N + R + C_1 = C_2 E$

$E + 1 = N$: $3 + R + C_1 = 12$

$N = 3$: $R = 9, C_1 = 0$

conflict.

$D + E = C_1 Y$

$D + 2 = 13$

→ a) $E \to 5$

$E + 1 = N$

$\Downarrow$

$N = 6$

1 2 3 4 5 7 8

S = 9 ✓

n = 1 ✓

0 → 0 ✓

E → 5 ✓

N → 6 ✓

R → 8 ✓

6) $N + R + C_1 = C_2 E$

$6 + R + 1 = 15$

$\Downarrow$

$R = 8$ and $C_1 = 1$

7) $D + E = C_1 Y$

$D + E = 1Y$

$D + 5 = 1Y$

$D = 7, Y = 2$

∴ Ans:

$$\begin{array}{r} 9\ 5\ 6\ 7 \\ 1\ 0\ 8\ 5 \\ \hline (1)0\ 6\ 5\ 2 \end{array}$$

1) Define Intelligence, Artificial Intelligence.
2) List four things to build a system to solve a problem.
3) What is Production System?
4) Explain water Jug problem as a state space search.
5) Explain production system characteristics.
6) Explain A* algorithm with example.
7) What is Means-Ends Analysis? Explain with an example.
8) What do you mean by heuristic?
9) Write a heuristic function for travelling salesman problem.
10) What is heuristic search?
11) Explain problem characteristics.
12) Write AO* algorithm and explain the steps in it.
13) What is constraint satisfaction problem? Explain it.
14) Explain annealing schedule.
15) Explain Breadth-first search and depth-first search. List down the advantages and disadvantages of both?
16) What do you mean by an AI technique?
17) Discuss the tic-tac-toe problem in detail and explain how it can be solved using AI techniques.
18) What are the advantages of Heuristic Search?
19) Explain Turing Test as Criteria for success.
20) Explain Hill Climbing and give its disadvantages.
21) Define Control Strategy and requirements for good search strategy.
22) Define State Space Search. Write algorithm for state space.

5. Predicate Logic

Introduction

Predicate logic is used to represent Knowledge. Predicate logic will be met in Knowledge Representation Schemes and reasoning methods. There are other ways but this form is popular.

Propositional Logic

It is simple to deal with and decision procedure for it exists. We can represent real-world facts as logical propositions written as well-formed formulas.

To explore the use of predicate logic as a way of representing knowledge by looking at a specific example.

$$It\ is\ raining. \rightarrow RAINING$$
$$It\ is\ sunny. \rightarrow SUNNY$$
$$It\ is\ windy. \rightarrow WINDY$$
$$If\ it\ is\ raining\ then\ it\ is\ not\ sunny. : RAINING \rightarrow \neg SUNNY$$

$$Socrates\ is\ a\ man \rightarrow SOCRATESMAN$$
$$Plato\ is\ a\ man \rightarrow PLATOMAN$$

The above two statements becomes totally separate assertion, we would not be able to draw any conclusions about similarities between Socrates and Plato.

$$MAN(SOCRATES)$$
$$MAN(PLATO)$$

These representations reflect the structure of the knowledge itself. These use predicates applied to arguments.

$$All\ men\ are\ mortal \rightarrow MORTALMAN$$

It fails to capture the relationship between any individual being a man and that individual being a mortal.

We need variables and quantification unless we are willing to write separate statements.

Predicate:

A Predicate is a truth assignment given for a particular statement which is either true or false. To solve common sense problems by computer system, we use predicate logic.

Logic Symbols used in predicate logic

$\forall - For\ all$
$\exists - There\ exists$
$\rightarrow - Implies$
$\neg - Not$
$\vee - OR$
$\wedge - AND$

Predicate Logic

- Terms represent specific objects in the world and can be constants, variables or functions.
- Predicate Symbols refer to a particular relation among objects.
- Sentences represent facts, and are made of terms, quantifiers and predicate symbols.
- Functions allow us to refer to objects indirectly (via some relationship).
- Quantifiers and variables allow us to refer to a collection of objects without explicitly naming each object.
- Some Examples
    - Predicates: Brother, Sister, Mother , Father
    - Objects: Bill, Hillary, Chelsea, Roger
    - Facts expressed as atomic sentences a.k.a. literals:
        - Father(Bill,Chelsea)
        - Mother(Hillary,Chelsea)
        - Brother(Bill,Roger)
        - Father(Bill,Chelsea)

Variables and Universal Quantification

Universal Quantification allows us to make a statement about a collection of objects:

- $\forall x\ Cat(x) \Rightarrow Mammel(x)$: All cats are mammels
- $\forall x\ Father(Bill,x) \Rightarrow Mother(Hillary,x)$: All of Bill's kids are also Hillary's kids.

Variables and Existential Quantification

Existential Quantification allows us to state that an object does exist (without naming it):

- $\exists x\ Cat(x) \wedge Mean(x)$: There is a mean cat.
- $\exists x\ Father(Bill,x) \wedge Mother(Hillary,x)$: There is a kid whose father is Bill and whose mother is Hillary

Nested Quantification

- ∀x,y Parent(x,y) → Child(y,x)
- ∀x ∃y Loves(x,y)
- ∀x [Passtest(x) ∨ (∃x ShootDave(x))]

Functions

- Functions are terms - they refer to a specific object.
- We can use functions to symbolically refer to objects without naming them.
- Examples:

  fatherof(x) age(x) times(x,y) succ(x)

- Using functions
  - ∀ x Equal(x,x)
  - Equal(factorial(0),1)
  - ∀ x Equal(factorial(s(x)), times(s(x),factorial(x)))

If we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge.

Representing facts with Predicate Logic

1) Marcus was a man man(Markus)
2) Marcus was a Pompeian pompeian(Markus)
3) All Pompeians were Romans $\forall x : pompeian(x) \rightarrow roman(x)$
4) Caeser was a ruler. ruler(caeser)
5) All romans were either loyal to caeser or hated him.

   $\forall x : roman(x) \rightarrow loyalto(x, caeser) \vee hate(x, caeser)$
6) Everyone loyal to someone. $\forall x, \exists y : loyalto(x, y)$
7) People only try to assassinate rulers they are not loyal to.

$$\forall x, \forall y: Person(x) \wedge Ruler(y) \wedge try\_assassinate(x, y) \rightarrow \neg Loyal\_to(x, y)$$

8) Marcus try to assassinate Ceaser $try\_assacinate(Marcus, Ceaser)$

Q. Prove that Marcus is not loyal to Ceaser by backward substitution

4. $\neg Loyal\_to(Marcus, Ceaser)$

↑

5. $Person(Marcus) \wedge Ruler(Ceaser) \wedge Try\_assacinate(Marcus, Ceaser)$

6. ↑

7. $Person(Marcus) \wedge Ruler(Ceaser)$

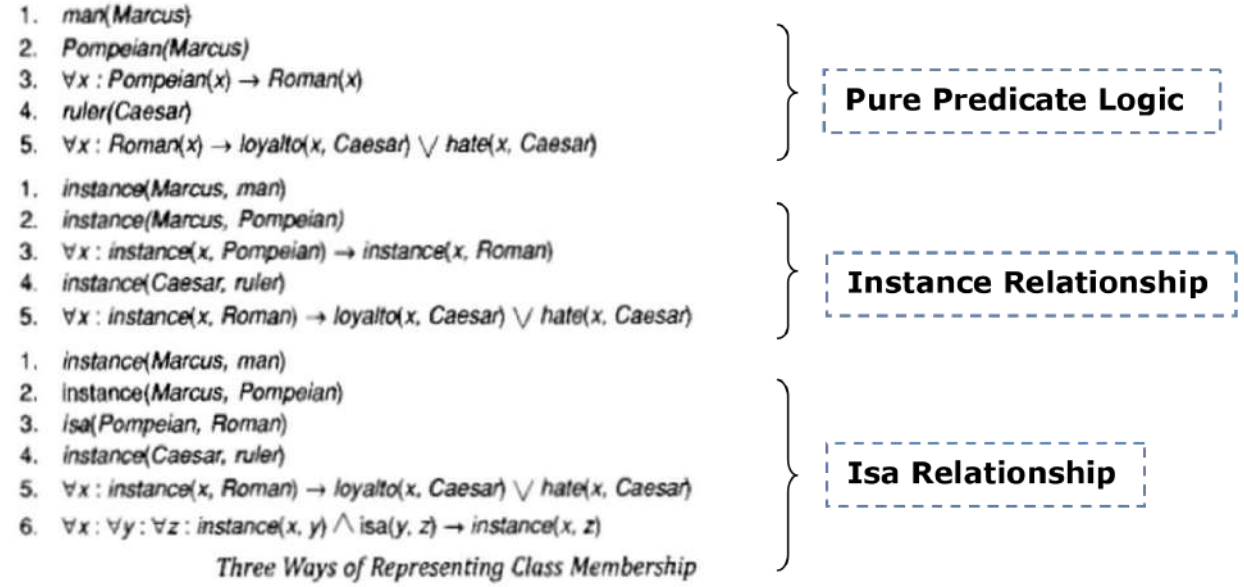
8. ↑

9. $Person(Marcus)$

Representing Instance and Isa Relationships

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

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



Three Ways of Representing Class Membership

In the figure above,

- The first five sentences of the represent the *pure predicate logic*. In these representations, class membership is represented with unary predicates (such as Roman), each of which corresponds to a class. Asserting that P(x) is true is equivalent to asserting that x is an instance of P.
- The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate instance is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit isa predicate.
- The third part contains representations that use both the instance and *isa* predicates explicitly. The use of the isa predicate simplifies the representation of sentence 3, but it requires that one additional axiom be provided. This additional axiom describes how an instance relation and an isa relation can be combined to derive a new instance relation.

Computable Functions and Predicates

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

gt(1,0) lt(0,1)

gt(2,1) lt(1,2)

gt(3,2) lt(2,3)

Clearly we do not want to have to write out explicit representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*.

1. Marcus was a Man
2. Marcus was a Pompeian
3. Marcus
4. All m
5. All P
6. No m
7. It is n
8. Alive
9. If son

$Man(Marcus)$

$Pompeian(Marcus)$

(x)

$79))$

$) \rightarrow died(x, t2)$

$e(x, t))$

$t, t2)$

Fig. 5.50 *One Way of Proving That Marcus Is Dead*

¬alive(Marcus, now)
↑ (9, substitution)
dead(Marcus, now)
↑ (7, substitution)
mortal(Marcus) ∧
born(Marcus, t_1) ∧
gt(now − t_1, 150)
↑ (4, substitution)
man(Marcus) ∧
born(Marcus, t_1) ∧
gt(now − t_1, 150)
↑ (1)
born(Marcus, t_1) ∧
gt(now − t_1, 150)
↑ (3)
gt(now − 40,150)
↑ (8)
gt(1991 − 40,150)
↑ (compute minus)
gt(1951,150)
↑ (compute gt)
nil

Another Way of Proving That Marcus is Dead

Resolution:

A procedure to prove a statement, Resolution attempts to show that Negation of Statement gives Contradiction with known statements. It simplifies proof procedure by first converting the statements into canonical form. Simple iterative process; at each step, 2 clauses called the parent clauses are compared, yielding a new clause that has been inferred from them.

Resolution refutation:

Resolution inference rule

$$\frac{(\alpha \vee \neg\beta) \wedge (\gamma \vee \beta) \text{ premise}}{(\alpha \vee \gamma) \text{ conclusion}}$$

- Convert all sentences to CNF (conjunctive normal form)
- Negate the desired conclusion (converted to CNF)

  Apply resolution rule until either
  - Derive false (a contradiction)
  - Can't apply any more

Resolution refutation is sound and complete

- If we derive a contradiction, then the conclusion follows from the axioms
- If we can't apply any more, then the conclusion cannot be proved from the axioms.

Sometimes from the collection of the statements we have, we want to know the answer of this question - "Is it possible to prove some other statements from what we actually know?" In order to prove this we need to make some inferences and those other statements can be shown true using Refutation proof method i.e. proof by contradiction using Resolution. So for the asked goal we will negate the goal and will add it to the given statements to prove the contradiction.

So resolution refutation for propositional logic is a complete proof procedure. So if the thing that you're trying to prove is, in fact, entailed by the things that you've assumed, then you can prove it using resolution refutation.

Clauses:

- Resolution can be applied to certain class of wff called clauses.
- A clause is defined as a wff consisting of disjunction of literals.

Conjunctive Normal Form or Clause Normal Form:

Clause form is an approach to Boolean logic that expresses formulas as conjunctions of clauses with an AND or OR. Each clause connected by a conjunction or AND must be wither a literal or contain a disjunction or OR operator. In clause form, a statement is a series of ORs connected by ANDs.

A statement is in conjunctive normal form if it is a conjunction (sequence of ANDs) consisting of one or more conjuncts, each of which is a disjunction (OR) of one or more literals (i.e., statement letters and negations of statement letters).

All of the following formulas in the variables A, B, C, D, and E are in conjunctive normal form:

- $\neg A \wedge (B \vee C)$
- $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$
- $A \vee B$
- $A \wedge B$

Conversion to Clause Form:

$$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$$

➔ Clause Form:

$$\neg Roman(x) \wedge \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$$

Algorithm:

1. Eliminate implies relation ($\rightarrow$) Using (Ex: $\rightarrow$ =>)

$$\forall x : \neg[Roman(x) \wedge know(x, Marcus)] \vee [hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x,y))]$$

2. Reduce the scope of each to a single term

$$\neg(\neg P) = P$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$
$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable.

$\forall x: P(x) \vee \forall x: Q(x)$ can be converted to

$\forall x: P(x) \vee \forall y: Q(y)$

4. Move all quantifiers to the left of the formulas without changing their relative order.

$$\exists x: \forall x, \forall y : P(x) \vee Q(x)$$

$$\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x\ Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x,y))]$$

5. Eliminate existential quantifiers. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. $\exists$y: President(y) => President(S1)

$\exists y$: President(y) => President(S1)

$\forall x, \exists y$: Fatherof(y,x) => $\forall x$: Fatherof(S2(s),x)

President(func()) → func is called a skolem function.

In general the function must have the same number of arguments as the number of universal quantifiers in the current scope.

Skolemize to remove existential quantifiers. This step replaces existentially quantified variables by Skolem functions. For example, convert (x)P(x) to P(c) where c is a brand

new constant symbol that is not used in any other sentence (c is called a Skolem constant). More generally, if the existential quantifier is within the scope of a universal quantified variable, then introduce a Skolem function that depends on the universally quantified variable. For example, " x y P(x,y) is converted to " x P(x, f(x)). f is called a Skolem function, and must be a brand new function name that does not occur in any other part of the logic sentence.

6. Drop the prefix. At this point, all remaining variables are universally quantified.

$$P(x) \vee Q(x)$$

$$[\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$$

7. Convert the matrix into a conjunction of disjunctions.

$$(a \vee b) \vee c = a \vee (b \vee c) \quad \text{Associative Law}$$
$$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c) \quad \text{Distributive Laws}$$
$$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$$
$$a \vee b = b \vee a \quad \text{Commutative Law}$$

$$\neg Roman(x) \vee \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, y)$$

8. Create a separate clause corresponding to each conjunct in order for a well formed formula to be true, all the clauses that are generated from it must be true.
9. Standardize apart the variables in set of clauses generated in step 8. Rename the variables. So that no two clauses make reference to same variable.

Convert the statements to clause form

1. man(marcus)
2. pompeian(marcus)
3. $\forall$ pompeian(x) → roman(x)
4. ruler(caeser)
5. $\forall$x: roman(x) → loyalto(x,caeser) V hate(x,caeser)

$\forall x, \exists y$: loyalto(x,y)

$\forall x, \forall y$: person(x) $\wedge$ ruler(y) $\wedge$ tryassacinate(x,y) → $\neg$ loyalto(x,y)

tryassacinate(marcus, caeser)

The resultant clause form is

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

Basis of Resolution:

Resolution process is applied to pair of parent clauses to produce a derived clause. Resolution procedure operates by taking 2 clauses that each contain the same literal. The literal must occur in the positive form in one clause and negative form in the other. The resolvent is obtained by combining all of the literals of two parent clauses except ones that cancel. If the clause that is produced in an empty clause, then a contradiction has been found.
Eg: winter and winter will produce the empty clause.

If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

Resolution in Propositional Logic:

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resulting clause, called the *resolvent*, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

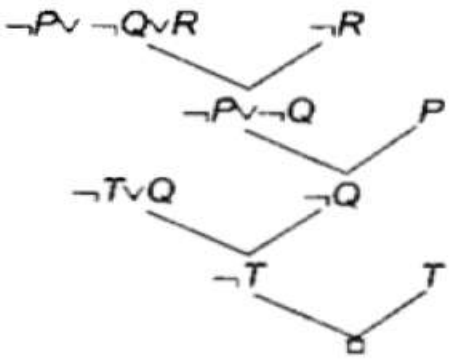
Example: Consider the following axioms

P $(P \wedge Q) \rightarrow R$ $(S \vee T) \rightarrow Q$ T

Convert them into clause form and prove that R is true

1. P
2. $(P \wedge Q) \rightarrow R \quad => \quad \neg (P \wedge Q) \vee R \quad \rightarrow \quad \neg P \vee \neg Q \vee R$
3. $(S \vee T) \rightarrow R$

   $\neg (S \vee T) \vee Q \quad -> \quad (\neg S \wedge \neg T) \vee Q \quad -> \quad (\neg S \vee Q) \wedge (\neg T \vee Q)$
4. T



$\neg$ ***R is contradiction***. Hence, R is true.

Unification Algorithm

- In propositional logic it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ~L . In predicate logic, this matching process is more complicated, since bindings of variables must be considered.
- In order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical.
- There is a recursive procedure that does this matching. It is called Unification algorithm.
- The process of finding a substitution for predicate parameters is called unification.
- We need to know:
  - that 2 literals can be matched.
  - the substitution is that makes the literals identical.
- There is a simple algorithm called the unification algorithm that does this.

The Unification Algorithm

1. Initial predicate symbols must match.
2. For each pair of predicate arguments:
   - Different constants cannot match.
   - A variable may be replaced by a constant.
   - A variable may be replaced by another variable.
   - A variable may be replaced by a function as long as the function does not contain an instance of the variable.

- When attempting to match 2 literals, all substitutions must be made to the entire literal.

- There may be many substitutions that unify 2 literals; the most general unifier is always desired.

Unification Example:

$P(x)\ and\ P(y)$: $substitution\ =\ (x/y) \rightarrow substitution\ x\ for\ y$

$P(x,x)\ and\ P(y,z)$: $P(z/y)(y/x) \rightarrow y\ for\ x, then\ z\ for\ y$

$P(f(x))\ and\ P(x)$: $can't\ do\ it!$

$P(x) \vee Q(Jane)\ and\ P(Bill) \vee Q(y)$: $(Bill/x, Jane/y)$

$Father(Bill, Chelsea)\ \neg Father(Bill, x) \vee Mother(Hillary, x)$
$Man(Marcus) \neg Man(x) \vee Mortal(x)$

$Loves(father(a), a) \neg Loves(x, y) \vee Loves(y, x)$

The object of the Unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many

$hate(x, y)$
$hate(Marcus, z)$
could be unified with any of the following substitutions:
$(Marcus/x, z/y)$
$(Marcus/x, y/z)$
$(Marcus/x, Caeser/y, Caeser/z)$
$(Marcus/x, Polonius/y, Polunius/z)$

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list.

The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are:

- Different constants, functions or predicates cannot match, whereas identical ones can.
- A variable can match another variable, any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).

- The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent. (a substitution y for x written as y/x)

Algorithm: Unify(Ll, L2)

1. If $L1$ or $L2$ are both variables or constants, then:
   (a) If $L1$ and $L2$ are identical, then return NIL.
   (b) Else if $L1$ is a variable, then if $L1$ occurs in $L2$ then return {FAIL}, else return ($L2/L1$).
   (c) Else if $L2$ is a variable then if $L2$ occurs in $L1$ then return {FAIL}, else return ($L1/L2$).
   (d) Else return {FAIL}.
2. If the initial predicate symbols in $L1$ and $L2$ are not identical, then return {FAIL).
3. If LI and $L2$ have a different number of arguments, then return {FAIL}.
4. Set *SUBST* to NIL. (At the end of this procedure, *SUBST* will contain all the substitutions used to unify $L1$ and $L2$.)
5. For $i \leftarrow 1$ to number of arguments in $L1$:
   (a) Call Unify with the /th argument of $L1$ and the ith argument of $L2$, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
      (i) Apply S to the remainder of both $L1$ and $L2$.
      (ii) *SUBST* := APPEND(S, *SUBST*).
6. Return *SUBST*.

Example:

Suppose we want to unify *p(X,Y,Y)* with *p(a,Z,b)*.

Initially *E* is *{p(X,Y,Y)=p(a,Z,b)}*.

The first time through the while loop, *E* becomes *{X=a,Y=Z,Y=b}*.

Suppose *X=a* is selected next.

Then *S* becomes*{X/a}* and *E* becomes *{Y=Z,Y=b}*.

Suppose *Y=Z* is selected.

Then *Y* is replaced by *Z* in *S* and *E*.

S becomes*{X/a,Y/Z}* and *E* becomes *{Z=b}*.

Finally *Z=b* is selected, *Z* is replaced by *b*, *S* becomes *{X/a,Y/b,Z/b}*, and *E* becomes empty.

The substitution *{X/a,Y/b,Z/b}* is returned as an MGU.

Unification:

$$\forall x: knows(John, x) \rightarrow hates(John, x)$$
$$knows(John, Jane)$$
$$\forall y: knows(y, Leonid)$$
$$\forall y: knows(y, mother(y))$$
$$\forall x: knows(x, Elizabeth)$$

$$UNIFY(knows(John, x), knows(John, Jane)) = \{Jane/x\}$$
$$UNIFY(knows(John, x), knows(y, Leonid)) = \{Leonid/x, John/y\}$$
$$UNIFY(knows(John, x), knows(y, mother(y))) = \{John/y, mother(John)/x\}$$
$$UNIFY(knows(John, x), knows(x, Elizabeth)) = FAIL$$

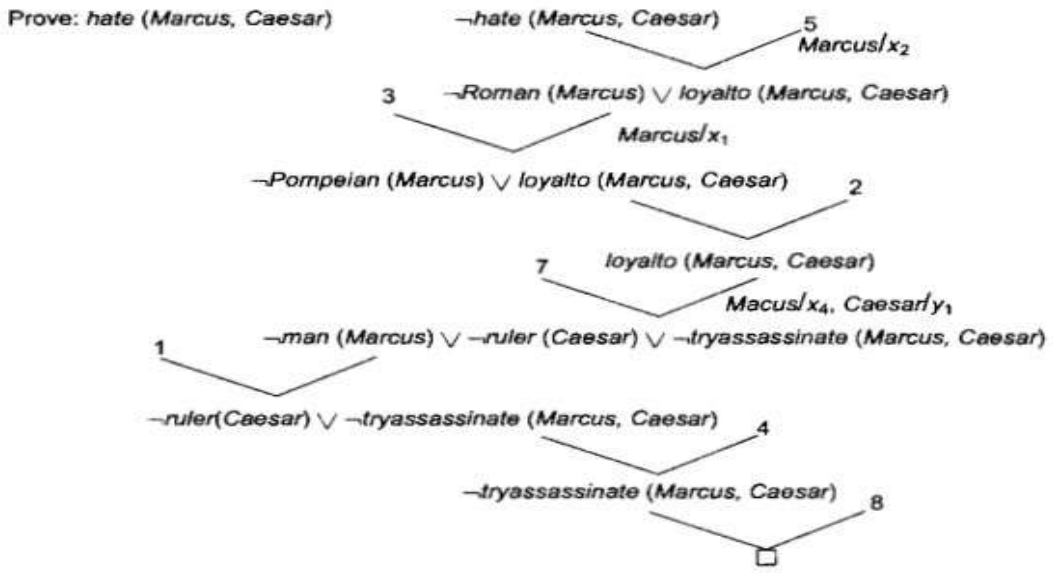
Resolution in Predicate Logic

- Two literals are contradictory if one can be unified with the negation of the other.
  - For example man(x) and man (Himalayas) are contradictory since man(x) and man(Himalayas) can be unified.
- In predicate logic unification algorithm is used to locate pairs of literals that cancel out.
- It is important that if two instances of the same variable occur, then they must be given identical substitutions

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg T2$ such that one of the parent clauses contains $T2$ and the other contains $T1$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $T2$ should appear in the resolvent. We call $T1$ and $T2$ *Complementary literals*. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Prove that Marcus hates ceaser using resolution.



Example:

John likes all kinds of food.
Apples are food.
Chicken is food.
Anything anyone eats and it is not killed is food.
Bill eats peanuts and is still alive.
Swe eats everything bill eats

(a) Convert all the above statements into predicate logic
(b) Show that John likes peanuts using back chaining
(c) Convert the statements into clause form
(d) Using Resolution show that "John likes peanuts"

Answer:

(a) Predicate Logic:

1. $\forall x: food(x) \rightarrow like(John)$
2. $Food(Apples)$
3. $Food(Chicken)$
4. $\forall x, \forall y: Eat(x,y) \wedge \neg Killed(x) \rightarrow Food(y)$
5. $Eats(Bill, Peanuts) \wedge Alive(Bill)$
6. $\forall x: Eats(Bill, x) \rightarrow Eats(Swe, x)$

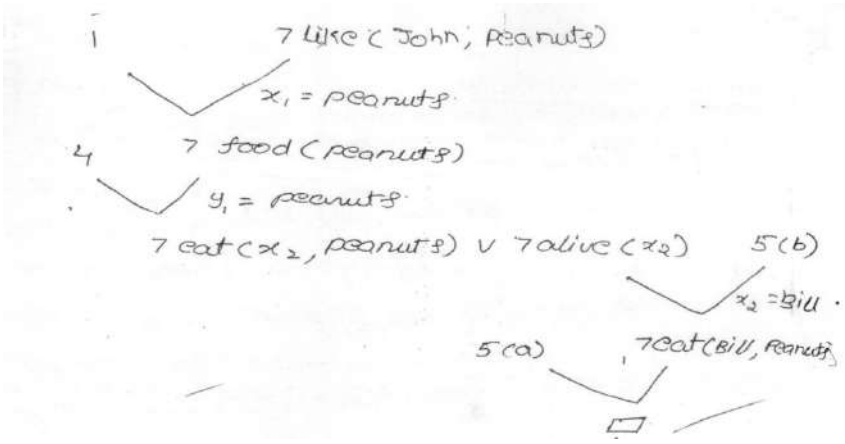
(b) Backward Chaining Proof:

$$Like\ (John, Peanuts)$$
$$\uparrow$$
$$Food(Peanuts)$$
$$\uparrow$$
$$Eat(Bill, Peanuts) \wedge Alive(Bill)$$
$$\uparrow$$
$$Nil$$

(c) Clause Form:

1. $\neg\ Food(x) \vee Like(John, x)$
2. $Food(Apples)$
3. $Food(Chicken)$
4. $\neg\ (Eat(x, y) \wedge \neg\ Killed(x)) \vee Food(y)\ \Rightarrow (\neg\ Eat(x, y) \vee Killed(x))\ \vee\ Food(y)$
5. $Eats(Bill, Peanuts)$
6. $Alive(Bill)$
7. $\neg\ (Eats(Bill, x))\ \vee Eats(Swe, x)$

(d) Resolution Proof:



Answering Questions

We can also use the proof procedure to answer questions such as “who tried to assassinate Caesar” by proving:

- Tryassassinate(y,Caesar).

- Once the proof is complete we need to find out what was substitution was made for y.

We show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or "Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the terms given in the question and then responding with another piece of the same statement that fills the slot demanded by the question.

From Clause Form to Horn Clauses

The operation is to convert Clause form to Horn Clauses. This operation is not always possible. Horn clauses are clauses in normal form that have one or zero positive literals. The conversion from a clause in normal form with one or zero positive literals to a Horn clause is done by using the implication property.

$$\neg P \vee Q \ \ Rewrites\ to\ P \rightarrow Q$$

Example:

Predicate

$\forall x\,(\neg literate(x) \supset (\neg writes(x) \wedge \neg\exists y(reads(x,y) \wedge book(y))))$

Simplify

$\forall x\,(literate(x) \vee (\neg writes(x) \wedge \neg\exists y(reads(x,y) \wedge book(y))))$

Move negations in

$\forall x\,(literate(x) \vee (\neg writes(x) \wedge \forall y(\neg(reads(x,y) \wedge book(y)))))$

$\forall x\,(literate(x) \vee (\neg writes(x) \wedge \forall y(\neg reads(x,y) \vee \neg book(y))))$

No Skolemize (there are no existential quantifiers)

Remove universal quantifier

$\forall x\,\forall y\,(literate(x) \vee (\neg writes(x) \wedge (\neg reads(x,y) \vee \neg book(y))))$

$literate(x) \vee (\neg writes(x) \wedge (\neg reads(x,y) \vee \neg book(y)))$

Distribute disjunctions

$(literate(x) \vee \neg writes(x)) \wedge (literate(x) \vee \neg reads(x,y) \vee \neg book(y))$

$(\neg writes(x) \vee literate(x)) \wedge (\neg reads(x,y) \vee \neg book(y) \vee literate(x))$

Convert to Clause Normal Form

$\neg writes(x) \vee literate(x)$

$\neg reads(x,y) \vee \neg book(y) \vee literate(x)$

Convert to Horn Clauses

$writes(x) \supset literate(x)$

$reads(x,y) \wedge book(y) \supset literate(x)$

Example 2
Predicate $\forall x$ (literate(x) $\supset$ reads(x) $\vee$ write(x)) Simplify $\forall x$ ($\neg$literate(x) $\vee$ reads(x) $\vee$ write(x)) The negations are already in $\forall x$ ($\neg$literate(x) $\vee$ reads(x) $\vee$ write(x)) No Skolemize (there are no existential quantifiers) Remove universal quantifier $\neg$literate(x) $\vee$ reads(x) $\vee$ write(x) No disjunctions It is already a Clause Normal Form $\neg$literate(x) $\vee$ reads(x) $\vee$ write(x) It is not possible to convert to Horn Clauses because there are two positive literals (reads(x) and write(x)).

Introduction:

Knowledge plays an important role in AI systems. The kinds of knowledge might need to be represented in AI systems:

- Objects: Facts about objects in our world domain. e.g. Guitars have strings, trumpets are brass instruments.
- Events: Actions that occur in our world. e.g. Steve Vai played the guitar in Frank Zappa's Band.
- Performance: A behavior like playing the guitar involves knowledge about how to do things.
- Meta-knowledge: Knowledge about what we know. e.g. Bobrow's Robot who plan's a trip. It knows that it can read street signs along the way to find out where it is.

Representations & Mappings:

In order to solve complex problems in AI we need:

- A large amount of knowledge
- Some mechanisms for manipulating that knowledge to create solutions to new problem.

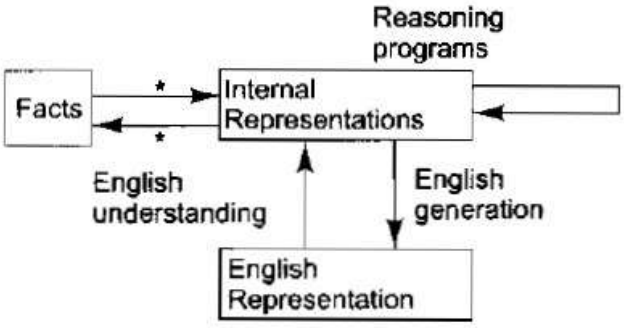
A variety of ways of representing knowledge have been exploited in AI problems. In this regard we deal with two different kinds of entities:

- Facts: truths about the real world and these are the things we want to represent.
- Representation of the facts in some chosen formalism. These are the things which we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

- Knowledge Level, at which facts are described.
- Symbol Level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

Mappings between Facts and Representations:



The model in the above figure focuses on facts, representations and on the 2-way mappings that must exist between them. These links are called *Representation Mappings*.

- Forward Representation mappings maps from Facts to Representations.
- Backward Representation mappings maps from Representations to Facts.

English or natural language is an obvious way of representing and handling facts. Regardless of representation for facts, we use in program, we need to be concerned with English

Representation of those facts in order to facilitate getting information into or out of the system.

Mapping functions from English Sentences to Representations: Mathematical logic as representational formalism.

Example:

"Spot is a dog"

The fact represented by that English sentence can also be represented in logic as:

dog(Spot)

Suppose that we also have a logical representation of the fact that

$$\text{"All dogs have tails"} \rightarrow \forall x: dog(x) \rightarrow hastail(x)$$

Then, using the deductive mechanisms of logic, we may generate the new representation object: astail(Spot)

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

Fact-Representation mapping may not be one-to-one but rather are many-to-many which are a characteristic of English Representation. Good Representation can make a reasoning program simple.

Example:

"All dogs have tails"

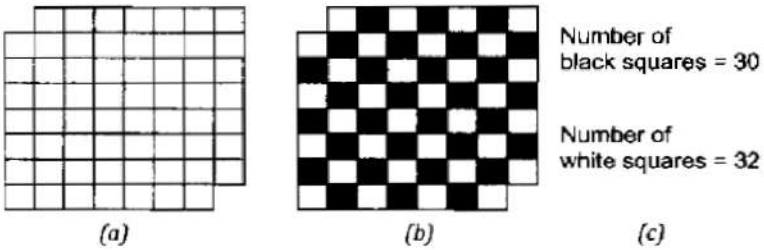
"Every dog has a tail"

From the two statements we can conclude that "Each dog has a tail." From the statement 1, we conclude that "Each dog has more than one tail."

When we try to convert English sentence into some other represent such as logical propositions, we first decode what facts the sentences represent and then convert those facts into the new representations. When an AI program manipulates the internal representation of facts these new representations should also be interpretable as new representations of facts.

Mutilated Checkerboard Problem:

Problem: In a normal chess board the opposite corner squares have been eliminated. The given task is to cover all the squares on the remaining board by dominoes so that each domino covers two squares. No overlapping of dominoes is allowed, can it be done? Consider three data structures

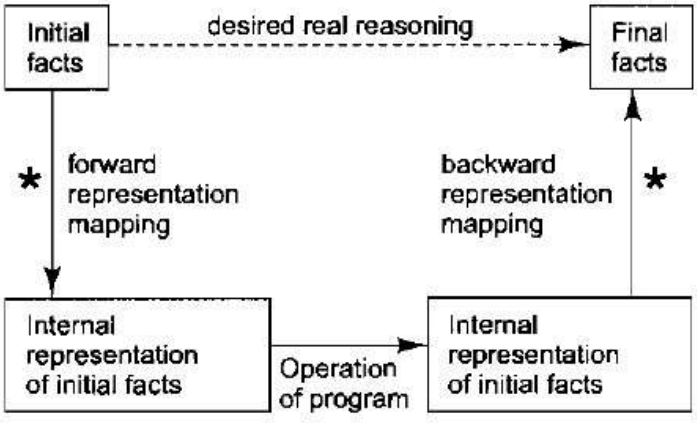


Three Representations of a Mutilated Checker board

The first representation does not directly suggest the answer to the problem. The second may suggest. The third representation does, when combined with the single additional facts that each domino must cover exactly one white square and one black square.

The puzzle is impossible to complete. A domino placed on the chessboard will always cover one white square and one black square. Therefore a collection of dominoes placed on the board will cover an equal numbers of squares of each color. If the two white corners are removed from the board then 30 white squares and 32 black squares remain to be covered by dominoes, so this is impossible. If the two black corners are removed instead, then 32 white squares and 30 black squares remain, so it is again impossible.

The solution is number of squares must be equal for positive solution.



Representation of Facts

In the above figure, the dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated.

If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

Using Knowledge

Let us consider to what applications and how knowledge may be used.

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

If a system on only knows:

- Miles Davis is a Jazz Musician.
- All Jazz Musicians can play their instruments well.

If things like *Is Miles Davis a Jazz Musician?* or *Can Jazz Musicians play their instruments well?* are asked then the answer is readily obtained from the data structures and procedures.

However a question like "*Can Miles Davis play his instrument well?*" requires reasoning. The above are all related. For example, it is fairly obvious that learning and reasoning involve retrieval etc.

Approaches to Knowledge Representation

A good Knowledge representation enables fast and accurate access to Knowledge and understanding of content. *The goal of Knowledge Representation (KR) is to facilitate conclusions from knowledge.*

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

- Representational Adequacy: the ability to represent all kinds of knowledge that are needed in that domain;
- Inferential Adequacy: the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;
- Inferential Efficiency: the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.
- Acquisitional Efficiency: the ability to acquire new information easily. The simplest case involves direct insertion, by a person of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

No single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist.

Knowledge Representation Schemes

There are four types of Knowledge Representation:

- Relational Knowledge:
  - provides a framework to compare two objects based on equivalent attributes
  - any instance in which two different objects are compared is a relational type of knowledge
- Inheritable Knowledge:
  - is obtained from associated objects
  - it prescribes a structure in which new objects are created which may inherit all or a subset of attributes from existing objects.
- Inferential Knowledge
  - is inferred from objects through relations among objects
  - Example: a word alone is simple syntax, but with the help of other words in phrase the reader may infer more from a word; this inference within linguistic is called semantics.

- Declarative Knowledge
  - a statement in which knowledge is specified, but the use to which that knowledge is to be put is not given.
  - Example: laws, people's name; there are facts which can stand alone, not dependent on other knowledge
- Procedural Knowledge
  - a representation in which the control information, to use the knowledge is embedded in the knowledge itself.
  - Example: computer programs, directions and recipes; these indicate specific use or implementation

Simple relational knowledge

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

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

Table - Simple Relational Knowledge

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

Given the facts it is not possible to answer simple question such as "Who is the heaviest player?" but if a procedure for finding heaviest player is provided, then these facts will enable that procedure to compute an answer. We can ask things like who "bats - left" and "throws - right".

Inheritable Knowledge

Here the knowledge elements inherit attributes from their parents. The knowledge is embodied in the design hierarchies found in the functional, physical and process domains. Within the hierarchy, elements inherit attributes from their parents, but in many cases not all attributes of the parent elements be prescribed to the child elements.

The inheritance is a powerful form of inference, but not adequate. The basic KR needs to be augmented with inference mechanism.

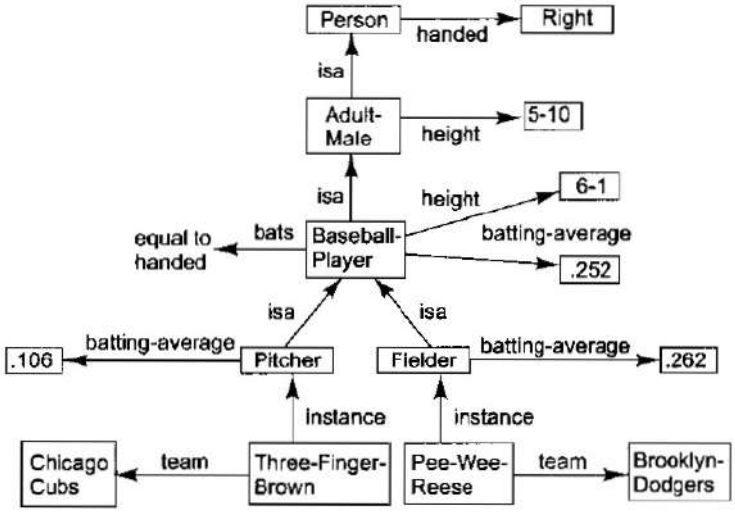
The KR in hierarchical structure, shown below, is called “semantic network” or a collection of “frames” or “slot-and-filler structure”. The structure shows property inheritance and way for insertion of additional knowledge.

Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes. The classes are organized in a generalized hierarchy.

Baseball Knowledge

- isa: show class inclusion

- i



Inheritable Knowledge

- The directed arrows represent attributes (isa, instance, team) originates at object being described and terminates at object or its value.
- The box nodes represent objects and values of the attributes.

Viewing a node as a frame

Example: Baseball-player

Isa: Adult-Male
Bats: EQUAL handed
Height: 6-1
Batting-average: 0.252

Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A.* If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attribute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A.* If there is, report it.

This algorithm is simple. It describes the basic mechanism of inheritance. It does not say what to do if there is more than one value of the instance or "isa" attribute.

This can be applied to the example of knowledge base, to derive answers to the following queries:

- team (Pee-Wee-Reese) = Brooklyn-Dodger
- batting-average (Three-Finger-Brown) = 0.106
- height (Pee-Wee-Reese) = 6.1
- bats (Three-Finger-Brown) = right

Inferential Knowledge:

This knowledge generates new information from the given information. This new information does not require further data gathering from source, but does require analysis of the given information to generate new knowledge. In this, we represent knowledge as formal logic.

Example:

- given a set of relations and values, one may infer other values or relations
- a predicate logic (a mathematical deduction) is used to infer from a set of attributes.
- inference through predicate logic uses a set of logical operations to relate individual data.
- the symbols used for the logic operations are:

" → " (implication), " ¬ " (not), " V " (or), " Λ " (and),
" ∀ " (for all), " ∃ " (there exists).

Examples of predicate logic statements :

1. *"Wonder"* is a name of a dog : **dog (wonder)**
2. All dogs belong to the class of animals : **∀ x : dog (x) → animal(x)**
3. All animals either live on land or in water : **∀ x : animal(x) → live (x, land) V live (x, water)**

From these three statements we can infer that :

***" Wonder* lives either on land or on water."**

Note : If more information is made available about these objects and their relations, then more knowledge can be inferred.

Procedural Knowledge

Procedural knowledge can be represented in programs in many ways. The most common way is simply as for doing something. The machine uses the knowledge when it executes the code to perform a task. Procedural Knowledge is the knowledge encoded in some procedures.

Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of inferential adequacy (because it is very difficult to write a program that can reason about another program's behavior) and acquisitional efficiency (because the process of updating and debugging large pieces of code becomes unwieldy).

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules.

If: ninth inning, and
score is close, and
less than 2 outs, and
first base is vacant, and
batter is better hitter than next batter,
Then: walk the batter.

Procedural Knowledge as Rules

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods. But making a clean distinction between declarative and procedural knowledge is difficult. The important difference is in how the knowledge is used by the procedures that manipulate it.

Heuristic or Domain Specific knowledge can be represented using Procedural Knowledge.

Issues in Knowledge Representation

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

- Important Attributes :
  Any attribute of objects so basic that they occur in almost every problem domain ?
- Relationship among attributes:
  Any important relationship that exists among object attributes ?
- Choosing Granularity :
  At what level of detail should the knowledge be represented ?
- Set of objects :
  How sets of objects be represented ?
- Finding Right structure :
  Given a large amount of knowledge stored, how can relevant parts be accessed ?

Important Attributes : *(Ref. Example - Fig. Inheritable KR)*

There are attributes that are of general significance.

There are two attributes **"instance"** and **"isa"**, that are of general importance. These attributes are important because they support *property inheritance*.

The attributes are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. The predicates are used in Logic Based Systems.

Relationship among Attributes

- The attributes to describe objects are themselves entities that we represent.
- The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:

  Inverses, existence in an isa hierachy, techniques for reasoning about values and single valued attributes.

Inverses :

This is about *consistency check*, while a value is added to one attribute. The entities are related to each other in many different ways. The figure shows attributes *(isa, instance, and team)*, each with a directed arrow, originating at the object being described and terminating either at the object or its value.

There are two ways of realizing this:

- first, represent two relationships in a *single representation*; e.g., a logical representation, ***team(Pee-Wee-Reese, Brooklyn–Dodgers)***, that can be interpreted as a statement about Pee-Wee-Reese or Brooklyn–Dodger.
- second, use attributes that focus on a *single entity but use them in pairs*, one the inverse of the other; for e.g., one, ***team = Brooklyn–Dodgers*** , and the other, ***team = Pee-Wee-Reese, ...***

The second way can be realized using semantic net and frame based systems. This Inverses is used in Knowledge Acquisition Tools.

Existence in an "isa" hierarchy :

This is about *generalization-specialization*, like, classes of objects and specialized subsets of those classes. There are attributes and specialization of attributes.

Example: the attribute *"height"* is a specialization of general attribute *"physical-size"* which is, in turn, a specialization of *"physical-attribute"*. These generalization-specialization relationships for attributes are important because they support inheritance.

This also provides information about constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for reasoning about values :

This is about *reasoning values of attributes* not given explicitly.

Several kinds of information are used in reasoning, like,

height : must be in a unit of length,

age : of person can not be greater than the age of person's parents.

The values are often specified when a knowledge base is created.

Several kinds of information can play a role in this reasoning, including:

Information about the type of the value.

- Constraints on the value often stated in terms of related entities.
- Rules for computing the value when it is needed. (Example: of such a rule in for bats attribute). These rules are called backward rules. Such rules have also been called ifneeded rules.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called forward rules, or sometimes if-added rules.

Single valued attributes :

This is about a *specific attribute* that is guaranteed to take a unique value.

Example : A baseball player can at time have only a single height and be a member of only one team. KR systems take different approaches to provide support for single valued attributes.

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

Choosing Granularity

What level should the knowledge be represented and
what are the primitives ?

- Should there be a small number or should there be a large number of low-level primitives or High-level facts.
- High-level facts may not be adequate for inference while Low-level primitives may require a lot of storage.

Example of Granularity :

- Suppose we are interested in following facts
  John spotted Sue.
- This could be represented as
  Spotted (agent(John), object (Sue))
- Such a representation would make it easy to answer questions such are
  Who spotted Sue ?
- Suppose we want to know
  Did John see Sue ?
- Given only one fact, we cannot discover that answer.
- We can add other facts, such as
  Spotted (x , y) → saw (x , y)
- We can now infer the answer to the question.

Choosing the Granularity of Representation Primitives are fundamental concepts such as holding, seeing, playing and as English is a very rich language with over half a million words it is clear we will find difficulty in deciding upon which words to choose as our primitives in a series of situations. Separate levels of understanding require different levels of primitives and these need many rules to link together similar primitives.

Set of Objects

Certain properties of objects that are true as member of a set but not as individual;

Example : Consider the assertion made in the sentences

"there are more *sheep* than *people* in Australia", and

"*English* speakers can be found all over the world."

To describe these facts, the only way is to attach assertion to the sets representing people, sheep, and English.

The reason to represent sets of objects is :

If a property is true for all or most elements of a set,

then it is more efficient to associate it once with the set

rather than to associate it explicitly with every elements of the set .

This is done in different ways :

- in logical representation through the use of *universal quantifier*, and
- in hierarchical structure where node represent sets, the *inheritance propagate* set level assertion down to individual.

Example: assert **large (elephant)**;

Remember to make clear distinction between,

- whether we are asserting some property of the set itself, means, **the set of elephants is large**, or
- asserting some property that holds for individual elements of the set , means, **any thing that is an elephant is large**.

There are three ways in which sets may be represented :

(a) Name, as in the example – Ref Fig. Inheritable KR, the node - Baseball-Player and the predicates as Ball and Batter in logical representation.

(b) Extensional definition is to list the numbers, and

(c) In tensional definition is to provide a rule, that returns true or false depending on whether the object is in the set or not.

$\{x: sun - planet(x) \wedge human - inhabited(x)\}$ – *Intensional Definition*

Extensional Definition – *Set of our sun planets on which people live is Earth*

Finding Right Structure

Access to right structure for describing a particular situation.

It requires, selecting an initial structure and then revising the choice.

While doing so, it is necessary to solve following problems :

- how to perform an initial selection of the most appropriate structure.
- how to fill in appropriate details from the current situations.
- how to find a better structure if the one chosen initially turns out not to be appropriate.
- what to do if none of the available structures is appropriate.
- when to create and remember a new structure.

There is no good, general purpose method for solving all these problems.

Some knowledge representation techniques solve some of them.

6. Representing Knowledge using Rules

Procedural versus Declaration Knowledge

Declarative Knowledge	Procedural Knowledge
Factual information stored in memory and known to be static in nature.	the knowledge of how to perform, or how to operate
knowledge of facts or concepts	a skill or action that you are capable of performing
knowledge about that something true or false	Knowledge about how to do something to reach a particular objective or goal
knowledge is specified but how to use to which that knowledge is to be put is not given	control information i.e., necessary to use the knowledge is considered to be embedded in the knowledge itself
E.g.: concepts, facts, propositions, assertions, semantic nets …	E.g.: procedures, rules, strategies, agendas, models
It is explicit knowledge (describing)	It is tacit knowledge (doing)

The declarative representation is one in which the knowledge is specified but how to use to which that knowledge is to be put is not given.

- Declarative knowledge answers the question 'What do you know?'
- It is your understanding of things, ideas, or concepts.
- In other words, declarative knowledge can be thought of as the who, what, when, and where of information.
- Declarative knowledge is normally discussed using nouns, like the names of people, places, or things or dates that events occurred.

The procedural representation is one in which the control information i.e., necessary to use the knowledge is considered to be embedded in the knowledge itself.

- Procedural knowledge answers the question 'What can you do?'
- While declarative knowledge is demonstrated using nouns,
- Procedural knowledge relies on action words, or verbs.
- It is a person's ability to carry out actions to complete a task.

The real difference between declarative and procedural views of knowledge lies in which the control information presides.

Example:

1. $man(marcus)$
2. $man(ceaser)$
3. $\forall x: man(x) \rightarrow person(x)$
4. $person(cleopatra)$

The statements *1, 2 and 3 are procedural knowledge* and *4 is a declarative knowledge*.

Forward & Backward Reasoning

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. There are actually two directions in which such a search could proceed:

- Forward Reasoning,
  - from the start states
  - LHS rule must match with initial state
  - Eg: A → B, B→C => A→C
- Backward Reasoning,
  - from the goal states
  - RHS rules must match with goal state
  - Eg: 8-Puzzle Problem

In both the cases, the control strategy is it must cause motion and systematic. The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes.

Consider the problem of solving a particular instance of the 8-puzzle problem. The rules to be used for solving the puzzle can be written as:

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile *n* →
Square 2 empty and Square 1 contains tile *n*
Square 1 empty and Square 4 contains tile *n* →
Square 4 empty and Square 1 contains tile *n*
Square 2 empty and Square 1 contains tile *n* →
Square 1 empty and Square 2 contains tile *n*

A Sample of the Rules for Solving the 8-Puzzle

Reasoning Forward from Initial State:

- Begin building a tree of move sequences that might be solved with initial configuration at root of the tree.
- Generate the next level of the tree by finding all the rules whose left sides match the root node and using their right sides to create the new configurations.
- Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it.
- Continue until a configuration that matches the goal state is generated.

Reasoning Backward from Goal State:

- Begin building a tree of move sequences that might be solved with goal configuration at root of the tree.

- Generate the next level of the tree by finding all the rules whose right sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree.
- Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes.
- Continue until a node that matches the initial state is generated.
- This method of reasoning backward from the desired final state is often called goaldirected reasoning.

To reason forward, the left sides (preconditions) are matched against the current state and the right sides (results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved.

The following 4 factors influence whether it is better to reason Forward or Backward:

1. Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
2. In which direction branching factor (i.e, average number of nodes that can be reached directly from a single node) is greater? We would like to proceed in the direction with lower branching factor.
3. Will the program be used to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
4. What kind of event is going to trigger a problem-solving episode? If it is arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Backward-Chaining Rule Systems

- Backward-chaining rule systems are good for goal-directed problem solving.
- For example, a query system would probably use backward chaining to reason about and answer user questions.
- Unification tries to find a set of bindings for variables to equate a (sub) goal with the head of some rule.
- Medical expert system, diagnostic problems

Forward-Chaining Rule Systems

- Instead of being directed by goals, we sometimes want to be directed by incoming data.
- For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away.
- Rules that match dump their right-hand side assertions into the state and the process repeats.
- Matching is typically more complex for forward-chaining systems than backward ones. ➢ Synthesis systems – Design/Configuration

Example of Typical Forward Chaining

Rules

1) If hot and smoky then ADD fire
2) If alarm_beeps then ADD smoky
3) If fire then ADD switchon_sprinkles

Facts

1) alarm_beeps (given)
2) hot (given)

……

(3) smoky (from F1 by R2)

(4) fire (from F2, F4 by R1)

(5) switch_on_sprinklers (from F2 by R3)

Example of Typical Backward Chaining

Goal: Should I switch on sprinklers?

Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition then forward chain on those concepts to try to deduce the nature and/or cause of the disease.

Now suppose that at some point, the left side of a rule was nearly satisfied – nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident.

Whether it is possible to use the same rules for both forward and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules then the rules will not be reversible.

Logic Programming

- Logic Programming is a programming language paradigm in which logical assertions are viewed as programs.
- There are several logic programming systems in use today, the most popular of which is PROLOG.
- A PROLOG program is described as a series of logical assertions, each of which is a Horn clause.
- A Horn clause is a clause that has at most one positive literal. Thus p, p q, p → q are all Horn clauses.

Programs written in pure PROLOG are composed only of Horn Clauses.
Syntactic Difference between the logic and the PROLOG representations, including:

- *In logic, variables are explicitly quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted.*
  - *The distinction between variables and constants is made in PROLOG by having all variables begin with uppercase letters and all constants begin with lowercase letters.*
- *In logic, there are explicit symbols for and ( ) and or ( ). In PROLOG, there is an explicit symbol for and (,), but there is none for or.*
- *In logic, implications of the form "p implies q" as written as p→q. In PROLOG, the same implication is written "backward" as q: -p.*

Example:

$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \vee dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$
$poodle(ftujfy)$

A Representation In Logic

```
apartmentpet(X) :- pet(X), small(X).
pet(X) :- cat(X).
pet(X) :- dog(X).
dog(X) :- poodle (X) .
small(X) :- poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

A Declarative and a Procedural Representation

The first two of these differences arise naturally from the fact that PROLOG programs are actually sets of Horn Clauses that have been transformed as follows:

1. If the Horn Clause contains no negative literals (i.e., it contains a single literal which is positive), then leave it as it is.
2. Otherwise, return the Horn clause as an implication, combining all of the negative literals into the antecedent of the implication and leaving the single positive literal (if there is one) as the consequent.

This procedure causes a clause, which originally consisted of a disjunction of literals (all but one of which were negative), to be transformed to single implication whose antecedent is a conjunction of (what are now positive) literals.

$\forall x : \forall y : cat(x) \wedge fish(y) \rightarrow likes - to - eat(x,y)$
$\forall x : calico(x) \rightarrow cat(x)$
$\forall x : tuna(x) \rightarrow fish(x)$
$tuna(Charlie)$
$tuna(Herb)$
$calico(Puss)$

(a) Convert these wff's into Horn clauses.
(b) Convert the Horn clauses into a PROLOG program.
(c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.

(a) Horn clauses:

1. $\neg cat(x) \vee \neg fish(y) \vee likes\text{-}to\text{-}eat(x.y)$
2. $\neg calico(x) \vee cat(x)$
3. $\neg tuna(x) \vee fish(x)$
4. $tuna(Charlie)$
5. $tuna(Herb)$
6. $calico(Puss)$

(b) PROLOG program:

```
likestoeat(X,Y)  :- cat(X), fish(Y).
cat(X)  :- calico(X).
fish(X) :- tuna(X).
tuna(charlie).
tuna(herb).
calico(puss).
```

(c) Query:

```
?- likestoeat(puss,X).
```

Answer: charlie

Matching

We described the process of using search to solve problems as the application of appropriate rules to individual problem states to generate new states to which the rules can then be applied and so forth until a solution is found.

How we extract from the entire collection of rules those that can be applied at a given point? To do so requires some kind of matching between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule based system.

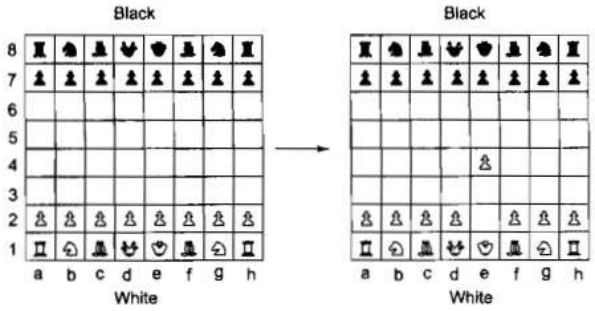
A more complex matching is required when the preconditions of rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others. An even more complex matching process is required if rules should be applied and if their pre condition approximately match the current situation. This is often the case in situations involving physical descriptions of the world.

Indexing

One way to select applicable rules is to do a simple search though all the rules comparing each one's precondition to the current state and extracting all the one's that match. There are two problems with this simple solution:

i. The large number of rules will be necessary and scanning through all of them at every step would be inefficient.
ii. It's not always obvious whether a rule's preconditions are satisfied by a particular state.

Solution: Instead of searching through rules use the current state as an index into the rules and select the matching one's immediately.



One Legal Chess Move

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3)
is empty
AND
Square(file e, rank 4)
is empty
→
move pawn from
Square(file e, rank 2)
to Square(file e, rank 4)

Another Way to Describe Chess Moves

Matching process is easy but at the price of complete lack of generality in the statement of the rules. Despite some limitations of this approach, Indexing in some form is very important in the efficient operation of rule based systems.

Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated conflict resolution strategies to choose among the applicable rules.

While it is possible to apply unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the many-many match problem, in which many

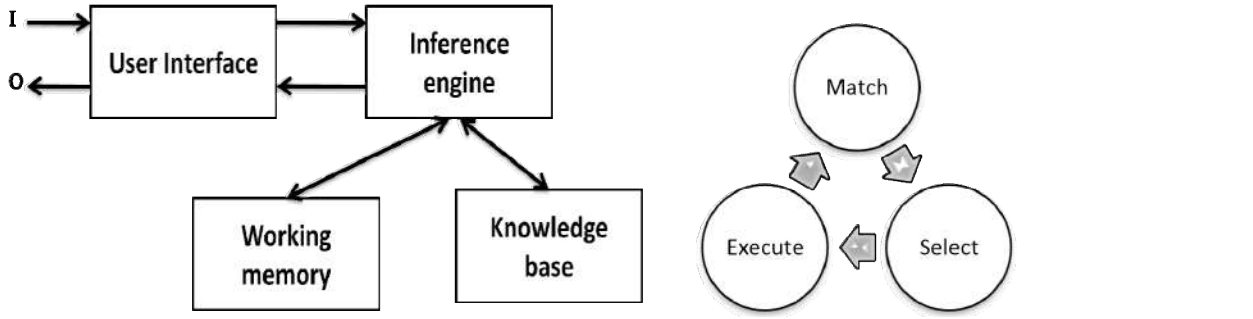
rules are matched against many elements in the state description simultaneously. One efficient many-many match algorithm is RETE.

RETE Matching Algorithm

The matching consists of 3 parts

1. Rules & Productions
2. Working Memory
3. Inference Engine

The inference Engine is a cycle of production system which is match, select, execute.



INFERENCE ENGINE

The above cycle is repeated until no rules are put in the conflict set or until stopping condition is reached. In order to verify several conditions, it is a time consuming process. To eliminate the need to perform thousands of matches of cycles on effective matching algorithm is called RETE.

The Algorithm consists of two Steps.

1. Working memory changes need to be examined.
2. Grouping rules which share the same condition & linking them to their common terms.

RETE Algorithm is many-match algorithm (In which many rules are matched against many elements). RETE uses forward chaining systems which generally employee sophisticated conflict resolution strategies to choose among applicable rules. RETE gains efficiency from 3 major sources.

1. RETE maintains a network of rule condition and it uses changes in the state description to determine which new rules might apply. Full matching is only pursued for candidates that could be affected by incoming/outgoing data.
2. Structural Similarity in rules: RETE stores the rules so that they share structures in memory, set of conditions that appear in several rules are matched once for cycle.
3. Persistence of variable binding consistency. While all the individual preconditions of the rule might be met, there may be variable binding conflicts that prevent the rule from firing.

$$son(Mary, John)\ and\ son\ (Bill, Bob)$$

$$son(x, y)\ ^\ son(y, z) \rightarrow grandparents(x, z)$$

can be minimized. RETE remembers its previous calculations and is able to merge new binding information efficiently.

Approximate Matching:

Rules should be applied if their preconditions approximately match to the current situation

Eg: Speech understanding program

Rules: A description of a physical waveform to phones

Physical Signal: difference in the way individuals speak, result of background noise.

Conflict Resolution:

When several rules matched at once such a situation is called conflict resolution. There are 3 approaches to the problem of conflict resolution in production system.

1. Preference based on rule match:
    a. Physical order of rules in which they are presented to the system
    b. Priority is given to rules in the order in which they appear

2. Preference based on the objects match:
    a. Considers importance of objects that are matched
    b. Considers the position of the match able objects in terms of Long Term Memory (LTM) & Short Term Memory(STM)

    LTM: Stores a set of rules

    STM (Working Memory): Serves as storage area for the facts deduced by rules in long term memory
3. Preference based on the Action:
    a. One way to do is find all the rules temporarily and examine the results of each. Using a Heuristic Function that can evaluate each of the resulting states compare the merits of the result and then select the preferred one.

Search Control Knowledge:

- It is knowledge about which paths are most likely to lead quickly to a goal state
- Search Control Knowledge requires Meta Knowledge.
- It can take many forms. Knowledge about

    which states are more preferable to others.

    which rule to apply in a given situation

    the Order in which to pursue sub goals

    useful Sequences of rules to apply.

MODULE-2

7. Symbolic Reasoning under Uncertainty

We have described techniques for reasoning with a complete, consistent and unchanging model of the world. But in many problem domains, it is not possible to create such models. So here we are going to explore techniques for solving problems with incomplete and uncertain models.

What is reasoning?

- When we require any knowledge system to do something it has not been explicitly told how to do it must *reason.*
- The system must figure out what it needs to know from what it already knows.
    - Reasoning is the act of deriving a conclusion from certain premises using a given methodology. (Process of thinking/ Drawing inference)

How can we Reason?

- To a certain extent this will depend on the knowledge representation chosen.
- Although a good knowledge representation scheme has to allow easy, natural and plausible reasoning.

Broad Methods of how we may reason

- Formal reasoning –
    - Basic rules of inference with logic knowledge representations.
- Procedural reasoning –
    - Uses procedures that specify *how to perhaps solve (sub) problems.*
- Reasoning by analogy –
    - Humans are good at this, more difficult for AI systems. E.g. If we are asked *Can robins fly?. The system might reason that robins are like sparrows and it knows sparrows can fly so ...*
- Generalization and abstraction –
    - Again humans effective at this. This is basically getting towards *learning and understanding methods.*
- Meta-level reasoning –
    - *Once again uses knowledge about what you know and perhaps ordering it in some kind of importance.*

Uncertain Reasoning

- Unfortunately the world is an uncertain place.
- Any AI system that seeks to model and reasoning in such a world must be able to deal with this.
- In particular it must be able to deal with:
    - Incompleteness – compensate for lack of knowledge.

    - Inconsistencies – resolve ambiguities and contradictions.
    - Change – it must be able to update its *world knowledge base over time.*
- Clearly in order to deal with this some decision that a made are more likely to be true (or false) than others and we must introduce methods that can cope with this *uncertainty.*

Monotonic Reasoning

Predicate logic and the inferences we perform on it is an example of monotonic reasoning. In monotonic reasoning if we enlarge at set of axioms we cannot retract any existing assertions or axioms.

A monotonic logic cannot handle
- Reasoning by default
    - Because consequences may be derived only because of lack of evidence of the contrary
- Abductive Reasoning
    - Because consequences are only deduced as most likely explanations.
- Belief Revision
    - Because new knowledge may contradict old beliefs.

Non-Monotonic Reasoning

- Non monotonic reasoning is one in which the axioms and/or the rules of inference are extended to make it possible to reason with incomplete information. These systems preserve, however, the property that, at any given moment, a statement is either
    - believed to be true,
    - believed to be false, or
    - not believed to be either.
- Statistical Reasoning: in which the representation is extended to allow some kind of numeric measure of certainty (rather than true or false) to be associated with each statement.
- In a system doing *non-monotonic reasoning* the set of conclusions may either grow or shrink when new information is obtained.
- Non-monotonic logics are used to formalize plausible reasoning, such as the following inference step:

```
Birds typically fly.
Tweety is a bird.
-------------------------
Tweety (presumably) flies.
```

- Such reasoning is characteristic of commonsense reasoning, where *default rules* are applied when case-specific information is not available. The conclusion of non-monotonic argument may turn out to be wrong. For example, `if Tweety is a penguin, it is incorrect to conclude that Tweety flies.`
- Non-monotonic reasoning often requires jumping to a conclusion and subsequently retracting that conclusion as further information becomes available.
- All systems of non-monotonic reasoning are concerned with the issue of consistency.

- Inconsistency is resolved by removing the relevant conclusion(s) derived previously by default rules.
- Simply speaking, the truth value of propositions in a nonmonotonic logic can be classified into the following types:
    - *facts* that are definitely true, such as "Tweety is a bird"
    - *default rules* that are normally true, such as "Birds fly"
    - *tentative conclusions* that are presumably true, such as "Tweety flies"
- When an inconsistency is recognized, only the truth value of the last type is changed.

Properties of FOPL

- It is complete with respect to the domain of interest.
- It is consistent.
- The only way it can change is that new facts can be added as they become available.
    - If these new facts are consistent with all the other facts that have already have been asserted, then nothing will ever be retracted from the set of facts that are known to be true.
    - This is known as “monotonicity”.

If any of these properties is not satisfied, conventional logic based reasoning systems become inadequate.

Non monotonic reasoning systems, are designed to be able to solve problems in which all of these properties may be missing Issues to be addressed:

- How can the knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?
    - We need to make clear the distinction between
        - It is known that P.
        - It is not known whether P.
    - First-order predicate logic allows reasoning to be based on the first of these. ○ In our new system, we call any inference that depends on the lack of some piece of knowledge a non-monotonic inference.
    - Traditional systems based on predicate logic are monotonic. Here number of statements known to be true increases with time.
    - New statements are added and new theorems are proved, but the previously known statements never become invalid.
- How can the knowledge base be updated properly when a new fact is added to the system(or when the old one is removed)?
    - In Non-Monotonic systems, since addition of a fact can cause previously discovered proofs to become invalid,
        - how can those proofs, and all the conclusions that depend on them be found?
        - Solution: keep track of proofs, which are often called justifications.
    - Such a recording mechanism also makes it possible to support,

        - monotonic reasoning in the case where axioms must occasionally be retracted to reflect changes in the world that is being modeled.
- How can knowledge be used to help resolve conflicts when there are several in consistent non monotonic inferences that could be drawn?
    - It turns out that when inferences can be based
        - on the lack of knowledge as well as on its presence,
        - contradictions are much likely to occur than they were in conventional logical systems in which the only possible contradictions.

Default Reasoning

- Non monotonic reasoning is based on default reasoning or "most probabilistic choice".
    - S is assumed to be true as long as there is no evidence to the contrary.
- Default reasoning (or most probabilistic choice) is defined as follows:
    - Definition 1 : If X is not known, then conclude Y.
    - Definition 2 : If X can not be proved, then conclude Y.
    - Definition 3: If X can not be proved in some allocated amount of time then conclude Y.

Logics for Non-Monotonic Reasoning

- Monotonicity is fundamental to the definition of first-order predicate logic, we are forced to find some alternative to support non-monotonic reasoning.
- We examine several because no single formalism with all the desired properties has yet emerged.
- We would like to find a formalism that does all of the following things:
    - Defines the set of possible worlds that could exist given the facts that we do have.
    - More precisely, we will define an interpretation of a
        - set of wff's to be a domain (a set of objects), together with a function that assigns; to each predicate, a relation;
        - to each n-ary function, an operator that maps from D'' to D; and to each constant, an element of D.
    - A model of a set of wff's is an interpretation that satisfies them.
- Provides a way to say that we prefer to believe in some models rather than others.
- Provides the basis for a practical implementation of this kind of reasoning.
- Corresponds to our intuitions about how this kind of reasoning works.

Default Reasoning

- This is a very common form of non-monotonic reasoning.
- Here we want to draw conclusions based on *what is most likely to be true*.
- Two Approaches to do this
    - Non-Monotonic Logic

    - Default Logic
- Non-Monotonic reasoning is generic descriptions of a class of reasoning.
- Non-Monotonic logic is a specific theory.
- The same goes for Default reasoning and Default logic.

Non-monotonic Logic

- One system that provides a basis for default reasoning is Non-monotonic Logic (NML).
- This is basically an extension of first-order predicate logic to include a modal operator, M.
    - The purpose of this is to allow for consistency.

$$\forall x: \text{plays\_instrument}(x) \wedge M\ \text{improvises}(x) \rightarrow \text{jazz\_musician}(x)$$

states that

- for all *x* is *x* plays an instrument and if the fact that *x* can improvise is consistent with all other knowledge
- then we can conclude that *x* is a jazz musician.

$$\forall x,y: \text{related}(x,y) \wedge M\ \text{GetAlong}(x,y) \rightarrow \text{WillDefend}(x,y)$$

states that

- for all *x* and y, if x and y are related and if the fact that x gets along with y is consistent with everything else that is beleieved,
- then we can conclude that x will defend y

How do we define *consistency*?

One common solution (consistent with PROLOG notation) is to show that fact *P* is true attempt to prove. If we fail we may say that *P* is consistent (since is false). However consider the famous set of assertions relating to President Nixon.

$$\forall x: \text{Republican}(x) \wedge M \neg\text{Pacifist}(x) \rightarrow \neg\text{Pacifist}(x)$$

$$\forall x: \text{Quaker}(x) \wedge M\ \text{Pacifist}(x) \rightarrow \text{Pacifist}(x)$$

Now this states that Quakers tend to be pacifists and Republicans tend not to be. BUT Nixon was both a Quaker and a Republican so we could assert:

Quaker(Nixon)

Republican(Nixon)

This now leads to our total knowledge becoming inconsistent.

What conclusions does the theory actually support?

NML defines the set of theorems that can be derived from a set of wff's A to be the intersection of the sets of theorems that result from the various ways in which the wff's of A might be combined.

Default Logic

An alternative logic for performing default based reasoning is Reiter's Default Logic (DL). Default logic introduces a new inference rule of the form:

$$\frac{A : B}{C}$$

which states if A is provable and it is consistent to assume B then conclude C.

Now this is similar to Non-monotonic logic but there are some distinctions:

> New inference rules are used for computing the set of plausible extensions. So in the Nixon example above Default logic can support both assertions since is does not say anything about how choose between them -- it will depend on the inference being made. In Default logic any nonmonotonic expressions are rules of inference rather than expressions. They cannot be manipulated by the other rules of inference. This leads to some unexpected results.

In Default Logic, A indicates prerequisite, B indicates justification, and C indicates Consequence.

Example: Typically "An American adult owns a car."

$$\frac{American(x) \wedge Adult(x) : M(\exists y\ car(y) \wedge owns(x, y))}{\exists y\ car(y) \wedge owns(x, y)}$$

If we can prove from our beliefs that x is American and adult and believing that there is some car that is owned by x does not lead to an inconsistency.

Inheritance:

One very common use of nonmonotonic reasoning is as a basis for inheriting attribute values from a prototype description of a class to the individual entities that belong to the class. Considering the Baseball example in Inheritable Knowledge, and try to write its inheriting knowledge as rules in DL.

We can write a rule to account for the inheritance of a default value for the height of a baseball player as:

$$\frac{Baseball\text{-}Player(x) : height(x, 6\text{-}1)}{height(x, 6\text{-}1)}$$

Now suppose we assert Pitcher(Three-Finger-Brown). Since this enables us to conclude that Three-Finger-Brown is a baseball player, our rule allows us to conclude that his height is 6-1. If, on the other hand, we had asserted a conflicting value for Three Finger had an axiom like:

$$\forall x, y, z : height(x, y) \wedge height(x, z) \rightarrow y = z,$$

Which prohibits someone from having more than one height, then we would not be able to apply the default rule. Thus an explicitly stated value will block the inheritance of a default value, which is exactly what we want.

Let's encode the default rule for the height of adult males in general. If we pattern it after the one for baseball players, we get

$$\frac{Adult\text{-}Male(x) : height(x, 5\text{-}10)}{height(x, 5\text{-}10)}$$

Unfortunately, this rule does not work as we would like. In particular, if we again assert Pitcher(Three-Finger-Brown) then the resulting theory contains 2 extensions: one in which our first rule fires and brown's height is 6-1 and one in which this rule applies and Brown's height is 510. Neither of these extensions is preferred. In order to state that we prefer to get a value from the more specific category, baseball player, we could rewrite the default rule for adult males in general as:

$$\frac{Adult\text{-}Male(x) : \neg Baseball\text{-}Player(x) \wedge height(x, 5\text{-}10)}{height(x, 5\text{-}10)}$$

This effectively blocks the application of the default knowledge about adult males in this case that more specific information from the class of baseball players is available. Unfortunately, this approach can become widely as the set of exceptions to the general rule increases. We would end up with a rule like:

$$\frac{Adult\text{-}Male(x) : \neg Baseball\text{-}Player(x) \wedge \neg Midget(x) \wedge - Jockey(x) \wedge height(x, 5\text{-}10)}{height(x, 5\text{-}10)}$$

A clearer approach is to say something like. Adult males typically have a height of 5-10 unless they are abnormal in some way. We can then associate with other classes the information that they are abnormal in one or another way. So we could write, for example:

$\forall x$: $Adult\text{-}Male(x) \wedge \neg AB(x, aspect1) \rightarrow height(x, 5\text{-}10)$
$\forall x : Baseball\text{-}Player(x) \rightarrow AB(x, aspect\ 1)$
$\forall x : Midget(x) \rightarrow AB(x, aspect\ 1)$
$\forall x : Jockey(x) \rightarrow AB(x, aspect\ 1)$

Then, if we add the single default rule:

$$\frac{: \neg AB(x, y)}{\neg AB(x, y)}$$

we get the desired result.

Abduction

Abductive reasoning is to abduce (or take away) a logical assumption, explanation, inference, conclusion, hypothesis, or best guess from an observation or set of observations. Because the conclusion is merely a best guess, the conclusion that is drawn may or may not be true. Daily decision-making is also an example of abductive reasoning.

Standard logic performs deductions. Given 2 axioms

- $\forall x : A(x) \rightarrow B(x)$, $A(C)$, We conclude $B(C)$ using deduction

$\forall x$: Measles(x) → Spots(x), Having measles implies having spots

If we notice Spots, we might like to conclude measles, but it may be wrong. But may be a best guess, we can make about what is going on. Deriving conclusions in this way is abductive reasoning (a form of default reasoning).

- Given 2 wff's (A→B) & (B), for any expression A & B, if it is consistent to assume A, do so.

Minimalist Reasoning

We describe methods for saying a very specific and highly useful class of things that are generally true. These methods are based on some variant of the idea of a minimal model. We will define a model to be minimal if there are no other models in which fewer things are true. The idea behind using minimal models as a basis for non-monotonic reasoning about the world is the following –

- *There are many fewer true statements than false ones.*
- *If something is true and relevant it makes sense to assume that it has been entered into our knowledge base.*
- *Therefore, assume that the only true statements are those that necessarily must be true in order to maintain the consistency.*

The Closed World Assumption

- CWA
  - Simple kind of minimalist reasoning.
  - says that the only objects that satisfy any predicate P are those that must.
  - Eg. A company's employee database, Airline example
- CWA is powerful as a basis for reasoning with Databases, which are assumed to be complete with respect to the properties they describe.
- Although the CWA is both simple & powerful, it can fail to produce an appropriate answer for either of the two reasons.
  - The assumptions are not always true in the world; some parts of the world are not realistically "closable". - unrevealed facts in murder case
  - It is a purely syntactic reasoning process. Thus, the result depends on the form of assertions that are provided.

Consider a KB that consists of just a single statement A(Joe) v B(Joe)

The CWA allows us to conclude both ?A(Joe) and ?B(Joe), since neither A nor B must necessarily be true of Joe.

The extended KB

(

$A(Joe) \vee B(Joe)$

$\neg A(Joe)$

$\neg B(Joe)$ is inconsistent.

The problem is that we have assigned a special status to the positive instances of predicates as opposed to negative ones. CWA forces completion of KB by adding negative assertion whenever it is consistent to do so.

CWA captures part of the idea that anything that must not necessarily be true should be assumed to be false, it does not capture all of it.

It has two essential limitations:

- It operates on individual predicates without considering interactions among predicates that are defined in the KB.
- It assumes that all predicates have all their instances listed. Although in many database applications this is true, in many KB systems it is not.

Circumscription

- *Circumscription* is a rule of conjecture (*conclusion formed on the basis of incomplete information*) that allows you
  - to jump to the conclusion that the objects you can show that posses a certain property, *p*, are in fact all the objects that posses that property.
- Circumscription can also cope with default reasoning. Several theories of circumscription have been proposed to deal with the problems of CWA.
- Circumscription together with first order logic allows a form of Non-monotonic Reasoning.

Suppose we know:

$$bird(tweety)$$
$$\forall x: penguin(x) \rightarrow bird(x)$$
$$\forall x: penguin(x) \rightarrow \neg flies(x)$$

And we wish to add the fact that *typically, birds fly*.

In circumscription this phrase would be stated as:

A bird will flyif it is not abnormal

and can thus be represented by:

$$\forall x: \quad ( ) \wedge \neg \quad ( ) \quad ' \quad ( ).$$

However, this is not sufficient. We cannot conclude

( ) since we cannot prove ( ).

This is where we apply circumscription and, in this case, *we will assume that those things that are shown to be abnormal are the only things to be abnormal.* Thus we can rewrite our *default rule* as:

$$\forall x: \quad ( ) \qquad \wedge \neg \qquad ( ) \neg \qquad ( )$$

and add the following

$$\forall x: \neg \qquad ( )$$

since there is nothing that cannot be shown to be abnormal.

If we now add the fact: ( ) Clearly we can prove ( ).

If we circumscribe abnormal now we would add the sentence, *penguin (tweety) is the abnormal thing*:

$$\forall x: \quad P(x) \rightarrow \quad Q(x).$$

Note the distinction between Default logic and circumscription:

- *Defaults are sentences in language itself not additional inference rules*.

Dependency Directed Backtracking

- To reduce the computational cost of non-monotonic logic, we need to be able to avoid re-searching the entire search space when a new piece of evidence is introduced
  - otherwise, we have to backtrack to the location where our assumption was introduced and start searching anew from there
- In dependency directed backtracking, we move to the location of our assumption, make the change and propagate it forward from that point without necessarily having to research from scratch
  - as an example, you have scheduled a meeting on Tuesday at 12:15 because everyone indicated that they were available
  - but now, you cannot find a room, so you backtrack to the day and change it to Thursday, but you do not re-search for a new time because you assume if everyone was free on Tuesday, they will be free on Thursday as well

Implementations: Truth Maintenance Systems

A variety of *Truth Maintenance Systems* (TMS) have been developed as a means of implementing Non-Monotonic Reasoning Systems.

- tracking the order in which sentences are told to the knowledge base by numbering them, this implies that the KB will be consistent.

The idea of truth maintenance system arose as a way of providing the ability to do dependencydirected backtracking and so to support nonmonotonic reasoning.

Types of TMS:

- justification-based TMS (JTMS)
- assumption-based TMS (ATMS)
- logic-based TMS (LTMS)

Basically TMSs:

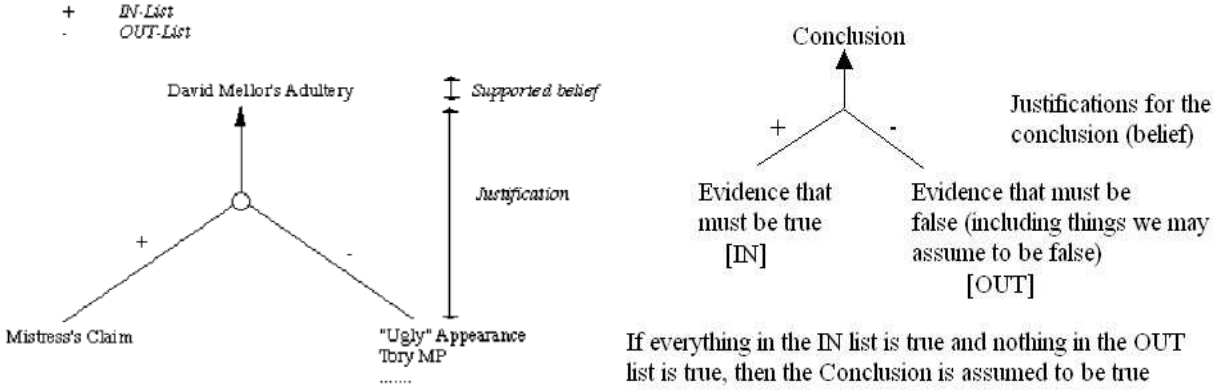
- all do some form of dependency directed backtracking Assertions are connected via a network of dependencies.

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- JTMS is one element of a TMS design space, a good model for most dependency systems and can quickly focus on how to use it.
- This TMS itself does not know anything about the structure of the assertions themselves.
- The only role is to serve as a bookkeeper for a separate problem solving system which in turn provides it with both assertions and dependencies among assertions.

How JTMS works?

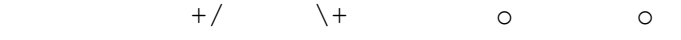
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:
  - An *IN-List* -- which supports beliefs held.
  - An *OUT-List* -- which supports beliefs *not* held.
- An assertion is connected to its justification by an arrow.
- One assertion can *feed* another justification thus creating the network.
- Assertions may be labelled with a *belief status*.
- An assertion is *valid* if every assertion in the IN-List is believed and none in the OUT-List are believed.
- An assertion is non-monotonic is the OUT-List is not empty or if any assertion in the IN-List is non-monotonic.



A Justification-based truth maintenance system (JTMS) is a simple TMS where one can examine the consequences of the current set of assumptions. In JTMS labels are attached to arcs from sentence nodes to justification nodes. This label is either "+" or "-". Then, for a justification node we can talk of its *IN-LIST,* the list of its inputs with "+" label, and of its *OUT-LIST*, the list of its inputs with "-" label.

The meaning of sentences is not known. We can have a node representing a sentence p and one representing ~p and the two will be totally unrelated, unless relations are established between them by justifications. For example, we can write:

```
          ~p^p Contradiction Node
            o
|
            x      'x' denotes a justification node
          /   \          'o' denotes a sentence node
```



p ~p which says that if both p and ~p are IN we have a contradiction.

The association of IN or OUT labels with the nodes in a dependency network defines an *in-outlabeling function*. This function is consistent if:

- The label of a junctification node is IN iff the labels of all the sentence nodes in its in-list are all IN and the labels of all the sentence nodes in its out-list are OUT.
- The label of a sentence node is IN iff it is a premise, or an enabled assumption node, or it has an input from a justification node with label IN.

A set of important reasoning operations that a JTMS does not perform, including:

- Applying rules to derive conclusions
- Creating justifications for the results of applying rules
- Choosing among alternative ways of resolving a contradiction
- Detecting contradictions

All of these operations must be performed by the problem-solving program that is using the JTMS.

Logic-Based Truth Maintenance Systems (LTMS)
Similar to JTMS except:

- Nodes (assertions) assume no relationships among them except ones explicitly stated in justifications.
- JTMS can represent P and P IN simultaneously. No contradiction will be detected automatically. An LTMS would throw a contradiction automatically in such a case here.
- If this happens network has to be reconstructed.

Assumption-Based Truth Maintenance Systems (ATMS)

- JTMS and LTMS pursue a single line of reasoning at a time and backtrack (dependencydirected) when needed → *depth first search*.
- ATMS maintain alternative paths in parallel → *breadth-first search*
- Backtracking is avoided at the expense of maintaining multiple contexts.
- However as reasoning proceeds contradictions arise and the ATMS can be *pruned* ○ Simply find assertion with no valid justification.

The ATMS like the JTMS is designed to be used in conjunction with a separate problem solver. The problem solver's job is to:

- Create nodes that correspond to assertions (both those that are given as axioms and those that are derived by the problem solver).
- Associate with each such node one or more justifications, each of which describes reasoning chain that led to the node.
- Inform the ATMS of inconsistent contexts.

This is identical to the role of the problem solver that uses a JTMS, except that no explicit choices among paths to follow need to be made as reasoning proceeds. Some decision may be necessary at the end, though, if more than one possible solution still has a consistent context.

The role of the ATMS system is then to:

- Propagate inconsistencies, thus ruling out contexts that include subcontexts (set of assertions) that are known to be inconsistent.
- Label each problem solver node with the contexts in which it has a valid justification. This is done by combining contexts that correspond to the components of a justification. In particular, given a justification of the form

  1 2 … →

  assign as a context for the node corresponding to C the intersection of the contexts corresponding to the nodes A1 through An.

Contexts get eliminated as a result of the problem-solver asserting inconsistencies and the ATMS propagating them. Nodes get created by the problem-solver to represent possible components of a problem solution. They may then get pruned from consideration if all their context labels get pruned.

8. Statistical Reasoning

Introduction:

Statistical Reasoning: The reasoning in which the representation is extended to allow some kind of numeric measure of certainty (rather than true or false) to be associated with each statement. A fact is believed to be true or false. For some kind of problems, describe beliefs that are not certain but for which there is a supporting evidence.

There are 2 class of problems:

- First class contain problems in which there is genuine randomness in the word.
    - Example: Cards Playing
- Second class contains problems that could in principle be modeled using the technique we described (i.e. resolution from predicate logic) ○ Example: Medical Diagnosis

Probability & Baye's Theorem

An important goal for many problem-solving systems is to collect evidence as the system goes along and to model its behavior on the basis of the evidence. To model this behavior, we need statistical theory of evidence. Bayesian statistics is such a theory. The fundamental notion of Bayesian statistics is that of conditional probability. Conditional Probability is the probability of an event given that another event has occurred.

Read this expression as the probability of hypothesis H given that we have observed evidence E. To compute this, we need to take into account the prior probability of H and the extent to which E provides evidence of H.

$$P(H_i \backslash E) = \frac{P(E \mid H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E \mid H_n) \cdot P(H_n)}$$

- P(H/E) = probability of hypothesis H given that we have observed evidence E
- P(Hi/E) = probability of hypothesis Hi is true under the evidence E
- P(E/Hi) = probability that we will observe evidence E given that hypothesis Hi is true
- P(Hi) = a priori probability that hypothesis is true in absence of any specific evidence
- k = number of possible hypothesis

Suppose, for example, that we are interested in examining the geological evidence at a particular location to determine whether that would be a good place to dig to find a desired mineral. If we know the prior probabilities of finding each of the various minerals and we know the probabilities that if a mineral is present then certain physical characteristics will be observed, then we use the Baye's formula to compute from the evidence we collect, how likely it is that the various minerals are present.

The key to using Baye's theorem as a basis for uncertain reasoning is to recognize exactly what it says.

Consider the following puzzle:

> A pea is placed under one of three shells, and the shells are then manipulated in such a fashion that all three appear to be equally likely to contain the pea. Nevertheless, you win a prize if you guess the correct shell, so you make a guess. The person running the game does know the correct shell, however, and uncovers one of the shells that you did not choose and that is empty. Thus, what remains are two shells: one you chose and one you did not choose. Furthermore, since the uncovered shell did not contain the pea, one of the two remaining shells does contain it. You are offered the opportunity to change your selection to the other shell. Should you?

Work through the conditional probabilities mentioned in this problem using Bayes' theorem. What do the results tell about what you should do'

Let $P(G) =$ probability that my initial guess is right.
$P(O) =$ probability that one of the other 2 shells has the pea.
$P(T) =$ Probability that a shell without a pea will be turned over.

$P(G) = \frac{1}{3}$

$P(O) = \frac{2}{3}$

$P(T) = 1$

The important fact here is that $P(T) = 1$, i.e., it is certain that a shell without a pea will be turned over. So the turning over provides no new information. So:

$P(G|T) = P(G) = \frac{1}{3}$

$P(O|T) = P(O) = \frac{2}{3}$

But O now contains only one shell that hasn't been turned over. So the probability that that shell contains the pea is $\frac{2}{3}$. You should change your guess to that other shell.

To see what is really going on here, we can contrast this problem with one in which, after you make your guess, the other person turns over one of the other shells are random (i.e., without knowing where the pea is). Then:

$P(G) = \frac{1}{3}$

$P(O) = \frac{2}{3}$

$P(T) = \frac{2}{3}$

$P(G|T) = \frac{P(G \wedge T)}{P(T)} = \frac{\frac{1}{3}}{\frac{2}{3}} = \frac{1}{2}$

We got this by using the simplest form of Bayes rule and observing that if your guess was right, then the other person is certain to pick a shell without a pea, so $P(G \wedge T)$ is just $P(G)$.

Suppose we are solving a medical diagnosis problem. Consider the following assertions:

:

:

:

- Without any additional evidence, the presence of spots serves as evidence in favor of measles. It also serves as evidence of fever since measles would cause fever.
- Suppose we already know that the patient has measles. Then the additional evidence that he has spots actually tells us nothing about fever.
- Either spots alone or fever alone would constitute evidence in favor of measles.
- If both are present, we need to take both into account in determining the total weight of evidence.

In a clinic, the probability of the patients having HIV virus is **0.15**.

A blood test done on patients :

If patient has virus, then the test is **+ve** with probability **0.95**.

If the patient does not have the virus, then the test is **+ve** with probability **0.02**.

Assign labels to events : **H** = patient has virus; **P** = test +ve

Given : $P(H) = 0.15$; $P(P|H) = 0.95$; $P(P|\neg H) = 0.02$

Find :

If the test is **+ve** what are the probabilities that the patient

i) has the virus ie $P(H|P)$; ii) does not have virus ie $P(\neg H|P)$;

If the test is **-ve** what are the probabilities that the patient

iii) has the virus ie $P(H|\neg P)$; iv) does not have virus ie $P(\neg H|\neg P)$;

Calculations :

i) For $P(H|P)$ we can write down Bayes Theorem as

$P(H|P) = [\, P(P|H)\, P(H)\,] \,/\, P(P)$

We know $P(P|H)$ and $P(H)$ but not $P(P)$ which is probability of a **+ve** result.

There are two cases, that a patient could have a **+ve** result, stated below :

1. Patient has virus and gets a **+ve** result : $H \cap P$
2. Patient does not have virus and gets a **+ve** result: $\neg H \cap P$

Find probabilities for the above two cases and then add

Find probabilities for the above two cases and then add

ie $P(P) = P(H \cap P) + P(\neg H \cap P)$.

But from the second axiom of probability we have :

$P(H \cap P) = P(P|H)\, P(H)$ and $P(\neg H \cap P) = P(P|\neg H)\, P(\neg H)$.

Therefore putting these we get :

$$P(P) = P(P|H)\, P(H) + P(P|\neg H)\, P(\neg H) = 0.95 \times 0.15 + 0.02 \times 0.85 = 0.1595$$

Now substitute this into Bayes Theorem and obtain $P(H|P)$

$$P(H|P) = \frac{P(P|H)\, P(H)}{P(P|H)\, P(H) + P(P|\neg H)\, P(\neg H)} = 0.95 \times 0.15 \,/\, 0.1595 = 0.8934$$

ii) Next is to work out $P(\neg H|P)$

$$P(\neg H|P) = 1 - P(H|P) = 1 - 0.8934 = 0.1066$$

iii) Next is to work out $P(H|\neg P)$; again we write down Bayes Theorem

$$P(H|\neg P) = \frac{P(\neg P|H)\, P(H)}{P(\neg P)}$$ here we need $P(\neg P)$ which is $1 - P(P)$

$$= (0.05 \times 0.15)/(1-0.1595) = 0.008923$$

iv) Finally, work out $P(\neg H|\neg P)$

It is just $1 - P(H|\neg P) = 1 - 0.008923 = 0.99107$

Disadvantages with Baye's Theorem

- The size of set of joint probability that we require in order to compute this function grows as 2^n if there are n different propositions considered.
- Baye's Theorem is hard to deal with for several reasons:
    - Too many probabilities have to be provided
    - the space that would be required to store all the probabilities is too large.
    - time required to compute the probabilities is too large.

Mechanisms for making easy to deal with uncertain reasoning

- Attaching Certainty factor to rules
- Bayesian Networks
- Dempster-Shafer Theory
- Fuzzy Logic

Certainty Factors and Rule-Based Systems

The certainty-factor model was one of the most popular model for the representation and manipulation of uncertain knowledge in the early 1980s Rule-based expert systems. Expert systems are an example for the certainty factors.

We describe one practical way of compromising on a pure Bayesian system. MYCIN system is an example of an expert system, since it performs a task normally done by a human expert. MYCIN system attempts to recommend appropriate therapies for patients with bacterial infections. It interacts with the physician to acquire the clinical data it needs. We concentrate on the use of probabilistic reasoning.

MYCIN represents most of its diagnostic knowledge as a set of rules. Each rule has associated with it a certainty factor, which is a measure of the extent to which the evidence is described by antecedent of the rule, supports the conclusion that is given in the rule's consequent. It uses backward reasoning to the clinical data available from its goal of finding significant diseasecausing organisms.

What do Certainty Factor Mean?

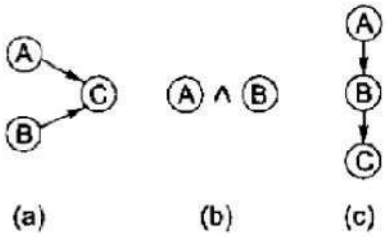
- It is an expert estimate of degree of belief or disbelief in an evidence hypothesis relation.
- A certainty factor (CF[h,e]) is defined in terms of two components ◦ MB [h, e]:
    - A measure between 0 & 1 of belief in hypothesis h given the evidence e.
    - MB measures the extent to which the evidence supports the hypothesis
    - MB=0, if the evidence fails to support hypothesis ◦ MD [h, e]:
    - A measure between 0 & 1 of disbelief in hypothesis h given by the evidence 'e'
    - MD measures the extent to which the evidence does not support hypothesis
    - MD=0, if the evidence supports the hypothesis.

[,] = [,] – [,]

Any particular piece of evidence either supports or denies a hypothesis (but not both), a single number suffices for each rule to define both the MB and MD and thus the CF. CF's reflect assessments of the strength of the evidence in support of the hypothesis.

CF's need to be combined to reflect the operation of multiple pieces of evidence and multiple rules applied to a problem. The combination scenarios are:

1. Several rules all provide evidence that relates to a single hypothesis
2. Our belief is a collection of several propositions taken together
3. The output of one rule provides the input to another



We must first need to describe some properties that we like combining functions to satisfy:

- Combining function should be commutative and Associative
- Until certainty is reached additional conforming evidence should increase MB
- If uncertain inferences are chained together then the result should be less certain than either of the inferences alone

Several rules provide evidences that related to single hypothesis

The measure of belief and disbelief of a hypothesis given two observations s1 and s2 are computed from:

$$MB[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MD[h, s_1 \wedge s_2] = 1 \\ MB[h, s_1] + MB[h, s_2] \cdot (1 - MB[h, s_1]) & \text{otherwise} \end{cases}$$

$$MD[h, s_1 \wedge s_2] = \begin{cases} 0 & \text{if } MB[h, s_1 \wedge s_2] = 1 \\ MD[h, s_1] + MD[h, s_2] \cdot (1 - MD[h, s_1]) & \text{otherwise} \end{cases}$$

One way to state these formulas in English is that

- The measure of belief in h is 0 if h is disbelieved with certainty.
- Otherwise, the measure of belief in h given two observations is the measure of belief given only one observation plus some increment for the second observation.
- This increment is computed by first taking the difference 1 (certainty) and the belief given only the first observation.
- This difference is the most that can be added by the second observation. The difference is then scaled by the belief in h given only the second observation.

From MB and MD, CF can be computed. If several sources of corroborating evidence are pooled, the absolute value of CF will increase. If conflicting evidence is introduced, the absolute value of CF will decrease.

A simple example shows how these functions operate. Suppose we make an initial observation that confirms our belief in h with $MB = 0.3$. Then $MD[h,s_1] = 0$ and $CF[h, s_1] = 0.3$. Now we make a second observation, which also confirms h, with $MB[h,s_2] = 0.2$. Now:

$$\begin{aligned} MB[h,s_1 \wedge s_2] &= 0.3 + 0.2 \cdot 0.7 \\ &= 0.44 \\ MD[h,s_1 \wedge s_2] &= 0.0 \\ CF[h,s_1 \wedge s_2] &= 0.44 \end{aligned}$$

Using MYCIN's rules for inexact reasoning, compute *CF*, *MB*, and *MD* of h_1 given three observations where

$$CF(h_1, o_1) = 0.5$$
$$CF(h_1, o_2) = 0.3$$
$$CF(h_1, o_3) = -0.2$$

$$MB[h_1, o_1] = 0.5$$

$$MB[h_1, o_1 \wedge o_2] = 0.5 + (0.3 \times (1 - 0.5)) = 0.65$$

$$MB[h_1, o_1 \wedge o_2 \wedge o_3] = 0.65$$

$$MD[h_1, o_1] = 0$$

$$MD[h_1, o_1 \wedge o_2] = 0$$

$$MD[h_1, o_1 \wedge o_2 \wedge o_3] = 0.2$$

$$CF[h_1, o_1 \wedge o_2 \wedge o_3] = 0.65 - 0.2 = 0.45.$$

Our belief is a collection of several propositions taken together

We need to compute the certainty factor of a combination of hypothesis. This is necessary when we need to know the certainty factor of a rule antecedent that contains several clauses. The combination certainty factor can be computed from its MB and MD. The formula for the MB of the conjunction {condition of being joined, proposition resulting from the combination of two or more propositions using the ^ operator} and disjunction {proposition resulting from the combination of two or more propositions using the v (OR) operator} of two hypotheses are:

$$[1 \quad 2, \quad] = \min([1, \quad], \quad [2, \quad])$$
$$[1 \quad 2, \quad] = \max([1, \quad], \quad [2, \quad])$$

MD can be computed analogously.

Output of one rule provides the input to another

In this rules are chained together with the result that the uncertain outcome of one rule must provide the input to another. The solution to this problem will also handle the case in which we must assign a measure of uncertainty to initial inputs. This could easily happen in situations where the evidence is the outcome of an experiment or a laboratory test whose results are not completely accurate.

The certainty factor of the hypothesis must take into account both the strength with which the evidence suggests the hypothesis and the level of confidence in the evidence. Let MB'[h,s] be the measure of belief in h given that we are absolutely sure of the validity of s. Let e be the

evidence that led us to believe in s (for example, the actual readings of the laboratory instruments or

results of applying other rules). Then:

$$MB[h, s] = MB'[h, s] \cdot \max(0, CF[s, e])$$

MB which can be thought of as a proportionate decrease in disbelief in h as a result of e as:

$$MB[h, e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \dfrac{\max[P(h|e), P(h)] - P(h)}{1 - P(h)} & \text{otherwise} \end{cases}$$

MD is the proportionate decrease in belief in h as a result of e

$$MD[h, e] = \begin{cases} 1 & \text{if } P(h) = 0 \\ \dfrac{\min[P(h|e), P(h)] - P(h)}{-P(h)} & \text{otherwise} \end{cases}$$

It turns out that these definitions are incompatible with a Bayesian view of conditional probability. Small changes to them however make them compatible. We can redefine MB as

$$MB[h, e] = \begin{cases} 1 & \text{if } P(h) = 1 \\ \dfrac{\max[P(h|e), P(h)] - P(h)}{(1 - P(h) \cdot P(h|e)} & \text{otherwise} \end{cases}$$

The definition of MD must also be changed similarly.

MYCIN uses CF. The CF can be used to rank hypothesis in order of importance. Example, if a patient has certain symptoms that suggest several possible diseases. Then the disease with higher CF would be investigated first. If E then H → CF(rule) = level of belief of H given E.

Example: CF(E) = CF(it will probably rain today) = 0.6 Positive
CF means evidence supports hypothesis.

MYCIN Formulas for all three combinations:

(i) Make the assumptions that all the rules are independent (ii)
The burden of guarantee independence is on rule writer

(iii) If each combination of scenarios are considered then independent assumption is violated because of large volumes of conditions

The first scenario (a), Our example rule has three antecedents with a single CF rather than three separate rules; this makes the combination rules unnecessary. The rule writer did this because the three antecedents are not independent.

To see how much difference MYCIN's independence assumption can make, suppose for the moment that we had instead had three separate rules and that the CF of each was 0.6. This could happen and still be consistent with the combined CF of 0.7 if three conditions overlap

$$\begin{aligned} MB[h, s_1 \wedge s_2] &= 0.6 + (0.6 \cdot 0.4) \\ &= 0.84 \\ MB[h, (s_1 \wedge s_2) \wedge s_3] &= 0.84 + (0.6 \cdot 0.16) \\ &= 0.936 \end{aligned}$$

substantially. If we apply the MYCIN combination formula to the three separate rules, we get

This is a substantially different result than the true value, as expressed by the expert of 0.7.

Let's consider what happens when independence assumptions are violated in the scenario of (c):

S: sprinkler was on last night
W: grass is wet
R: it rained last night

We can write MYCIN-style rules that describe predictive relationships among these three events:

```
If:  the sprinkler was on last night
then there is suggestive evidence (0.9) that
     the grass will be wet this morning
```

Taken alone, this rule may accurately describe the world. But now consider a second rule:

```
If:  the grass is wet this morning
then there is suggestive evidence (0.8) that
     it rained last night
```

Taken alone, this rule makes sense when rain is the most common source of water on the grass. But if the two rules are applied together, using MYCIN's rule for chaining, we get

$MB[W,S] = 0.8$ {sprinkler suggests wet}
$MB[R,W] = 0.8 \cdot 0.9 = 0.72$ {wet suggests rains}

BAYESION NETWORKS

CFs is a mechanism for reducing the complexity of a Bayesian reasoning system by making some approximations to the formalism. Bayesian networks in which we preserve the formalism and rely instead on the modularity of the world we are trying to model. Bayesian Network is also called Belief Networks.

The basic idea of Bayesian Network is knowledge in the world is modular. Most events are conditionally independent of other events. Adopt a model that can use local representation to allow interactions between events that only affect each other. The main idea is that to describe the real world it is not necessary to use a huge list of joint probabilities table in which list of probabilities of all conceivable combinations of events. Some events may only be unidirectional others may be bidirectional events may be casual and thus get chained tighter in network.

Implementation:

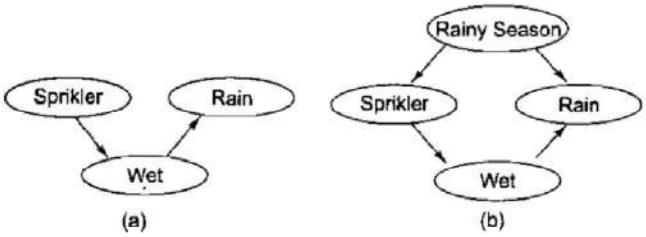
A Bayesian Network is a directed acyclic graph. A graph where the directions are links which indicate dependencies that exist between nodes. Nodes represent propositions about events or events themselves. Conditional probabilities quantify the strength of dependencies.

Eg: Consider the following facts

S: Sprinklers was on the last night

W: Grass is wet

R: It rained last night



From the above diagram, Sprinkler suggests Wet and Wet suggests Rain. (a) shows the flow of constraints.

There are two different ways that propositions can influence the likelihood of each other.

- The first is that causes. Influence the likelihood of their symptoms.
- The second is that the symptoms affect the likelihood of all of its possible causes.

Rules:

(i) If the sprinkler was ON last night then the grass will be wet this morning

(ii) If grass is wet this morning then it rained last night

(iii) By chaining (if two rules are applied together) we believe that it rained because we believe that sprinkler was ON.

The idea behind the Bayesian network structure is to make a clear distinction between these two kinds of influence.

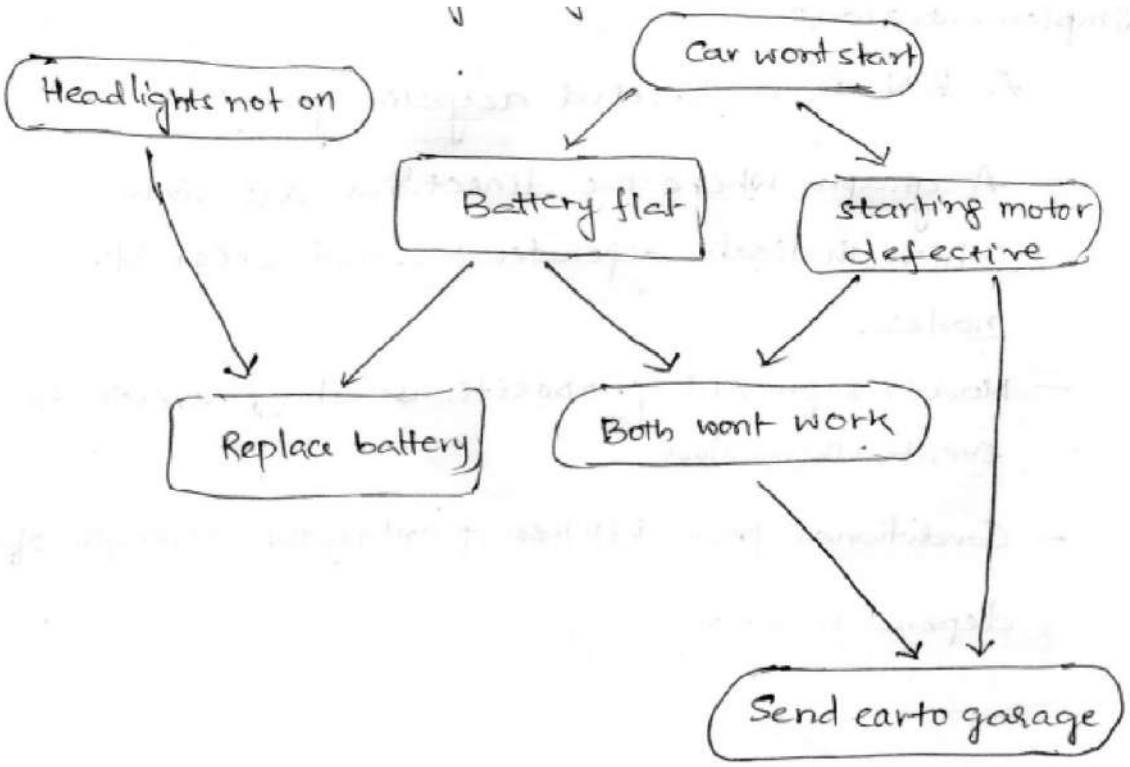
Bayesian Network Example:

If my car won't start then it is likely that

- battery is flat (or)
- starting motor is broken

In order to decide whether to fix the car myself or send it to the garage I make the following decision:

- If the headlights do not work then the battery is likely to be flat so i fix it myself.
- If the starting motor is defective then send car to garage.
- If battery and starting motor both gone send car to garage



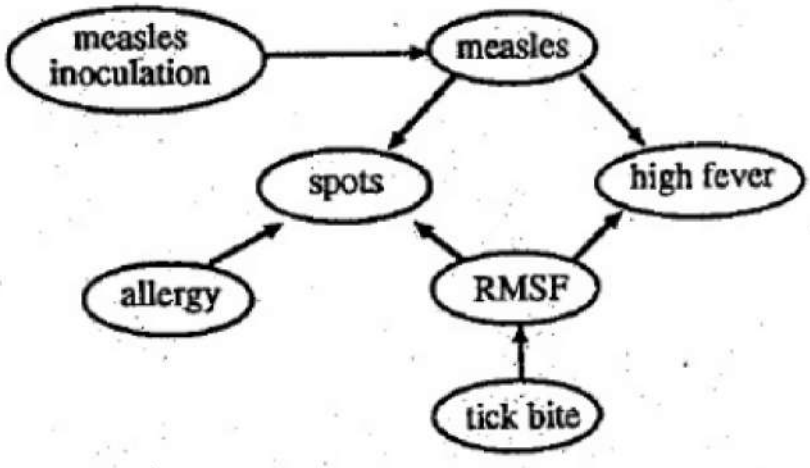
Consider the following set of propositions:

patient has spots
patient has measles
patient has high fever

patient has Rocky Mountain Spotted Fever patient has previously been innoculated against measles patient was recently bitten by a tick

patient has an allergy

Create a network that defines the casual connections among these nodes.



We have four binary elements, represent by four random variables.

W = true : Grass is wet

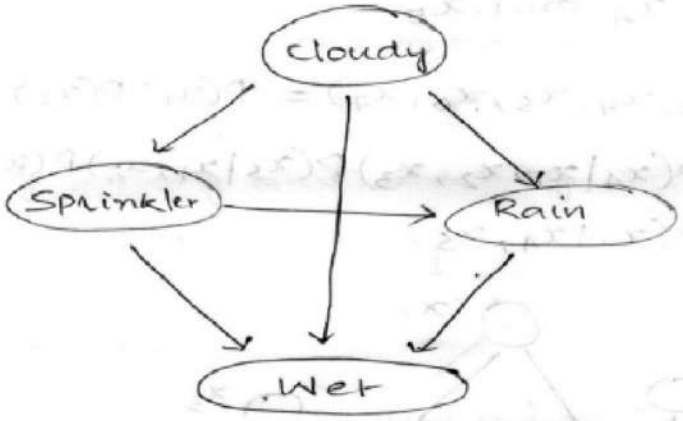
S = true : Water Sprinkler is on

R = true : It is raining

C = true : It is cloudy

By chain rule

$$P(C, S, R, W) = P(C)\, P(S|C)\, P(R|C, S)\, P(W|C, S, R)$$

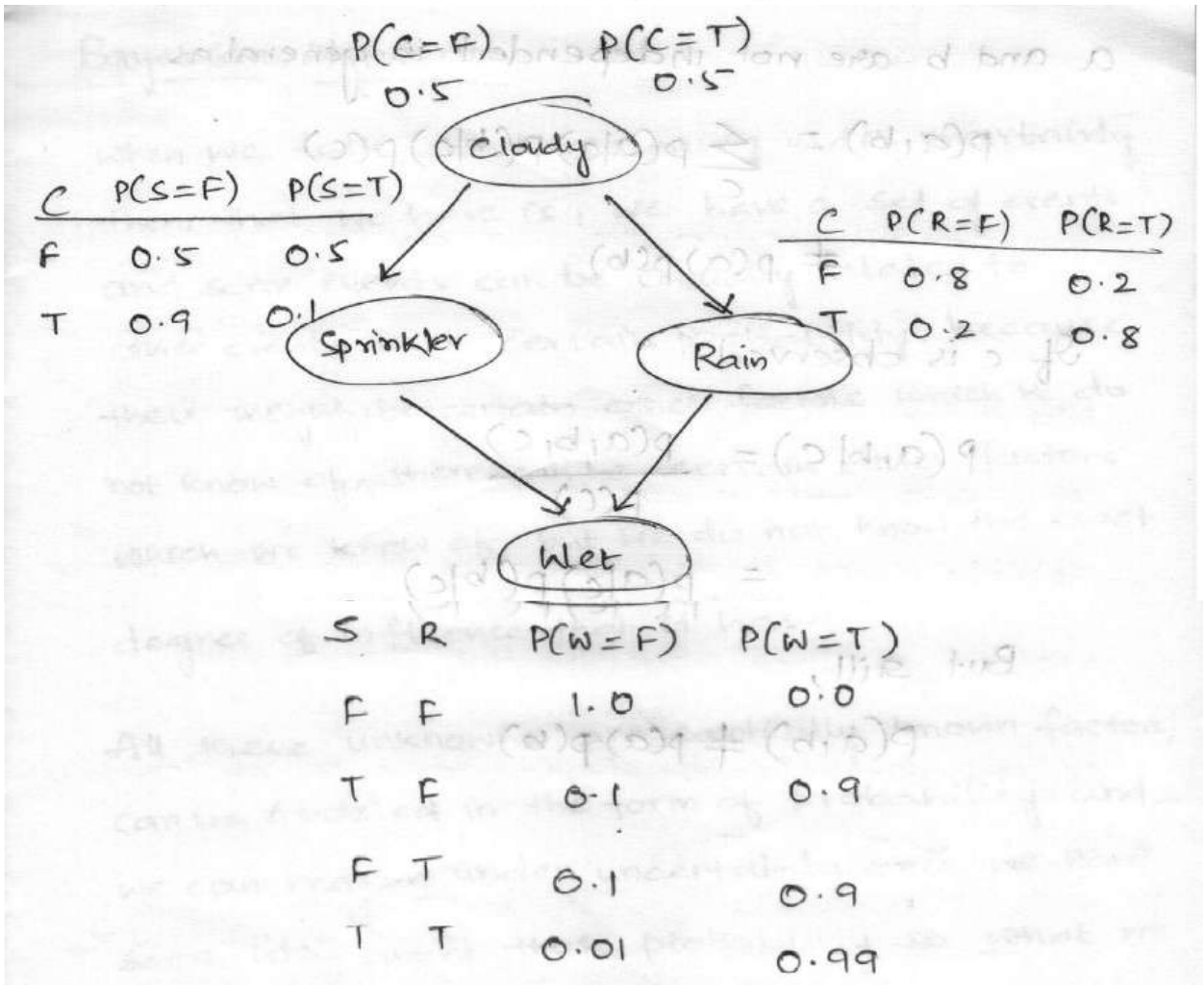


Conditional independence relationships allow us to represent the joint more compactly.

$$P(C, S, R, W) = P(C)\, P(S|C)\, P(R|C)\, P(W|S, R)$$

Attribute	Probability
p(Wet\Sprinkler, Rain)	0.95
P(Wet\Sprinkler, ¬Rain)	0.9
p(Wet\¬Sprinkler, Rain)	0.8
p(Wet\¬Sprinkler, ¬Rain)	0.1
p(Sprinkler\RainySeason)	0.0
p(Sprinkler\¬RainySeason)	1.0
p(Rain \RainySeason)	0.9
p(Rain \ ¬RainySeason)	0.1
p(RainySeason)	0.5

Conditional Probabilities for a Bayesian Network



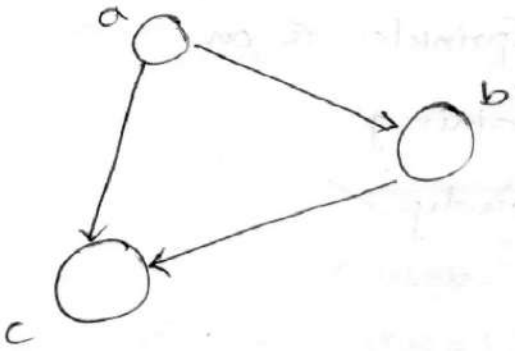
Conditional Probability Table

Each node in Bayesian Network has an associated Conditional Probability table (CPT). This gives the probability values for the random variable at the node conditioned on values for its parents.

$$P(W = T|S = T, R = F) = 0.9 \Rightarrow P(W = F|S = T, R = F) = 1 - 0.9 = 0.1$$

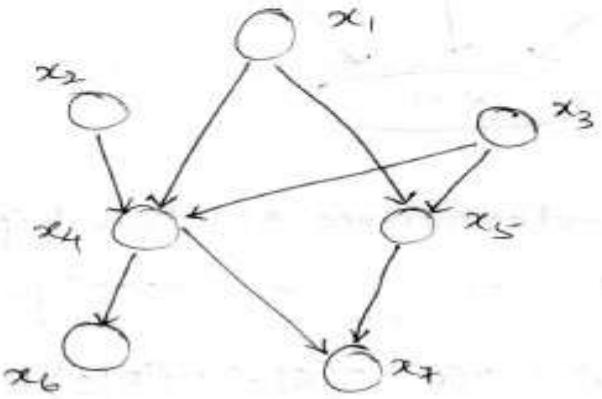
Since each row must sum to one. Since the C node has no parents, its CPT specifies the prior probability that is cloudy (in this case, 0.5).

$$P(a,b,c) = P(c|a,b)\,P(a,b)$$
$$= P(c|a,b)\,P(b|a)\,P(a)$$



Conditional Independence : x_7 is only dependent on x_4 and x_5.

$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = P(x_1)\,P(x_2)\,P(x_3)$$
$$P(x_4|x_1, x_2, x_3)\,P(x_5|x_1, x_3)\,P(x_6|x_4)$$
$$P(x_7|x_4, x_5)$$



Dempster-Shafer Theory

So far we considered individual propositions and assign each of them a point of degree of belief that is warranted for given evidence. The Dempster Shafer theory approach considers sets of propositions and assigns each of them an interval

{ , }

in which the degree of belief must lie.

Belief measures the strength of evidence in favor of the set of propositions. It ranges from 0 to 1 where 0 indicates no evidence and 1 denoting certainty.

Plausability (PL) is defined as

( ) = 1 − ( )

It also ranges from 0 to 1 and measures the extent to which evidence in favour of ¬S leaves room for belief in S.

The confidence interval is then defined as [*B*(*E*),*PL*(*E*)]
where

$$B(E) = \sum_{A} M$$

$A \subseteq E$

where *i.e.* all the evidence that makes us believe in the correctness of *P*, and

$$PL(E) = 1 - B(\neg E) = 1 - \sum_{\neg A} M$$

where *i.e.* all the evidence that contradicts *P*.

$\neg A \subseteq \neg E$

Set up a confidence interval – an interval of probabilities within which the true probability lies with a certain confidence based on belief B and plausibility PL provided by some evidence E for a proposition P.

Suppose we are given two belief statements M1 ^ M2. Let S be the subset of Θ which M1 assigns a

non-zero value & let y be corresponding set to M2. We define the combination M3 of M1 & M2.

$$m_3(Z) = \frac{\sum_{X \cap Y = Z} m_1(X) \cdot m_2(Y)}{1 - \sum_{X \cap Y = \phi} m_1(X) \cdot m_2(Y)}$$

$fever = \{flu, cold, pneu\}$
$\Theta = 1.0$ (total probability)

$\{Flu, Cold, Pneu\}$	(0.6)	**m1**
$\{\Theta\}$	(0.4)	

Suppose m_2 corresponds to our belief after observing a runny nose:

$\{All, Flu, Cold\}$	(0.8)
Θ	(0.2)

Consider the same propositions again, and assume our task is to identify the patient's disease using Dempster-Shafer theory.

(a) What is Θ?

(b) Define a set of m functions that describe the dependencies among sources of evidence and elements of Θ.

(c) Suppose we have observed spots, fever, and a tick bite. In that case, what is our $Bel(\{RockyMountainSpottedFever\})$?

- $\Theta = \{M, A, R\}$:

  M: measles
  A: allergy
  R: RMSF

- In all of these, remember that we are assuming that Θ contains the universe of possibilities, so our patient must have one of the three diseases. Also, notice that these numbers aren't the same as the numbers in problem 4. There the numbers described probabilities of symptoms given diseases. Here we're talking about likelihoods of diseases given symptoms.

m_1 (belief given just spots):

$\{M, A, R\}$	(0.9)
$\{\Theta\}$	(0.1)

m_2 (belief given just fever):

$\{M, R\}$	(0.9)
$\{\Theta\}$	(0.1)

m_3 (belief given just measles shot):

$\{A, R\}$	(0.95)
$\{\Theta\}$	(0.05)

m_4 (belief given just tick bit):

$\{R\}$	(0.95)
$\{\Theta\}$	(0.05)

Combining m_1 and m_2 to form m_5 (belief given spots and fever):

		$\{M, R\}$	(0.9)	Θ	(0.1)
$\{M, A, R\}$	(0.9)	$\{M, R\}$	(0.81)	$\{M, A, R\}$	(0.09)
Θ	(0.1)	$\{M, R\}$	(0.09)	Θ	(0.01)

The total $Bel(\{M, R\})$ at this point is .9.

Combining m_5 and m_4 (belief given spots, fever, and tick bite):

		$\{R\}$	(0.95)	Θ	(0.05)
$\{M, R\}$	(0.9)	$\{R\}$	(0.855)	$\{M, R\}$	(0.045)
$\{M, A, R\}$	(0.09)	$\{R\}$	(0.0855)	$\{M, A, R\}$	(0.0045)
Θ	(0.01)	$\{R\}$	(0.0095)	Θ	(0.0005)

So our overall $Bel(\{R\})$ is 0.95.

Fuzzy Logic

Fuzzy logic is an alternative for representing some kinds of uncertain knowledge. Fuzzy logic is a form of many-valued logic; it deals with reasoning that is approximate rather than fixed and exact. Compared to traditional binary sets (where variables may take on true or false values), fuzzy logic variables may have a truth value that ranges in degree between 0 and 1. Fuzzy logic has been extended to handle the concept of partial truth, where the truth value may range between completely true and completely false. Fuzzy set theory defines set membership as a possibility distribution.

Fuzzy logic is a totally different approach to representing uncertainty:

- It focuses on ambiguities in describing events rather the uncertainty about the occurrence of an event.
- Changes the definitions of set theory and logic to allow this.
- Traditional set theory defines set memberships as a boolean predicate.

Fuzzy Set Theory

- *Fuzzy* set theory defines set membership as a *possibility distribution*. The general rule for this can expressed as:

  $$f : [0,1]^n \rightarrow [0,1].$$

  where *n* some number of possibilities.

  This basically states that we can take *n* possible events and us *f* to generate as single possible outcome.

  This extends set membership since we could have varying definitions of, say, hot curries. One person might declare that only curries of Vindaloo strength or above are hot whilst another might say madras and above are hot. We could allow for these variations definition by allowing both possibilities in fuzzy definitions.
- Once set membership has been redefined we can develop *new logics* based on combining of sets *etc.* and reason effectively.

Intersection Search

8. Weak Slot and Filler Structures

Weak slot and filler structures turns out to be useful one for reasons besides the support of inheritance, though, including

Introduction

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

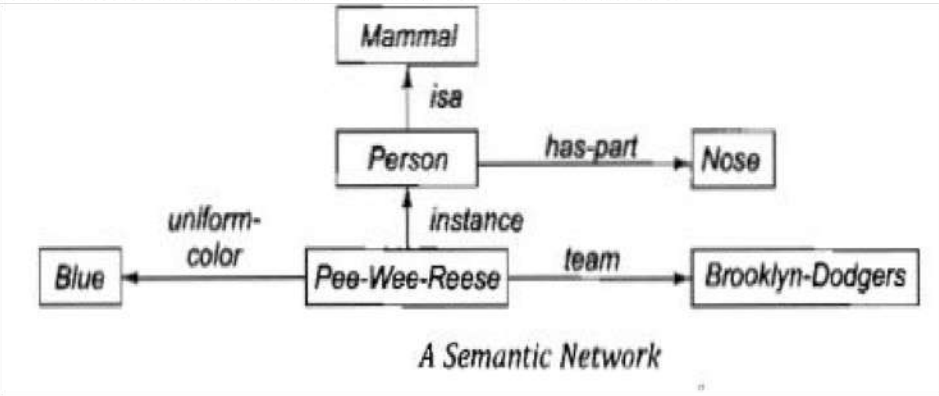
Weak slot and filler structures describe two views: **semantic nets and frames**. These talk about the representations themselves and about techniques for reasoning with them. They do not say much about the specific knowledge that the structures should contain. We call these as "knowledge poor" structures.

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

Semantic Nets

Semantic Nets were originally designed as a way to represent the meaning of English words. The main idea is that the meaning of a concept comes from the ways in which it is connected to other concepts. The information is stored by interconnecting nodes with labeled arcs. Semantic nets

initially we used to represent labeled connections between nodes.



A Semantic Network

These values can also be represented in logic as: *isa(person, mammal), instance(Pee-Wee-Reese, person) team(Pee-Wee-Reese, Brooklyn-Dodgers).* This network contains examples of both the isa and instance relations as well as some other more domain-specific relations. In this network, we could use inheritance to derive the additional relation

has-part(Pee-Wee-Reese,Nose)

The Semantic Nets can be represented in different ways by using relationships. Semantic nets have been used to represent a variety of knowledge in a variety of different programs.
One of the early ways that semantic nets were used was to find relationships among objects by

spreading activation out from each of two nodes and seeing where the activation met. This process is called ***"intersection search"***.

Using this process, it is possible to use the network to answer questions such as ***"What is the connection between the Brooklyn Dodgers and blue?"***

Representing Non-binary predicates

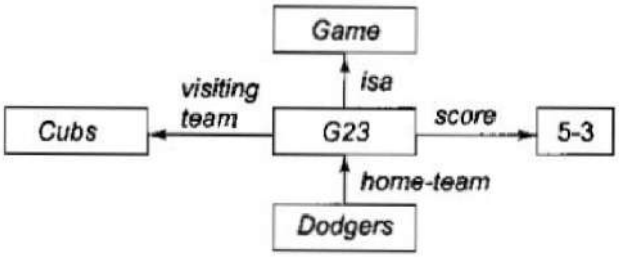
Semantic nets are natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic. Arcs from the figure could be represented in logic as

- *isa(Person, Mammal)*
- *instance(Pee-Wee-Reese, Person)*
- *team(Pee-Wee-Reese, Brooklyn-Dodgers)*
- *uniform-color(Pee-Wee-Reese, Blue)*

Many unary predicates in logic can be thought as binary predicates using isa and instance.

Example: man(marcus) → instance(Marcus,man)

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments. ***Example:*** score(Cubs, Dodgers, 5-3) can be represented in semantic net by creating a node to represent the specific game & then relating each of the three pieces of information to it.

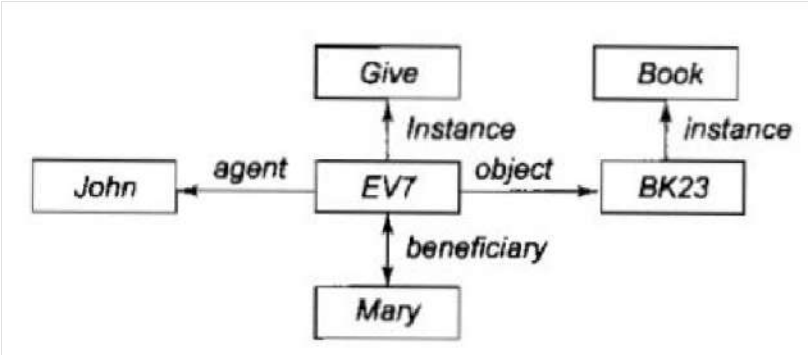


A Semantic Net for an n-Place Predicate

This technique is particularly useful for representing the contents of a typical declarative sentence

that describes several aspects of a particular event.

Example: John gave the book to Mary.



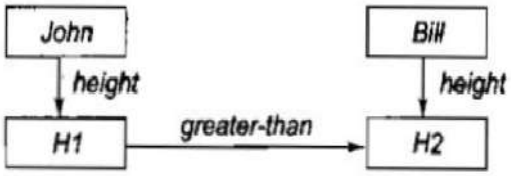
Making Some Important Distinctions

In the networks, some distinctions are glossed that are important in reasoning. For example, there should be difference between a link that defines a new entity and one that relates two existing entities.

John height is 72

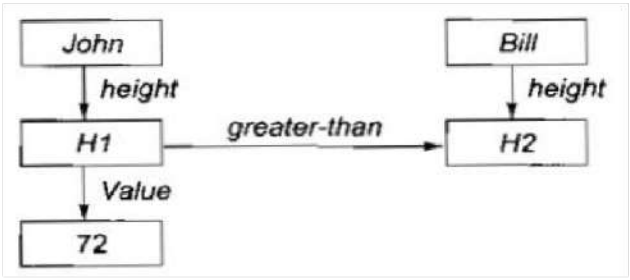
Both nodes represent objects that exist independently of their relationship to each other.

John is taller than Bill.



H1 and H2 are new concepts representing John's height and Bill's height. They are defined by their relationships to the nodes John and Bill.

John is taller than Bill and his height is 72.



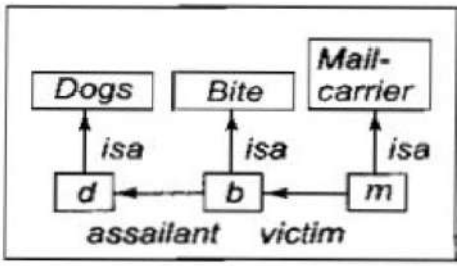
The procedures that operate on nets such as this can exploit the fact that some arcs such as define new entities, while others such as ***greater-than*** and ***value***, merely describe relationships among existing entities.

height

Partitioned Semantic Nets

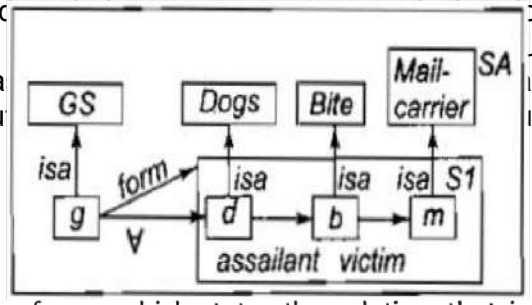
To represent simple quantified expressions in semantic nets. The way is to partition the semantic net into a hierarchical set of spaces, each of which corresponds to the scope of one or more variables.

The statement: ***The dog bit the mail carrier.***



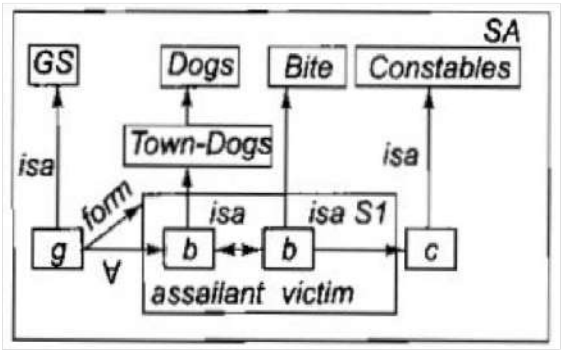
The statement: ***Every dog has bitten a mail-carrier***

The nodes Dogs, Bite, Mail-carrier represent the classes of dogs, biting and mail carriers respectively, while the noc…og, biting and a particular mailcarrier. This fact can be …ut no partitioning.
The node g stands for the a…ince of the special class GS of general statements abou…antifiers). Every element of GS



has at least two attributes: a form, which states the relation that is being asserted, and one or more ∀ connections, one for each of the universally quantified variables. In this example, for *every dog d, there exists a biting event b, and a mail-carrier m, such that d is the assailant of b and m is the victim.*

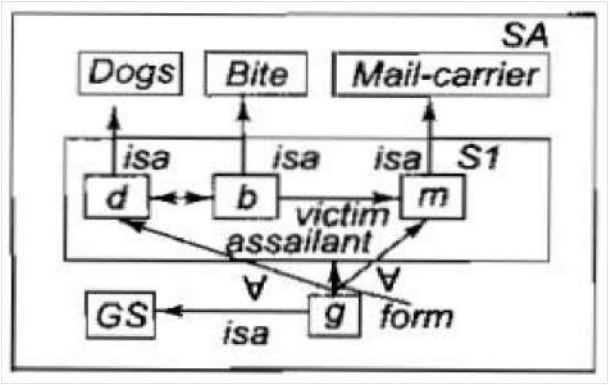
The statement: ***Every dog in town has bitten the constable***



In this net, the node c representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of d.

T

Every dog has bitten every mail carrier.



In this case, g has two ∀ links, one pointing to d, which represents any dog and one pointing to m, representing any mail carrier.
he statement.

FRAMES

Evolution into Frames

-
-
-

Frames

-

As we expand the range of problem solving tasks that the representation must support, the representation necessarily begins to become more complex.

It becomes useful to assign more structure to nodes as well as to links.

The more structure the system has, the more likely it is to be termed a frame system.

Natural language understanding requires inference i.e.,

- assumptions about what is typically true of the objects or
- Situations under consideration.

- Such information can be coded in structures known as frames.
- A *frame* is a collection of attributes or slots and associated values that describe some real world entity.
- Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning.
- Frame is a type of schema used in many AI applications including vision and natural language processing.
- Frames provide a convenient structure for representing objects that are typical to stereotypical situations. The situations to represent may be visual scenes, structure of complex physical objects, etc.
- Frames are also useful for representing commonsense knowledge. As frames allow nodes to have structures they can be regarded as three-dimensional representations of knowledge.
- A frame is similar to a record structure and corresponding to the fields and values are slots and slot fillers. Basically it is a group of slots and fillers that defines a stereotypical object.
- A single frame is not much useful.
- Frame systems usually have collection of frames connected to each other. Value of
- an attribute of one frame may be another frame. A frame for a book is given below.

Slots	Fillers
publisher	Thomson
title	Expert Systems
author	Giarratano
edition	Third
year	1998
pages	600

Frames as Sets and Instances

- Set theory provides a good basis for understanding frame systems.
-

Each frame represents:

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

Person	
isa :	*Mammal*
cardinality :	6,000,000,000
** handed :*	*Right*
Adult-Male	
isa :	*Person*
cardinality :	2,000,000,000
** height :*	5-10
ML-Baseball-Player	
isa :	*Adult-Male*
cardinality :	624
**height :*	6-1
** bats :*	equal to handed
** batting-average :*	.252
** team :*	
** uniform-color :*	
Fielder	
isa :	*ML-Baseball-Player*
cardinality :	376
**batting-average :*	.262
Pee-Wee-Reese	
instance :	*Fielder*
height :	5-10
bats :	*Right*
batting-average :	.309
team :	*Brooklyn-Dodgers*
uniform-color :	*Blue*

ML-Baseball-Team	
isa:	*Team*
cardinality :	26
** team-size :*	24
** manager :*	
Brooklyn-Dodgers	
instance :	*ML-Baseball-Team*
team-size :	24
manager :	*Leo-Durocher*
players :	*{Pee-Wee-Reese,...}*

A Simplified Frame System

- The is-a relationship is in fact the subset relation.
  - The set of adult males is a subset of the set of people.
  - The set of major league baseball players is a subset of adult males and so forth
- Theinstance relation corresponds to the relation element-of.
  - Wee-Reese is the element of Fielder.

Pee-

A

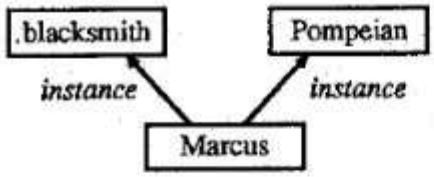
- class represents a set.
- There are 2 kinds of attributes that can be associated with it:
  - Attributes about the set itself and
  - Attributes that are to be inherited by each element of the set
  - Prefixed with *

Weak Slot Filler Structures (Continued)

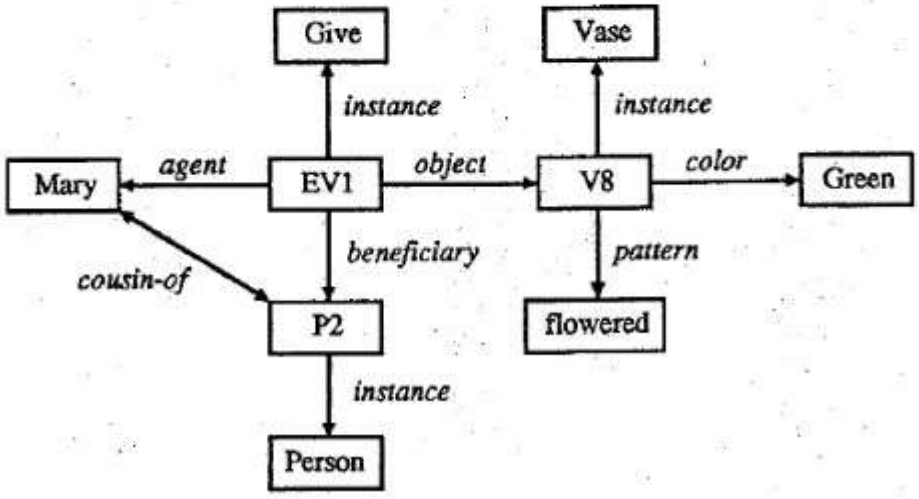
- *Slot and Filler Structures* are a device to support property inheritance along isa and instance links.
  - Knowledge in these is structured as a set of entities and their attributes.
- This structure turns out to be useful for following reasons:
  - It enables attribute values to be retrieved quickly
    - assertions are indexed by the entities
    - binary predicates are indexed by first argument. *E.g. team(Mike-Hall , Cardiff)*.
  - Properties of relations are easy to describe .
  - It allows ease of consideration as it embraces aspects of object oriented programming.
    - Modularity
    - Ease of viewing by people.

1. Construct semantic net representations for the following:
   (a) *Pompeian(Marcus), Blacksmith(Marcus)*
   (b) Mary gave the green flowered vase to her favorite cousin.

(a) Marcus:



(b) Mary:



Inheritance

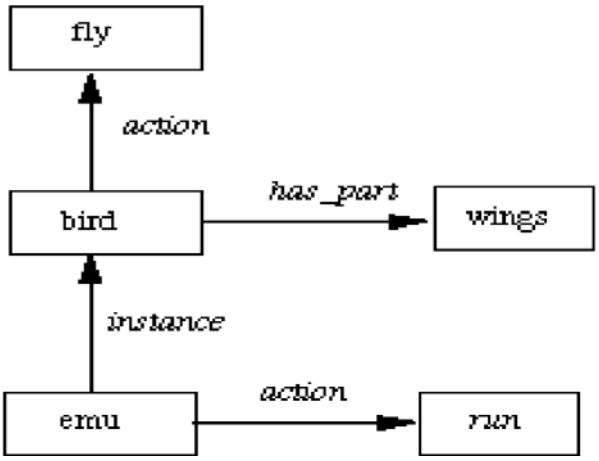
-- the *isa* and *instance* representation provide a mechanism to implement this.

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

Emus are birds.

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

in the following Semantic net:



A Semantic Network for a Default Reasoning

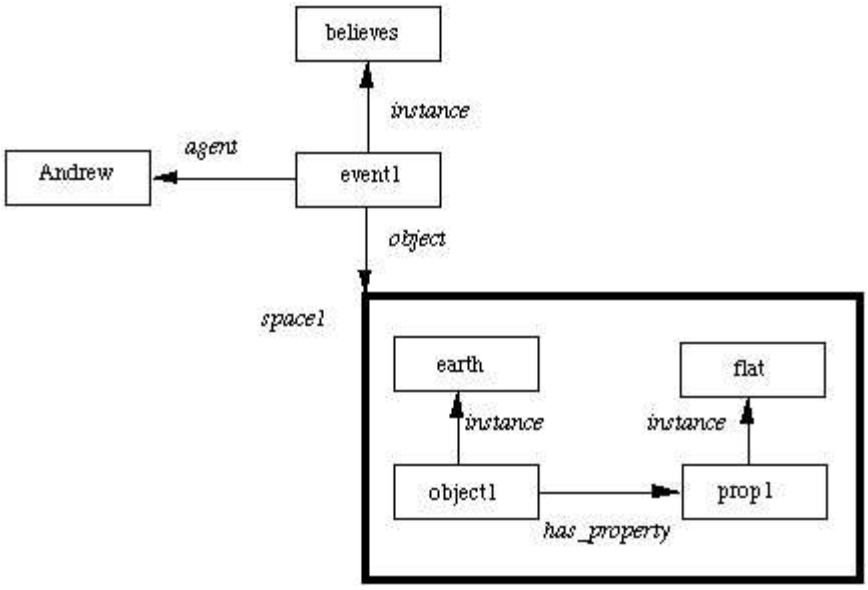
Partitioned Networks

Partitioned Semantic Networks allow for:

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

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

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



WHAT ARE FRAMES?

Natural language understanding requires inference i.e., assumptions about what is typically true of the objects or situations under consideration. Such information can be coded in structures known as frames.

NEED OF FRAMES

Frame is a type of schema used in many AI applications including vision and natural language processing. Frames provide a convenient structure for representing objects that are typical to stereotypical situations. The situations to represent may be visual scenes, structure of complex physical objects, etc. Frames are also useful for representing commonsense knowledge. As frames allow nodes to have structures they can be regarded as three-dimensional representations of knowledge.

A frame is similar to a record structure and corresponding to the fields and values are slots and slot fillers. Basically it is a group of slots and fillers that defines a stereotypical object. A single frame is not much useful. Frame systems usually have collection of frames connected to each other. Value of an attribute of one frame may be another frame. A frame for a book is given below.

Slots	Fillers
publisher	Thomson
title	Expert Systems
author	Giarratano

edition	Third
year	1998
pages	600

The above example is simple one but most of the frames are complex. Moreover with filler slots and inheritance provided by frames powerful knowledge representation systems can be built.

Frames can represent either generic or frame. Following is the example for generic frame.

Slot	Fillers
name	computer
specialization_of	a_kind_of machine
types	(desktop, laptop,mainframe,super) if-added: Procedure ADD_COMPUTER
speed	default: faster if-needed: Procedure FIND_SPEED
location	(home,office,mobile)
under_warranty	(yes, no)

The fillers may values such as computer in the name slot or a range of values as in type's slot. The procedures attached to the slots are called procedural attachments. There are mainly three types of procedural attachments: if-needed, default and if-added. As the name implies if-needed types of procedures will be executed when a filler value is needed. Default value is taken if no other value exists. Defaults are used to represent *commonsense knowledge*. Commonsense is generally used when no more situation specific knowledge is available.

The if-added type is required if any value is to be added to a slot. In the above example, if a new type of computer is invented ADD_COMPUTER procedure should be executed to add that information. An if-removed type is used to remove a value from the slot.

Frame Knowledge Representation

Consider the example first discussed in Semantics Nets :

Person

isa: Mammal Cardinality:

Adult-Male

isa: Person

Cardinality:

Rugby-Player

isa: Adult-Male Cardinality:

Height:
Weight:
Position:
Team:
Team-Colours:

Back

isa: Rugby-Player Cardinality:

Tries:

Mike-Hall

instance: Back Height: 6-0
Position: Centre
Team: Cardiff-RFC
Team-Colours: Black/Blue

Rugby-Team

isa: Team Cardinality: Team-size: 15
Coach:

Cardiff-RFC Rugby-Team
instance:
Team-size: 15
Coach: Terry Holmes
Players: {Robert-Howley, Gwyn-Jones, ... }

Figure: A simple frame system

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

Note

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

DISTINCTION BETWEEN SETS AND INSTANCES

It is important that this distinction is clearly understood.

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

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

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

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

BUT There is a problem here:

- A class is a set and its elements have properties.
- We wish to use inheritance to bestow values on its members.
- But there are properties that the set or class itself has such as the manager of a team. This is why we need to view *Cardiff-RFC* as a subset of one class players and an instance of teams. We seem to have a CATCH 22.

Solution: MetaClasses

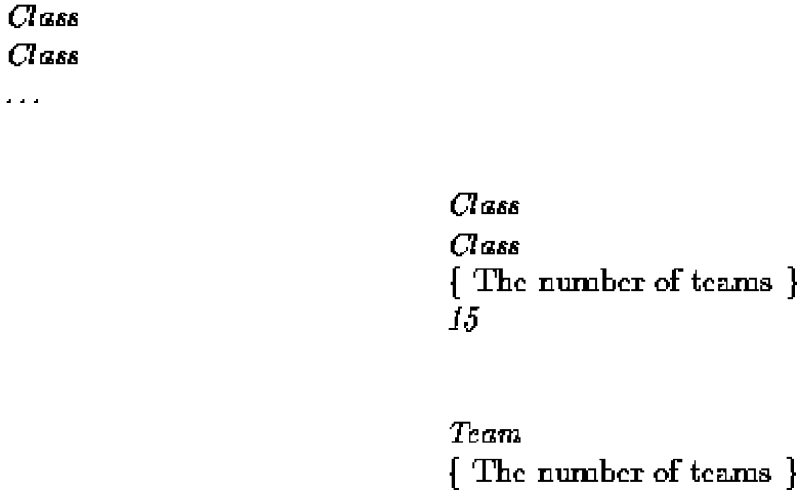
A metaclass is a special class whose elements are themselves classes.

Now consider our rugby teams as:

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

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

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



Rugby-Team
15
Terry Holmes
Class *Back*
instance: *6-0*
isa: *Scrum Half*
Cardinality: *Cardiff-RFC*
Black/Blue
Team
instance:
isa:
Cardinality:
Team-Size:

Rugby-Team
isa:
Cardinality:
Team-size: 15
Coach:

Cardiff-RFC
instance:
Team-size:
Coach:

Robert-Howley
instance:
Height:
Position:
Team:
Team-Colours:

Figure: A Metaclass frame system

Slots as Objects

How can we to represent the following properties in frames?

- Attributes such as *weight, age* be attached and make sense.
- Constraints on values such as *age* being less than a hundred
- Default values
- Rules for inheritance of values such as children inheriting parent's names Rules for computing values Many values for a slot.

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

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

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

Consider the following:

SLOT

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

Coach

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

Colour

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

Team-Colours

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

Position

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

NOTE the following:

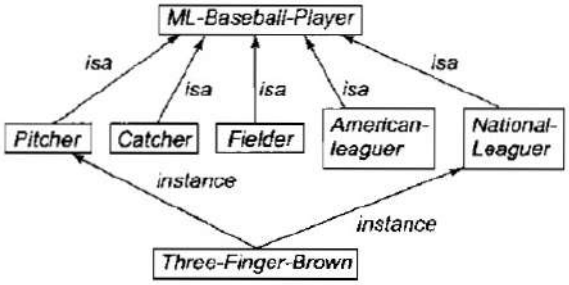
- Instances of *SLOT* are slots
- Associated with *SLOT* are attributes that each instance will inherit.

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

Slots as Full-Fledged Objects

So far, we have provided a way to describe sets of objects and individual objects, both in terms of attributes and values. Thus we have made extensive use of attributes, which we have represented as slots attached to frames. But it turns out that there are several reasons why we would like to be able to represent attributes explicitly and describe their properties. Some of the properties we would like to be able to represent and use in reasoning include:

- The classes to which the attribute can be attached, i.e. for what classes does it make sense? For example, weight makes sense for physical objects but not for conceptual ones (except in some metaphorical sense).
- Constraints on either the type or the value of the attribute. For example, the age of a person must be a numeric quantity measured in some time frame, and it must be less than the ages of the person's biological parents.
- A value that all instances of a class must have by the definition of the class.
- A default value for the attribute.
- Rules for inheriting values for the attribute. The usual rule is to inherit down *isa* and *instance* links. But some attributes inherit in other ways. For example, *last-name* inherits down the *child-of* link.
- Rules for computing a value separately from inheritance. One extreme form of such a rule is a procedure written in some procedural programming language such as LISP.
- An inverse attribute.
- Whether the slot is single-valued or multivalued.



ML-Baseball-Player
is-covered-by : {*Pitcher, Catcher, Fielder*}
{*American-Leaguer, National-Leaguer*}

Pitcher
isa : *ML-Baseball-Player*
mutually-disjoint-with : {*Catcher, Fielder*}

Catcher
isa : *ML-Baseball-Player*
mutually-disjoint-with: {*Pitcher, Fielder*}

Fielder
isa : *ML-Baseball-Player*
mutually-disjoint-with : {*Pitcher, Catcher*}

American-Leaguer
isa : *ML-Baseball-Player*
mutually-disjoint-with : {*National-Leaguer*}

National-Leaguer
isa : *ML-Baseball-Player*
mutually-disjoint-with : {*American-Leaguer*}

Three-Finger-Brown
instance : *Pitcher*
instance : *National-Leaguer*

9. Strong Slot and Filler Structures

Introduction

Individual semantic networks and frame systems may have specialized links and inference procedures, but no hard and fast rules about what kinds of objects and links are good in general for knowledge representation.

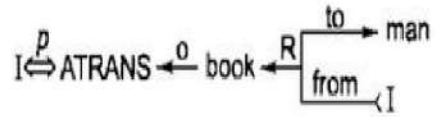
3 Structures: *Conceptual Dependency, Scripts and CYC* embody specific notions of what types of objects and relations are permitted. They stand for powerful theories of how AI programs can represent and use knowledge about common situations.

CONCEPTUAL DEPENDENCY (CD)

It is the theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences. The goal is to represent the knowledge in a way that Facilitates drawing

- inferences from the sentences.
- Is independent of the language in which the sentences are originally stated. CD provides a structure into which nodes representing information can be placed a specific set of primitives at a given level of granularity.

Representation of Conceptual Dependency: ***I gave the man a book***



where the symbols have the following meanings:

- Arrows indicate direction of dependency.
- Double arrow indicates two way link between actor and action.
- p indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- o indicates the object case relation.
- R indicates the recipient case relation.

In CD representation of actions are built from a set of primitive Acts.

ATRANS	Transfer of an abstract relationship (e.g., give)
PTRANS	Transfer of the physical location of an object (e.g., go)
PROPEL	Application of physical force to an object (e.g., push)
MOVE	Movement of a body part by its owner (e.g., kick)
GRASP	Grasping of an object by an actor (e.g., clutch)
INGEST	Ingestion of an object by an animal (e.g., eat)
EXPEL	Expulsion of something from the body of an animal (e.g., cry)
MTRANS	Transfer of mental information (e.g., tell)
MBUILD	Building new information out of old (e.g., decide)
SPEAK	Production of sounds (e.g., say)
ATTEND	Focusing of a sense organ toward a. stimulus (e.g., listen)

A second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are 4 primitive conceptual categories from which dependency structures can be built.

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
PAs	Modifiers of PPs (picture aiders)

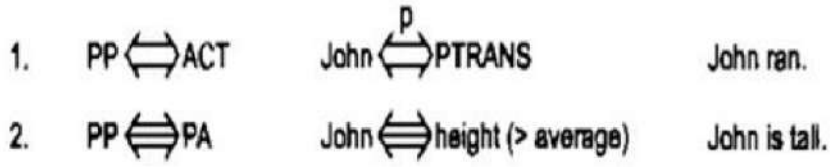
In addition, dependency structures are themselves conceptualizations and can serve as components of larger dependency structures.

The dependencies among conceptualizations correspond to semantic relations among the underlying concepts. The **first column** contains the rules; the **second** contains examples of use and the **third** theircontains an English version of each example. The rules are interpreted as follows:

- **Rule 1**

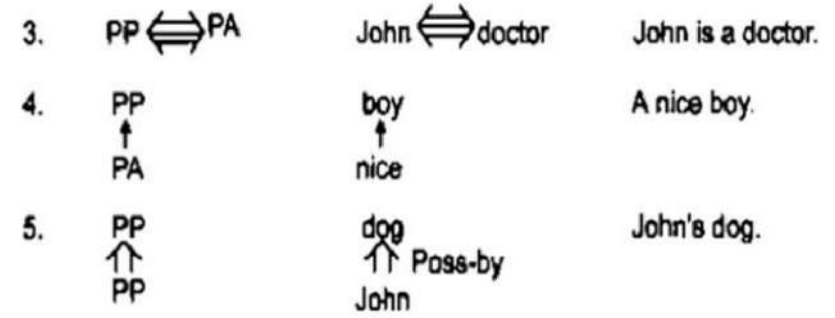
describes the relationship between an actor and the event he or she causes. This is a two-way dependency since neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.

- **Rule 2** describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.



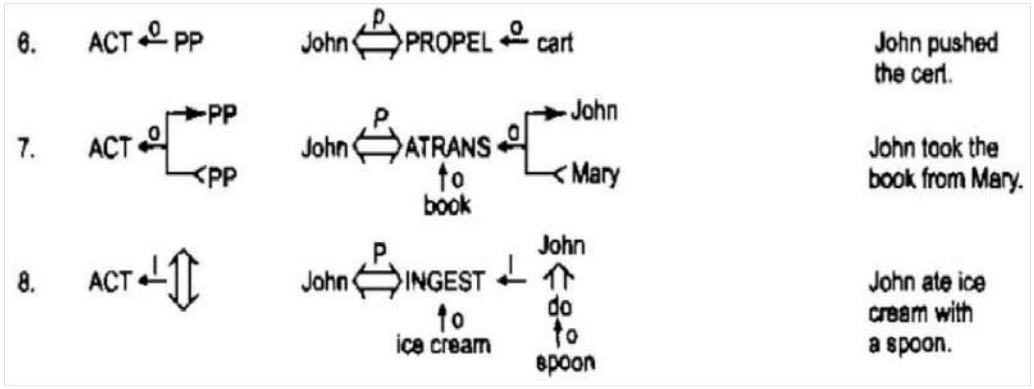
- **Rule 3** describes the relationship between two PPs, one of which belongs to the set defined by the other.
- **Rule 4** describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.
- **Rule 5** describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are ○ possession (shown as POSS-BY), ○ location (shown as LOC) and ○ physical containment (shown as CONT).

  The direction of the arrow is again toward the concept being described.

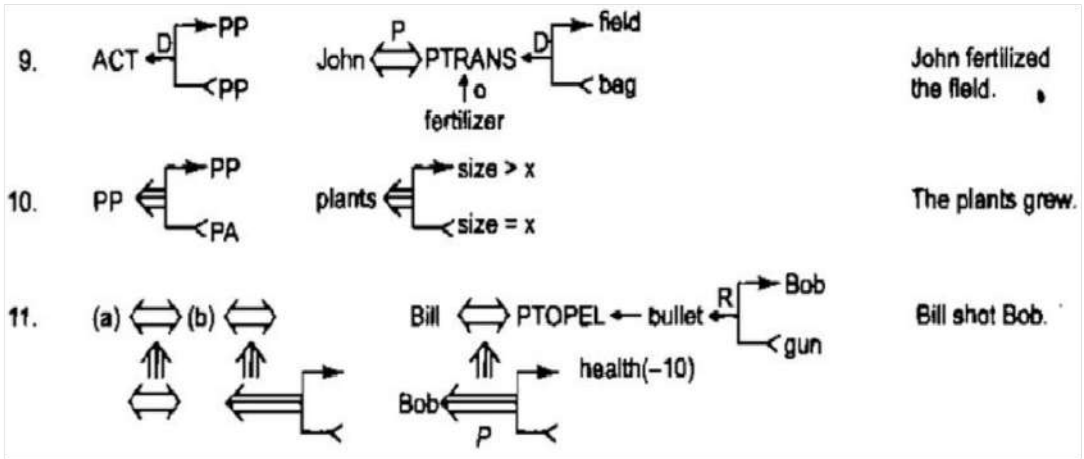


- **Rule 6** describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.
- **Rule 7** describes the relationship between an ACT and the source and the recipient of the ACT.
- **Rule 8** describes the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualization (i.e., it must contain

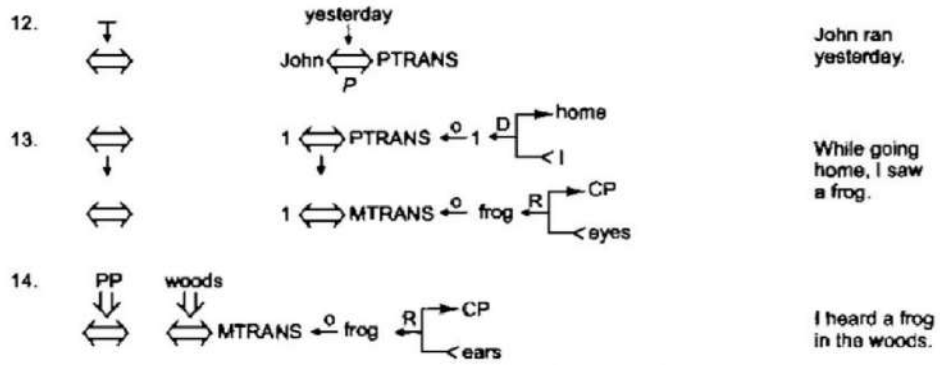
an ACT), not just a single physical object.



- **Rule 9** describes the relationship between an ACT and its physical source and destination.
- **Rule 10** represents the relationship between a PP and a state in which it started and another in which it ended.
- **Rule 11** describes the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.



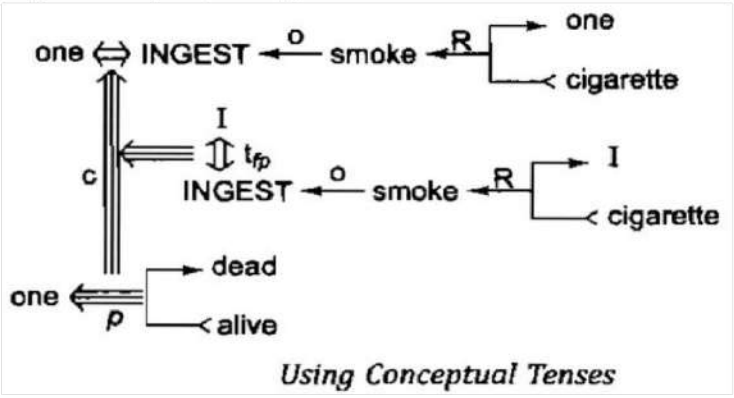
- **Rule 12** describes the relationship between a conceptualization and the time at which the event it describes occurred.
- **Rule 13** describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system.
- **Rule 14** describes relationship between a conceptualization and the place at which it occurred.



Conceptualizations representing events can be modified in a variety of ways to supply the information normally indicated in language by the tense, mood or aspect of a verb form. The set of conceptual tenses includes

p	Past
f	Future
t	Transition
t_s	Start transition
t_f	Finished transition
k	Continuing
?	Interrogative
/	Negative
nil	Present
delta	Timeless
c	Conditional

Example: ***Since smoking can kill you, I stopped.***



Using Conceptual Tenses

Advantages of CD:

- Using these primitives involves fewer inference rules.
- Many inference rules are already represented in CD structure.
- The holes in the initial structure help to focus on the points still to be established.

Disadvantages of CD:

- Knowledge must be decomposed into fairly low level primitives.
- Impossible or difficult to find correct set of primitives.
- A lot of inference may still be required.

- Representations can be complex even for relatively simple actions. Consider:
  Dave bet Frank five pounds that Wales would win the Rugby World Cup.
  Complex representations require a lot of storage

Applications of CD:

- **MARGIE** (*Meaning Analysis, Response Generation and Inference on English*) -- model natural language understanding.
- **SAM** (*Script Applier Mechanism*) -- Scripts to understand stories. See next section.
- **PAM** (*Plan Applier Mechanism*) -- Scripts to understand stories.

SCRIPTS

We present a mechanism for representing knowledge about common sequences of events.
A **script** is a structure that describes stereotyped sequence of events in a particular event. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available.

A script is a structure that prescribes a set of circumstances which could be expected to follow on from one another. It is similar to a thought sequence or a chain of situations which could be anticipated. It could be considered to consist of a number of slots or frames but with more specialized roles.

Scripts provide an ability for default reasoning when information is not available that directly states that an action occurred. So we may assume, unless otherwise stated, that a diner at a restaurant was served food, that the diner paid for the food, and that the dinner was served by a waiter/waitress.

Scripts are beneficial because:

- Events tend to occur in known runs or patterns.
- Causal relationships between events exist.
- Entry conditions exist which allow an event to take place
- Prerequisites exist upon events taking place. E.g. when a student progresses through a degree scheme or when a purchaser buys a house.

The important components of the script are:

Entry Conditions	these must be satisfied before events in the script can occur.
Results	Conditions that will be true after events in script occur.
Props	objects involved in events.
Roles	events.
Track	Variations on the script. Different tracks may share components of the same script.
Scenes	The sequence of events that occur. Events are represented in conceptual dependency form.

s representing
s involved in the

Scripts are useful in describing certain situations such as robbing a bank. This might involve:

- Getting a gun.
- Hold up a bank.
- Escape with the money.

- Here the ***Props*** might be
  - Gun, *G*.
  - Loot, *L*.
  - Bag, *B*
  - Getaway car, *C*.
- The ***Roles*** might be:
  - Robber, *S*.
  - Cashier, *M*.
  - Bank Manager, *O*.
  - Policeman, *P*.
- The ***Entry*** *Conditions* might be:
  - *S* is poor.
  - *S* is destitute.
- The ***Results*** might be:
  - *S* has more money.
  - *O* is angry.
  - *M* is in a state of shock.
  - *P* is shot.

Script: ROBBERY — *Track: Successful Snatch*

Props:
G = Gun,
L = Loot,
B= Bag,
C = Get away car.

Roles:
R = Robber,
M = Cashier,
O = Bank Manager,
P = Policeman.

Entry Conditions:
R is poor.
R is destitute.

Results:
R has more money.
O is angry.
M is in a state of shock.
P is shot.

Scene 1: Getting a gun

R PTRANS R into Gun Shop
R MBUILD R choice of G
R MTRANS choice.
R ATRANS buys G

(go to scene 2)

Scene 2 Holding up the bank

R PTRANS R into bank
R ATTEND eyes M, O and P
R MOVE R to M position
R GRASP G
R MOVE G to point to M
R MTRANS "Give me the money or ELSE" to M
P MTRANS "Hold it Hands Up" to R
R PROPEL shoots G
P INGEST bullet from G
M ATRANS L to M
M ATRANS L puts in bag, B
M PTRANS exit
O ATRANS raises the alarm

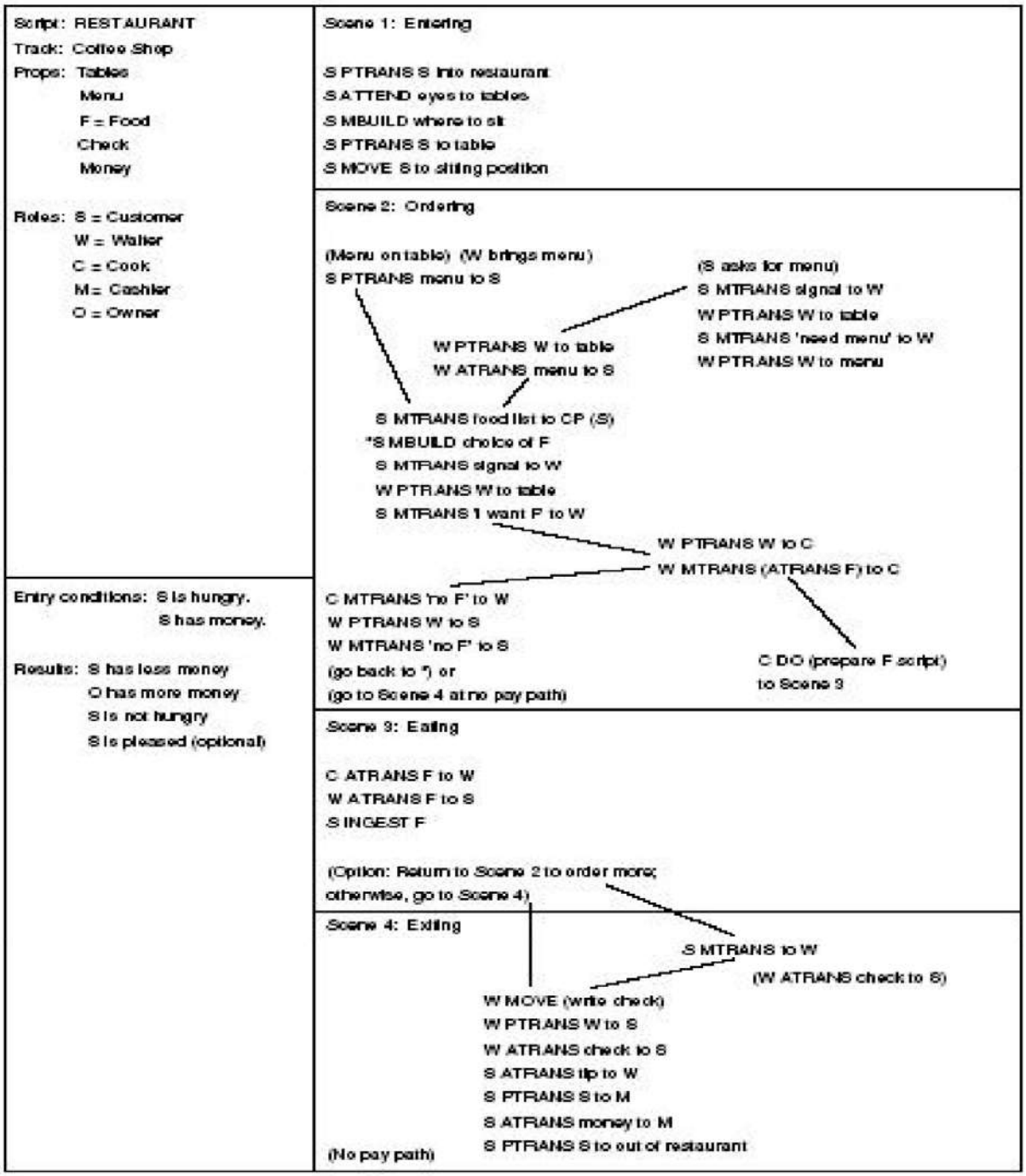
(go to scene 3)

Scene 3: The getaway

M PTRANS C

Restaurant Script

The script does not contain typical actions although there are options such as whether the customer was pleased or not. There are multiple paths through the scenes to make for a robust script what would a "going to the movies" script look like? Would it have similar props, actors, scenes? How about "going to class"?



Advantages of Scripts:

- Ability to predict events.
- A single coherent interpretation may be build up from a collection of observations.

Disadvantages:

- Less general than frames.
- May not be suitable to represent all kinds of knowledge.

CYC

CYC is a very large knowledge base project aimed at capturing human commonsense knowledge. The goal of CYC is to encode the large body of knowledge that is so obvious that it is easy to forget to state it explicitly. Such a knowledge base could then be combined with specialized knowledge bases to produce systems that are less brittle than most of the ones available today.

Why build large knowledge bases?

- **Brittleness**-- Specialized knowledge bases are *brittle*. Hard to encode new situations and non-graceful degradation in performance. Commonsense based knowledge bases should have a firmer foundation.
- **Form and Content**- Knowledge representation may not be suitable for AI. Commonsense strategies could point out where difficulties in content may affect the form.
- **Shared Knowledge** -- Should allow greater communication among systems with common bases and assumptions.

Building an immense knowledge base is a staggering task. There are two possibilities for acquiring this knowledge automatically:

1. **Machine Learning:** In order for a system to learn a great deal, it must already know a great deal. In particular, systems with a lot of knowledge will be able to employ powerful analogical reasoning.
2. **Natural Language Understanding:** Humans extend their own knowledge by reading books and talking with other humans.

How is CYC coded?

- By hand.
- Special CYCL language:
  - LISP like.
  - Frame based
  - Multiple inheritance
  - Slots are fully fledged objects.
  - Generalized inheritance -- any link not just *isa* and *instance*.

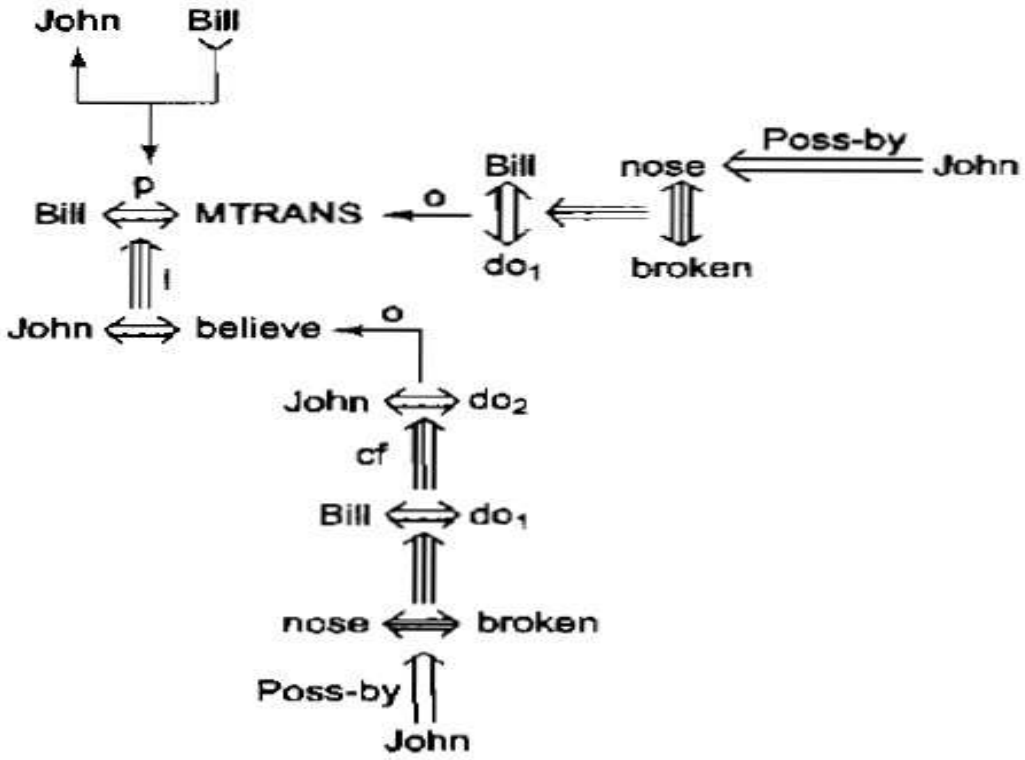
How do we do this?

- We require special implicit knowledge or commonsense such as:
  - We only die once.
  - You stay dead.
  - You cannot learn of anything when dead.
  - Time cannot go backwards.

Strong Slot and Filler Structures (Continued)

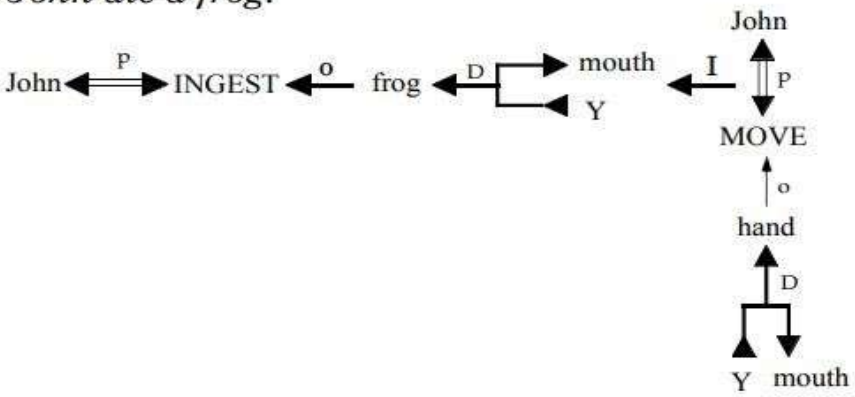
Conceptual Dependency Examples:

Consider the sentence, “Bill threatened John with a broken Nose”.

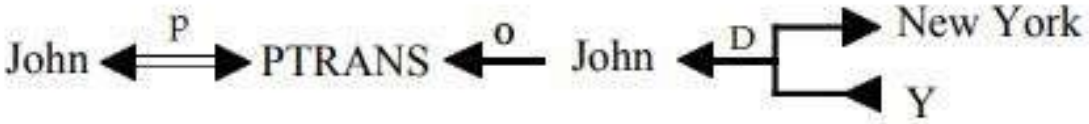


The CD representation of the information contained in the sentence is shown above. It says that Bill informed John that he (Bill) will do something to break John’s nose. Bill did this so that John will believe that if he (John) does something (different from what Bill will do to break his nose), then Bill will break John’s nose. In this representation, the word believe has been used to simplify the example. But the idea behind believe can be represented in CD as MTRANS of a fact into John’s memory. The actions do1 and do2 are dummy placeholders that refer to some as yet unspecified actions.

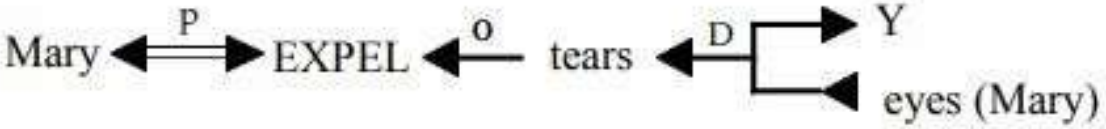
John ate a frog.



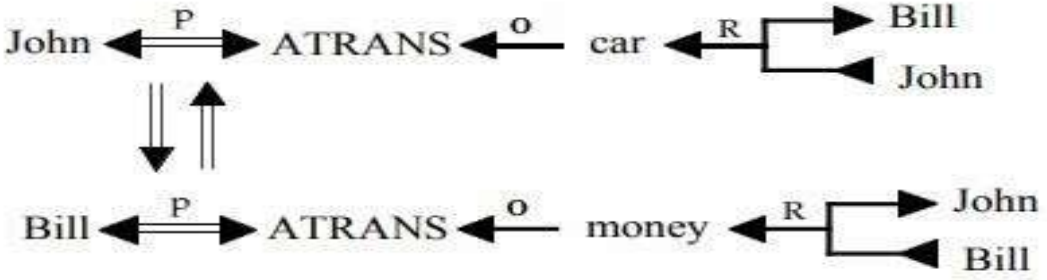
John went to New York.



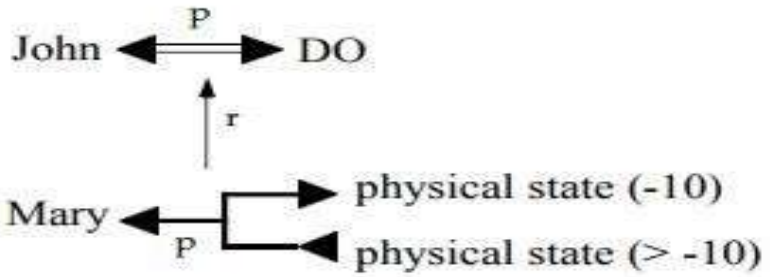
Mary cried.



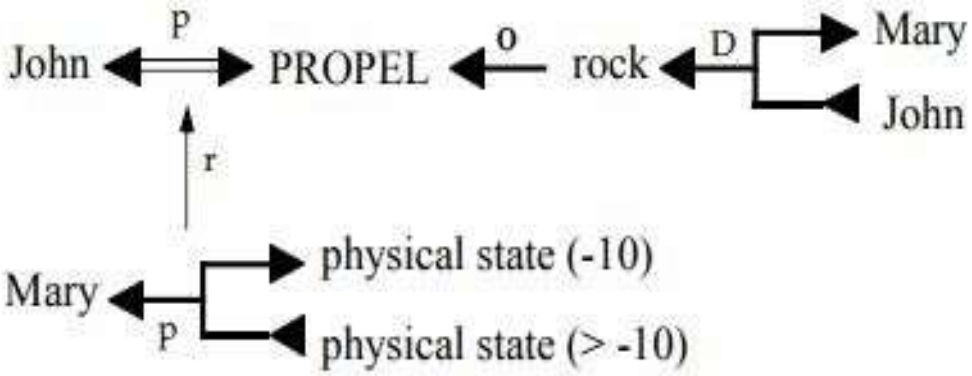
John sold his car to Bill.



John killed Mary.



John killed Mary by throwing a rock at her.



CYCL

- CYCs knowledge is encoded in a representation language called CYCL.
- CYCL is a frame based system that incorporates most of the techniques.
- Generalizes the notion of inheritance so that properties can be inherited along any link, not just isa and instance.
- CYCL contains a constraint language that allows the expression of arbitrary first-order logical expressions.

```
Mary
  likes:                    ???
  constraints:              (LispConstraint)
LispConstraint
  slotConstrained:          (likes)
  slotValueSubsumes:
    (TheSetOf X (Person allInstances)
       (And (programsIn X LispLanguage)
          (Not (ThereExists Y (Languages allInstances)
             (And (Not (Equal Y LispLanguage))
                (programsIn X Y))))))
  propagationDirection: forward
Bob
  programsIn:               (LispLanguage)
Jane
  programsIn:               (LispLanguage CLanguage)
```

Frames and Constraint Expressions in CYC

11. Learning

What is Learning?

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

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

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

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

Why is it hard?

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

How can we learn?

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

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

Rote Learning

Rote Learning is basically *memorisation*.

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

A simple example of rote learning is *caching*

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

Memorisation is a key necessity for learning:

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

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

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

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

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

Store v Compute

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

How can we decide?

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

Learning by Taking Advice

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

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

There are two basic approaches to advice taking:

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

Automated Advice Taking

The following steps summarise this method:

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

The steps can be iterated.

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

Many have concentrated on:

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

EMYCIN is an example of such a system.

Example Learning System - FOO

Learning the game of hearts

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

Hearts:

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

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

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

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

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

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

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

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

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

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

(exists c2 (point-cards)

(during (take (trick-winner) c1)

(take me c2))))))

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

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

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

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

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

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

(possible (trick-has-points)))

(low(card-of me)))

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

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

FOO has 2 basic shortcomings:

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

Learning by Problem Solving

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

Learning by Parameter Adjustment

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

However, many programs have a static evaluation function.

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

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

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

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

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

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

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

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

Learning by Macro Operators

The basic idea here is similar to Rote Learning:

Avoid expensive recomputation

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

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

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

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

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

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

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

MACROP can now be used in future operation.

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

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

Learning by Chunking

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

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

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

Basic Outline of SOAR's Method

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

Inductive Learning

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

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

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

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

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

A Blocks World Learning Example -- Winston (1975)

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

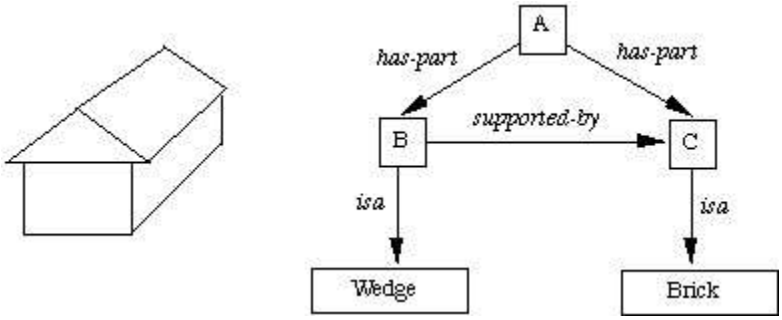


Fig. 27 House object and semantic net

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

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

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

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

Version Spaces

Structural concept learning systems are not without their problems.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Candidate Elimination Algorithm

Let us now formally describe the algorithm.

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

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

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

Algorithm:

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

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

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

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

Let the first example be positive:

Then we set

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

Then set

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

No the second example is negative:

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

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

S is unaffected. $\{5\clubsuit, 6\clubsuit, 9\clubsuit, 10\clubsuit, Q\clubsuit\}$

Our third example is positive:

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

$$
\begin{aligned}
G \quad = \quad & \{(x_1, x_2 = \clubsuit, x_3, x_4, x_5), \\
& (x_1, x_2, x_3 = \clubsuit, x_4, x_5), \\
& (x_1, x_2, x_3, x_4 = \clubsuit, x_5), \\
& (x_1, x_2, x_3, x_4, x_5 = \clubsuit)\} \\
S \quad = \quad & \{(x_1 = 5\clubsuit, x_2 = \clubsuit, x_3 = \clubsuit, x_4 = \clubsuit, x_5 = \clubsuit)\} \\
& \{A\heartsuit, 6\heartsuit, 9\heartsuit, 10\heartsuit, Q\heartsuit\}
\end{aligned}
$$

Our fourth example is also positive:

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

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

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

Decision Trees

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

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

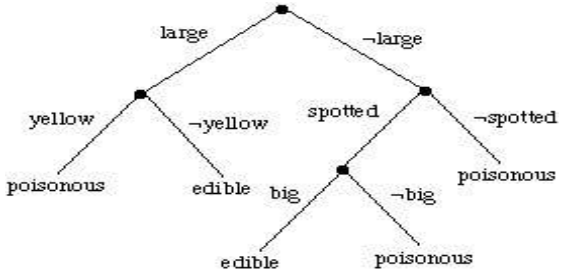


Fig. 28 *Edible Mushroom* decision tree

Building decision trees

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

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

Adding new nodes

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

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

Decision tree advantages:

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

Disadvantages:

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

Explanation Based Learning (EBL)

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

EBL example

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

The Domain Knowledge is:

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

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

Our goal can expressed as follows:

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

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

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

We can fly to Cardiff and then drive to Brecon.

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

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

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

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

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

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

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

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

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

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

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

Discovery

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

Theory Driven Discovery - AM (1976)

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

AM has 2 inputs:

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

Given the above information AM *discovered*:

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

How does AM work?

AM employs many general-purpose AI techniques:

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

Data Driven Discovery -- BACON (1981)

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

BACON is an attempt at provided such an AI system.

BACON system outline:

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

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

Analogy

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

Bill is built like a large outdoor brick lavatory.

He was like putty in her hands

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

Transformational Analogy

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

E.g. Geometry.

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

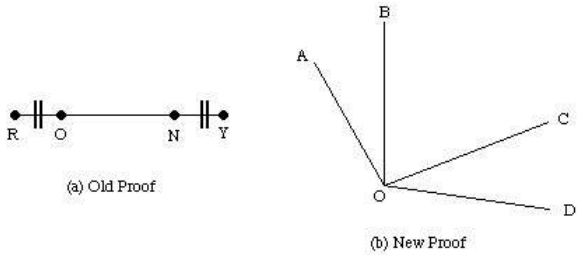


Fig. 29 Transformational Analogy Example

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

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

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

Derivational Analogy

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

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

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

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

12. Expert Systems

Expert Systems

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

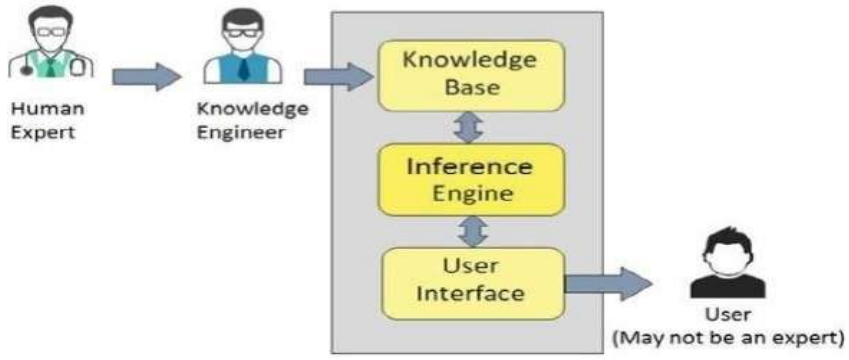
- Expert systems solve problems that are normally solved by human “experts”. To solve expert-level problems, expert systems need access to a substantial domain knowledge base, which must be built as efficiently as possible. They also need to exploit one or more reasoning mechanisms to apply their knowledge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

Capabilities of Expert Systems

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

Components of Expert Systems

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



Knowledge Base

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

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

Inference Engine

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

User Interface

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

Requirements of Efficient ES User Interface

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

Expert Systems Limitations

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

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

Applications of Expert System

The following table shows where ES can be applied.

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

Expert System Technology

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

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

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

Representing and Using Domain Knowledge

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

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

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

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

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

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

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

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

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

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

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

Reasoning with the Knowledge

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

EXPERT SYSTEM SHELLS

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

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

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

EXPLANATION

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

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

KNOWLEDGE ACQUISITION

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

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

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

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

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

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

MOLE (Knowledge Acquisition System)

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

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

1. Initial Knowledge base construction.

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

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

2. Refinement of the knowledge base.

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

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

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

SALT Program

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

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

SALT uses the following heuristics to guide the acquisition process:

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

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

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