DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: ARTIFICIAL INTELLIGENCE **SUBJECT CODE: CS T71**

PREPARED BY : Ms.S.VINITHA,AP/CSE

UNIT I

Introduction: History of AI - problem spaces and search- Heuristic Search techniques –Best-first search- Problem reduction-Constraint satisfaction-Means Ends Analysis. Intelligent agents: Agents and environment – structure of agents and its functions

INTRODUCTION

ARTIFICIAL INTELLIGENCE

Artificial Intelligence is a branch of computer science that deals with the creation of computer programs that can provide solutions, otherwise human would have to solve.

Artificial Intelligence definitions are given on the basis of

(i) Based on thought process and reasoning.

(ii) Based on the behavior.

(iii)Based on human performance.

(iv)Based on Rationality.

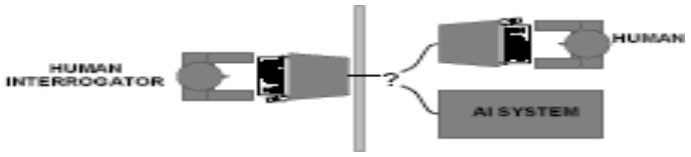
Views of AI fall into four categories:

Thinking humanly	**Thinking rationally**
"The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning…"	" The study of mental faculties through the use of computational models"

Acting humanly	**Acting rationally**
“The study of how to make computers do things at which, at the moment, people are better”	“Computational intelligence is the study of the design of intelligent agents”

Acting humanly: Turing Test

- The Turing Test, Proposed by Alan Turing (1950) ,was designed to provide a satisfactory operational definition of Intelligence ."Computing machinery and intelligence":
- "Can machines think?" → "Can machines behave intelligently?"
- Operational test for intelligent behavior: The Imitation Game



- Predicted that by 2000, a machine might have a 30% chance of fooling a person for 5 minutes
- Anticipated all major arguments against AI in following 50 years
- Suggested major components of AI: knowledge, reasoning, language understanding, learning.

Turing test

•Three rooms contain a person, a computer, and an interrogator.

•The interrogator can communicate with the other two by teleprinter.

•The interrogator tries to determine which the person is and which the machine is.

•The machine tries to fool the interrogator into believing that it is the person.

•If the machine succeeds, then we conclude that the machine can think.

The computer would need to possess the following capabilities:

- **Natural language processing** to enable it to communicate successfully in English.
- **Knowledge representation** to store what it knows or hears;
- ***Automated reasoning*** to use the stored information to answer questions and to draw new conclusions.
- **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns.
- **Computer vision** to perceive objects.
- **Robotics** to manipulate objects and move about.

Thinking humanly: cognitive modeling

- In 1960's "cognitive revolution": information-processing psychology
- Requires scientific theories of internal activities of the brain
- How to validate? Requires

  1) Predicting and testing behaviour of human subjects (top-down)

  2) Direct identification from neurological data (bottom-up)
- Both approaches (roughly, Cognitive Science and Cognitive Neuroscience) are now distinct from Artificial Intelligence

Try to understand how the mind works. How do we think?

Two possible routes to find answers:

- By introspection: We figure it out ourselves!
- By experiment: Draw upon techniques of psychology to conduct controlled experiments. (Rat in a box!)

Thinking rationally: "laws of thought"

- Aristotle: what are correct arguments/thought processes?
- Several Greek schools developed various forms of logic: notation and rules of derivation for thoughts; may or may not have proceeded to the idea of mechanization
- Direct line through mathematics and philosophy to modern AI
- Problems:
  - Not all intelligent behaviour is mediated by logical deliberation
  - What is the purpose of thinking? What thoughts should I have?
- Trying to understand how we actually think is one route to AI. But how about how we should think.
- Use logic to capture the laws of rational thought as symbols.
- Reasoning involves shifting symbols according to well-defined rules (like algebra).
- Result is idealized reasoning.

Acting rationally: rational agent

- Rational behaviour: doing the right thing
- The right thing: that which is expected to maximize goal achievement, given the available information
- Doesn't necessarily involve thinking – e.g., blinking reflex – but thinking should be in the service of rational action

Rational agents

- An agent is an entity that perceives and acts
- This course is about designing rational agents
- Abstractly, an agent is a function from percept histories to actions:

[f: P* → A]

- For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.
- Caveat: computational limitations make perfect rationality unachievable.

  → Design best program for given machine resources.

2. HISTORY OF ARTIFICIAL INTELLIGENCE

The gestation of artificial intelligence

- Pitts and McCulloch (1943): simplified mathematical model of neurons (resting/firing states) can realize all propositional logic primitives (can compute all Turing computable functions)
- Allen Turing: Turing machine and Turing test (1950)

They drew on three sources: knowledge of the basic physiology and function of neurons in the brain; a formal analysis of propositional logic. They proposed a model of artificial neurons in which each neuron is characterized as being "on" or "off," with a switch to "on" occurring in response to stimulation by a sufficient number of neighboring neurons. A simple updating rule for modifying the connection strengths between neurons. This rule, now called as Hebbian learning.

- Claude Shannon: information theory; possibility of chess playing computers
- Tracing back to Boole, Aristotle, Euclid (logics, syllogisms)

The birth of artificial intelligence (1956)

- U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at **Dartmouth in the summer of 1956.**
- The others had ideas and in some cases programs for particular applications such as checkers.
- They invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind-body problem.

Early enthusiasm, great expectations (1952-1969)

- Given the primitive computers and programming tools of the time, and the fact that only a few years earlier computers were seen as things that could do arithmetic and no more.
- **Newel and Simon's** early success was followed up with the General Problem Solver.
- Unlike Logic Theorist, this program was designed from the start to imitate human problem-solving protocols. Within the limited class of puzzles it could handle, it turned out that the order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems

A dose of reality (1966-1973)

- The first kind of difficulty arose because most early programs contained little or no knowledge of their subject matter; they succeeded by means of simple syntactic manipulations.

- The second kind of difficulty was the intractability of many of the problems that A1 was attempting to solve. Most of the early **A1** programs solved problems by trying out different combinations of steps until the solution was found.
- The illusion of unlimited computational power was not confined to problem-solving programs.
- A third difficulty arose because of some fundamental limitations on the basic structures being used to generate intelligent behavior.

Knowledge-based systems:

- A general-purpose search mechanism trying to string together elementary reasoning steps to find complete solutions. Such approaches have been called weak methods, because, although general, they do not scale up to large or difficult problem instances. The alternative to weak methods is to use more powerful, domain-specific knowledge that allows larger reasoning steps and can more easily handle.

AI becomes an industry:

- In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. In response the United States formed the **Microelectronics and Computer Technology Corporation (MCC**) as a research consortium designed to assure national competitiveness. In both cases, A1 was part of a broad effort, including chip design and human-interface research. However, the A1 components of MCC and the Fifth Generation projects never met their ambitious goals.
- Overall, the AI industry boomed from a few million dollars in **1980** to billions of dollars in **1988.**

The return of neural networks (1986-present)

- Although computer science had largely abandoned the field of neural networks in the late 1970s, work continued in other fields.
- The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection Parallel Distributed Processing.

AI becomes a science (1987-present)

- Recent years have seen a revolution in both the content and the methodology of work in artificial intelligence.

- It is now more common to build on existing theories than to propose brand new ones, to base claims on rigorous theorems or hard experimental evidence rather than on intuition, and to show relevance to real-world applications rather than toy examples.

The emergence of intelligent agents (1995-present)

- AI, researchers have also started to look at the "whole agent" problem again.
- One of the most important environments for intelligent agents is the Internet. AS systems have become so common in web-based applications that the "-bot" suffix has entered everyday language. Moreover, AS technologies underlie many Internet tools, such as search engines, recommender systems, and Web site construction systems.
- A second major consequence of the agent perspective is that A1 has been drawn into much closer contact with other fields, such as control theory and economics that also deal with agents.

3. APPLICATIONS OF AI

Autonomous planning and scheduling

1. Route planning 2. Automated scheduling of actions in spacecrafts

Game playing

- IBM's Deep Blue defeated G.Kasparov (the human world champion) (1997)
- The program FRITZ running on an *ordinary PC* drawed with V.Kramnik (the human world champion) (2002)

Autonomous control

- Automated car steering and the Mars mission.

Diagnosis

- Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.
- Literature describes a case where a leading expert was convinced by a computer diagnostic.

Logistic planning

- Defence Advanced Research Project Agency stated that this single application more than paid back DARPA's 30-year investment in AI

Robotics

- Microsurgery and RoboCup. By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.

4. INTELLIGENT AGENTS AND ITS ENVIRONMENTS

AGENTS AND ENVIRONMENTS

- **An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators**.
- **Human agent:** eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators
- **Robotic agent:** cameras and infrared range finders for sensors; various motors for actuators.
- A **software agent** receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

- We use the term **percept** to refer to the agent's perceptual inputs at any given instant. An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

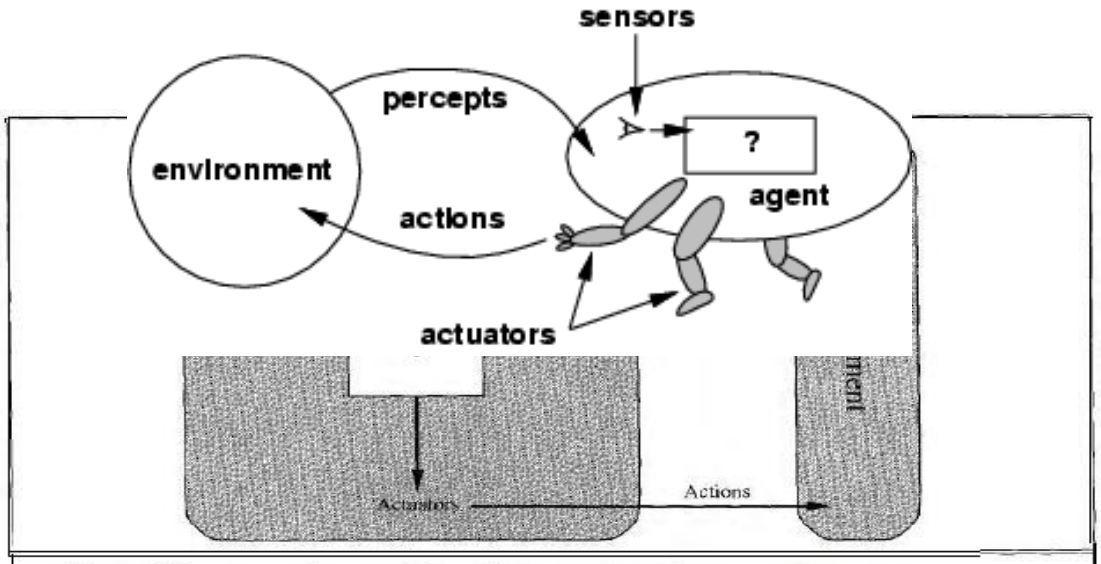


Figure 2.1 Agents interact with environments through sensors and actuators.

- We can imagine tabulating the agent function that describes any given agent; for most agents, this would be a very large table-infinite, in fact, unless we place a bound on the length of percept sequences we want to consider.
- Construct this table by trying out all possible percept sequences and recording which actions the agent does in response.

AGENT PROGRAM:

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and1 whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square.

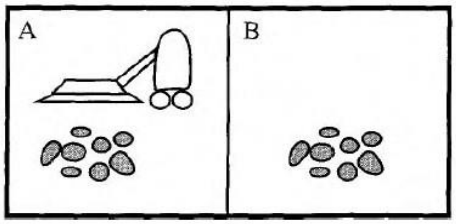


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
[A, *Clean*]	*Right*
[A, *Dirty*]	Suck
[B,Clean]	*Left*
[B,Dirty]	Suck
[A,Clean],[A,Clean]	*Right*
[A,Clean],[A,Dirty]	Suck
[A,Clean],[A,Clean],[A,Clean]	*Right*
[A,Clean],[A,Clean],[A,Dirty]	Suck

Partial tabulation of a simple agent function for the vacuum-cleaner world

GOOD BEHAVIOUR: THE CONCEPT OF RATIONALITY

Rational agents

- An agent should strive to "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful.
- Performance measure: An objective criterion for success of an agent's behavior

- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.
- **Rational Agent:** For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.
- Rationality is distinct from omniscience (all-knowing with infinite knowledge)
- Agents can perform actions in order to modify future percepts so as to obtain useful information (information gathering, exploration)
- An agent is autonomous if its behavior is determined by its own experience (with ability to learn and adapt)

PEAS

- PEAS: Performance measure, Environment, Actuators, Sensors
- Must first specify the setting for intelligent agent design
- Consider, e.g., the task of designing an automated taxi driver:
    - Performance measure
    - Environment
    - Actuators
    - Sensors
- Must first specify the setting for intelligent agent design
- Consider, e.g., the task of designing an automated taxi driver:
    - Performance measure: Safe, fast, legal, comfortable trip, maximize profits
    - Environment: Roads, other traffic, pedestrians, customers
    - Actuators: Steering wheel, accelerator, brake, signal, horn
    - Sensors: Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard

Agent: Medical diagnosis system

- Performance measure: Healthy patient, minimize costs, lawsuits
- Environment: Patient, hospital, staff
- Actuators: Screen display (questions, tests, diagnoses, treatments, referrals)
- Sensors: Keyboard (entry of symptoms, findings, patient's answers)

Agent: Part-picking robot

- Performance measure: Percentage of parts in correct bins
- Environment: Conveyor belt with parts, bins
- Actuators: Jointed arm and hand
- Sensors: Camera, joint angle sensors

Agent: Interactive English tutor

- Performance measure: Maximize student's score on test
- Environment: Set of students
- Actuators: Screen display (exercises, suggestions, corrections)
- Sensors: Keyboard

ENVIRONMENT TYPES

- **Fully observable (vs. partially observable)**: An agent's sensors give it access to the complete state of the environment at each point in time.
- **Deterministic (vs. stochastic):** The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)
- **Episodic (vs. sequential):** The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
- **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating. (The environment is semi dynamic if the environment itself does not change with the passage of time but the agent's performance score does)
- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions**.**
- **Single agent (vs. multiagent):** An agent operating by itself in an environment.

	Chess with a clock	**Chess without a clock**	**Taxi Driving**
Fully	Yes	Yes	No
Deterministic	Strategic	Strategic	No
Episodic	No	No	No
Static	Semi	Yes	No
Discrete	Yes	Yes	No
Single agent	No	No	No

- The environment type largely determines the agent design.
- The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

5. STRUCTURE OF AGENTS AND ITS FUNCTIONS

AGENT FUNCTIONS AND PROGRAMS

- An agent is completely specified by the agent function mapping percept sequences to actions.
- One agent function (or a small equivalence class) is rational.
- Aim: find a way to implement the rational agent function concisely.

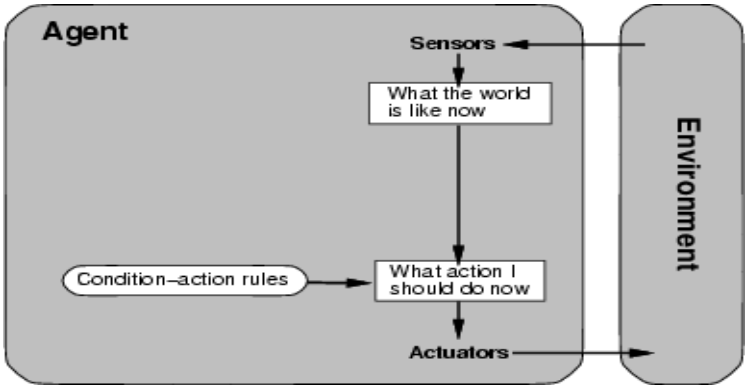
Table-lookup agent and its Drawbacks:

- Huge table
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

AGENT TYPES

- Four basic types in order of increasing generality:
  - Simple reflex agents
  - Model-based reflex agents
  - Goal-based agents
  - Utility-based agents
  - Learning agents

Simple reflex agents



The simplest kind of agent is the simplex reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

Example: Agent function for vacuum agent

Percept sequence	Action
[A, clean]	Right
[A, Dirty]	Suck
[B,Clean]	Left
[B, Dirty]	Suck
[A, clean],[A, clean]	Right
[A, clean],[A, Dirty]	Suck.
[A, clean],[A, clean], [A, clean]	Right
[A, clean],[A, clean], [A, dirty]	Suck

An agent program for this agent is

Function REFLEX-VACUUM-AGENT([location,status]) returns an action

If status=dirty then return suck

Else if location = A then return right

Else if location = B then return left

The vacuum agent program is very small. But some processing is done on the visual input to establish the condition-action rule.

Rectangles: Current internal state of the agent's decision process

Ovals: Background information used in the process

The **agent program** is given below:

Function SIMPLE-REFLEX-AGENT (percept) returns an action

Static: rules, a set of condition-action rules

State<- INTERPRET-INPUT (percept)

Rule<- RULE-MATCH (state, rules)

Action <- RULE-ACTION [rule]

Return action

Function

INTERPRET-INPUT: generates an abstracted description of the current state from the percept

RULE-MATCH: returns the first rule in the set of rules that matches the given state description.

This agent will work only if the correct decision can be made on the basis of only the current percept. i.e. only if the environment is fully observable.

MODEL-BASED REFLEX AGENTS

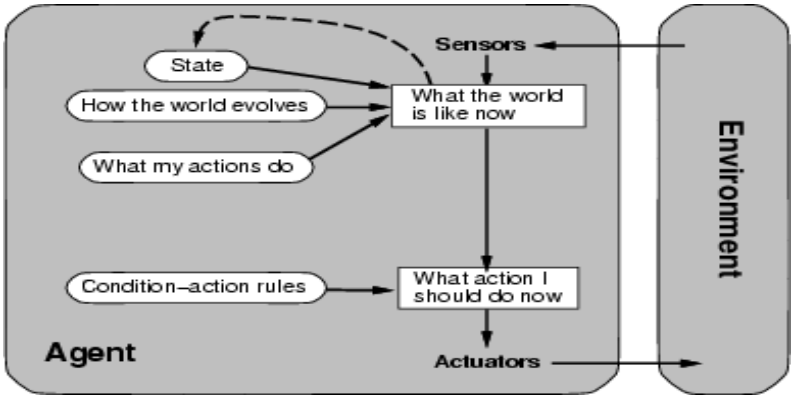
To handle partial observability, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Updating this internal state information requires two kinds of knowledge to be encoded in the agent program.

- How the world evolves independently of the agent
- How the agent's actions affect the world.

This knowledge can be implemented in simple Boolean circuits called model of the world. An agent that uses such a model is called a model-based agent.

The following figure shows the structure of the reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state.



The agent program is shown below:

Function REFLEX-AGENT-WITH-STATE(percept)returns an action Static: state, a description of the current world state

Rules, a set of condition-action rules

Action, the most recent action, initially none

State <- UPDATE-STATE (state, action , percept)

Rule<- RULE-MATCH (state, rules)

Action<- RULE-ACTION [rule]

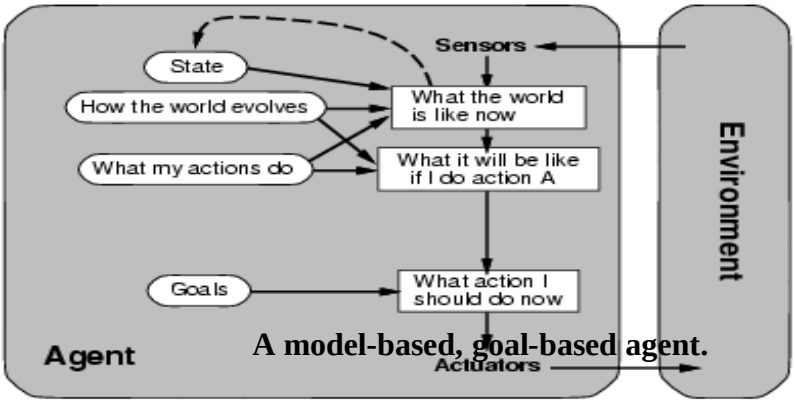
Return action

UPDATE-STATE: for creating the new internal state description.

GOAL-BASED AGENTS

Here, along with current-state description, the agent needs some sort of goal information that describes situations that are desirable – for eg, being at the passenger's destination.

Goal –based agents structure is shown below:



A model-based, goal-based agent.

Knowing about the current state of the environment is not always enough to decide what to do. **For example,** at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to.

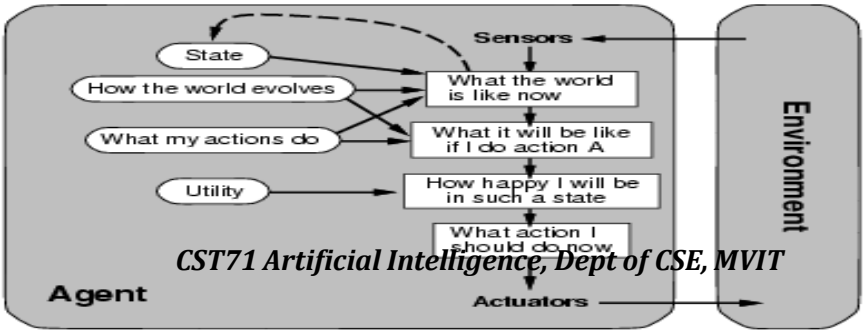
In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable-for example, being at the passenger's destination. The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal.

Sometimes **goal-based action selection is straightfor**ward, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky, when the agent has to consider long sequences of twists and turns to find a way to achieve the goal.

UTILITY-BASED AGENTS

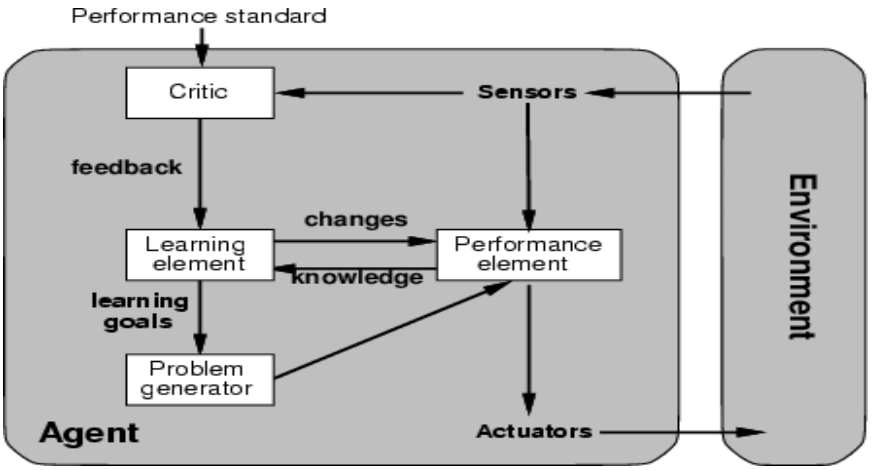
Goals alone are not enough to generate high-quality behavior in most environments. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved.

A utility function maps a state onto a real number, which describes the associated degree of happiness. The utility-based agent structure appears in the following figure.



LEARNING AGENTS

It allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. A learning agent can be divided into four conceptual components, as shown in figure:



Learning element: responsible for making improvement.

Performance element: responsible for selecting external actions

The learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future. The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent's success. The last component of the learning agent is the problem generator. It is responsible for suggesting actions that will lead to new and informative experiences.

5. PROBLEM SPACES AND SEARCH

Solving problems by searching

- *Problem-solving agents*
- *Problem types*
- *Problem formulation*
- *Example problems*
- *Basic search algorithms*

PROBLEM-SOLVING AGENTS

Problem solving agent is a goal-based agent decides what to do by finding sequences of actions that lead to desirable states. Let us take for an example, an agent in the city of Arad, Romania, enjoying a touring holiday. **Goal formulation**, based on the current situation and the

agent's performance measure, is the first step in problem solving. We will consider a goal to be a set of world states- exactly those states in which the goal is satisfied.

Problem formulation is the process of deciding what actions and states to consider, given a goal. Let us assume that the agent will consider actions at the level of driving from one major town to another. Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad. There are three roads out of Arad. The agent will not know which of its possible actions is best, because it does not know enough about the state that results from taking each action. If the agent has a map, it provides the agent with information about the states it might get itself into, and the actions it can take.

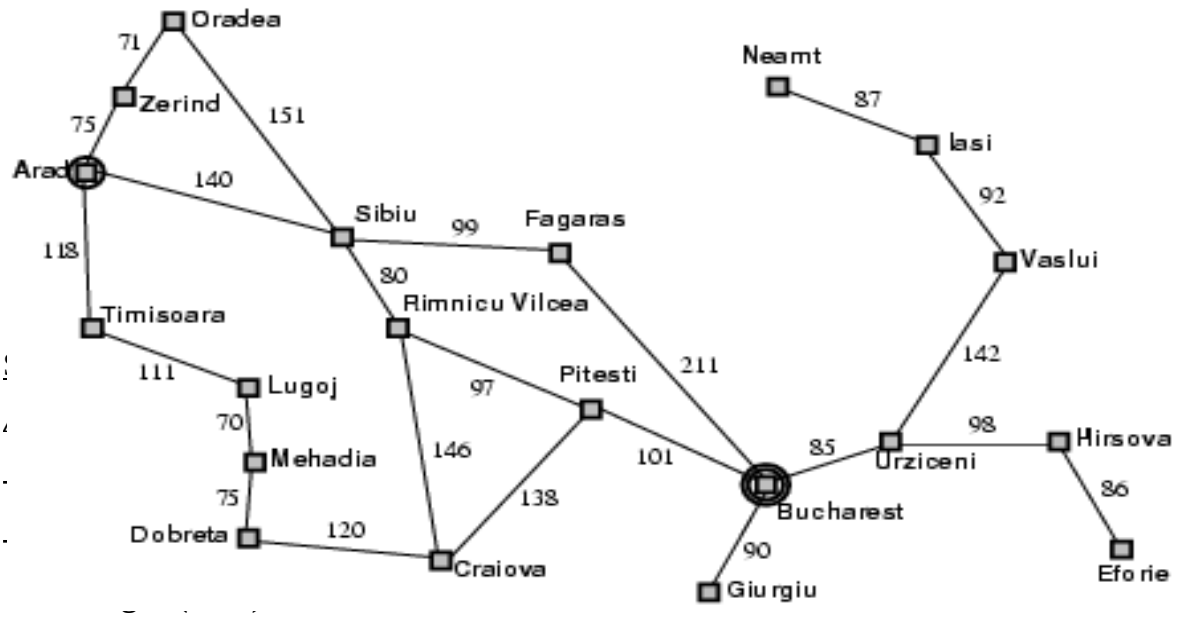
An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence. The process of looking for such a sequence is called a **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation
   state ← UPDATE-STATE(state, percept)
   if seq is empty then do
       goal ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, goal)
       seq ← SEARCH( problem)
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

This is called the **execution phase**. The design for such an agent is shown in the following function:

Example: Romania

- *On holiday in Romania; currently in Arad.*
- *Flight leaves tomorrow from Bucharest*
- *Formulate goal: be in Bucharest*
- *Formulate problem: states: various cities*
  actions: drive between cities
- *Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest*



- *goal test, can be*
  - explicit, e.g., *x* = "at Bucharest"
  - implicit, e.g., *Checkmate(x)*
- *path cost (additive)*
  - e.g., sum of distances, number of actions executed, etc.
  - *c(x,a,y)* is the step cost, assumed to be ¡Ã 0
- *A solution is a sequence of actions leading from the initial state to a goal state*

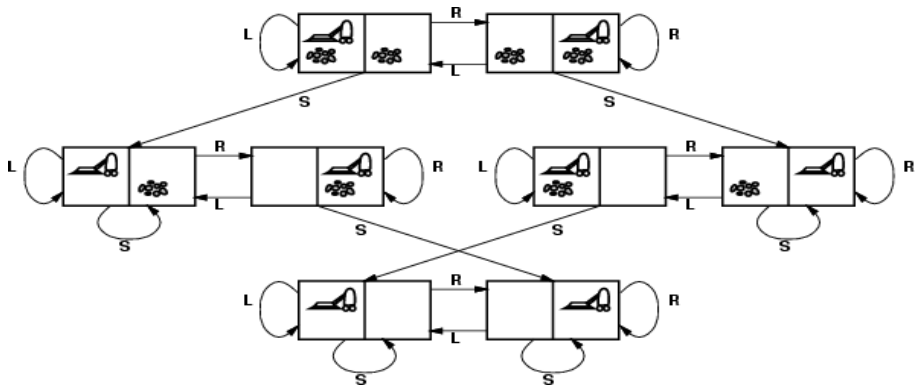
<u>Selecting a state space</u>

- *Real world is absurdly complex*

→ state space must be abstracted for problem solving

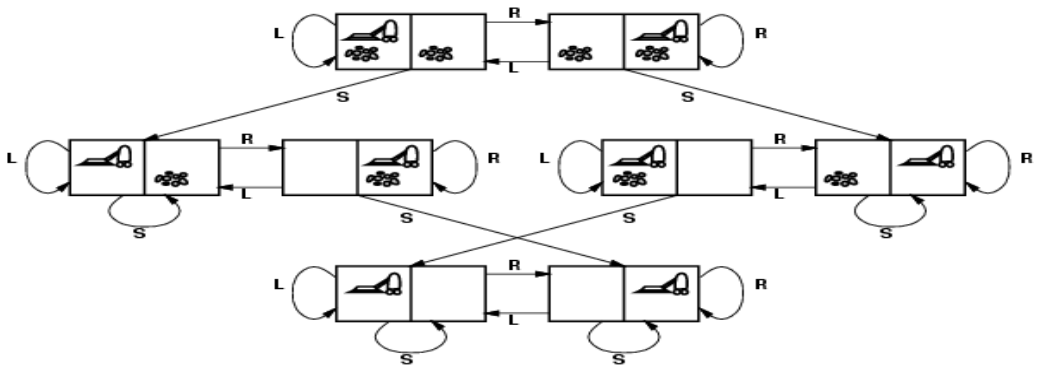
- *(Abstract) state = set of real states*
- *(Abstract) action = complex combination of real actions*
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- *For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"*
- *(Abstract) solution =*
  - set of real paths that are solutions in the real world
- *Each abstract action should be "easier" than the original problem*

Vacuum world state space graph



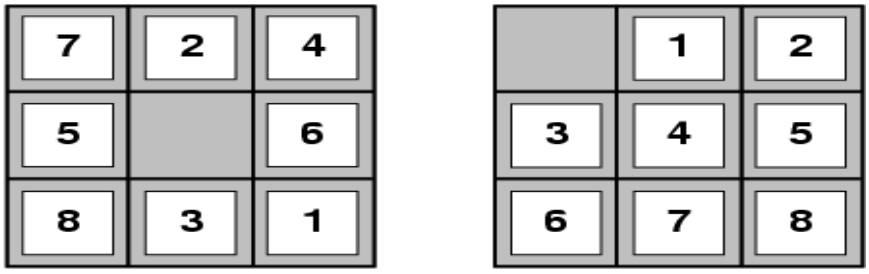
- *states?*
- *actions?*
- *goal test?*
- *path cost?*

Vacuum world state space graph



- *states?*
- *actions?* **Left, Right, Suck**
- *goal test?* ***no dirt at all locations***
- *path cost?* ***1 per action***

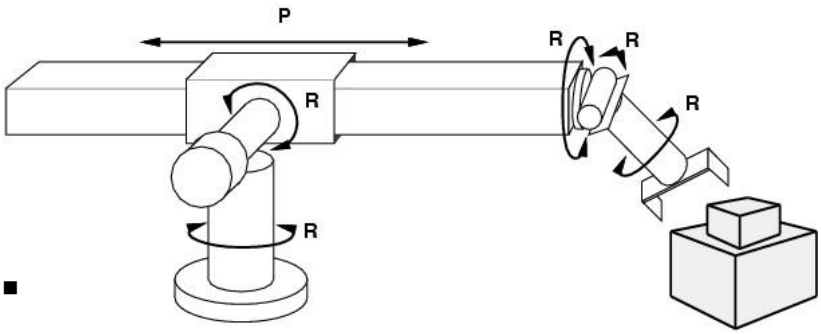
Example: The 8-puzzle



Start State Goal State

- *states?* ***locations of tiles***
- *actions?* ***move blank left, right, up, down***
- *goal test?* ***= goal state (given)***
- *path cost?* ***1 per move***

Example: robotic assembly



- ***states?*: real-valued coordinates of robot joint angles parts of the object to be assembled**
- *actions?: continuous motions of robot joints*
- *goal test?: complete assembly*
- *path cost?: time to execute*

SEARCHING FOR SOLUTIONS:

Solving the formulated problem can be done by a search through the state space. One of the search technique is an explicit **search tree** that is generated by the initial state and the successor function that together define the state space.

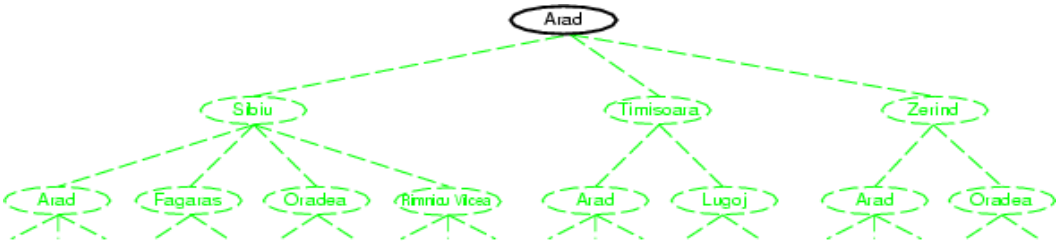
Tree search algorithms

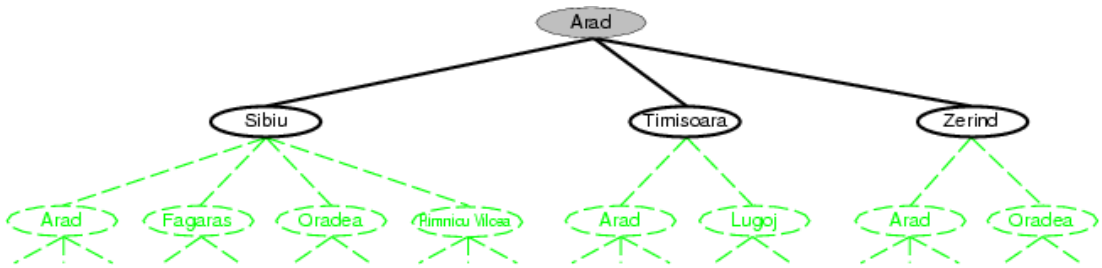
Basic idea

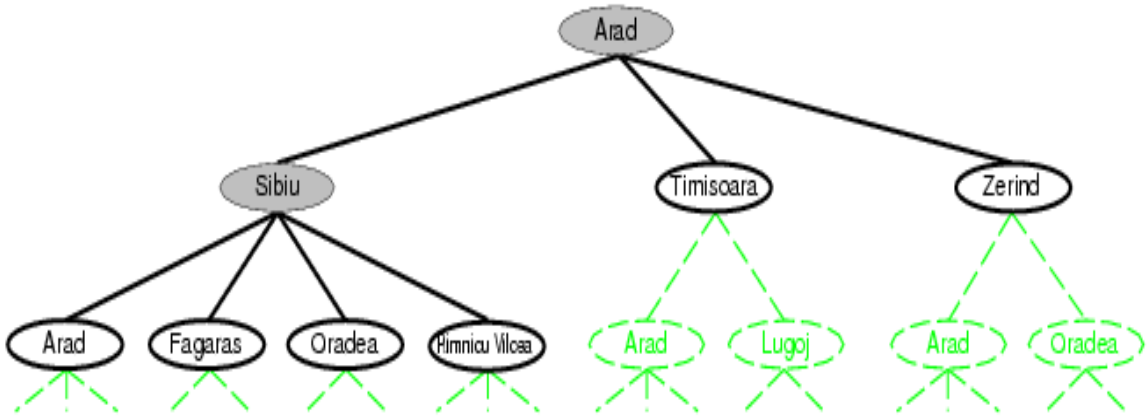
– offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

The following figure shows some of the expansions in the search tree for finding a route from Arad to Bucharest. The root of the search tree is a search node corresponding to the initial state, Arad. The first step is to test whether this is a goal state. If this is not the goal state, expand the current state by applying the successor function to the current state, thereby generating a new set of states.

Tree search example







Th ee-
sea

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

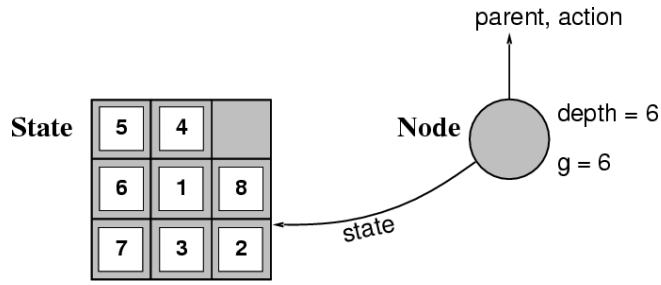
Assume that a node is a data structure with five components:

- STATE: the state in the state space to which the node corresponds
- PARENT-NODE: the node in the search tree that generated this node
- ACTION: the action that was applied to the parent to generate the node
- PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers; and
- DEPTH: the number of steps along the path from the initial state.

Implementation: states vs. nodes

- *A state is a (representation of) a physical configuration*
- *A node is a data structure constituting part of a search tree includes state, parent node, action, path cost* g(x), *depth*

The node data structure is depicted in the following figure:



The collection of nodes is implemented as a queue. The operations on a queue are as follows:

- MAKE-QUEUE(element….) creates a queue with the given element(s)
- EMPTY?(queue)returns true only if there are no more elements in the queue
- FIRST(queue) returns the first element of the queue
- REMOVE-FIRST(queue) returns FIRST(queue) and removes it from the queue.
- INSERT(element, queue) inserts an element into the queue and returns the resulting queue.
- INSERT-ALL(elements, queue) inserts a set of elements into the queue and returns the resulting queue.

With these definitions, the more formal version of the general tree search algorithm is shown below:

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

– *The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.*

Measuring problem-solving performance

– *A search strategy is defined by picking the order of node expansion*

– *Strategies are evaluated along the following dimensions:*

- completeness: does it always find a solution if one exists?

- time complexity: number of nodes generated
- space complexity: maximum number of nodes in memory
- optimality: does it always find a least-cost solution?

– ***Time and space complexity are measured in terms of***
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space

6. UNINFORMED SEARCH STRATEGIES

– Uninformed or blind search strategies use only the information available in the problem definition. Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies.
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

BREADTH-FIRST SEARCH

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their succesors, and so on. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element from the queue and place all the children at the end of the queue in any order.
5. Return to step 2.

– ***Implementation:***
  - By calling TREE-SEARCH with an empty fringe
  - ***fringe* is a FIFO queue, i.e., new successors go at end**

The following figure shows the progress of the search on a simple binary tree.

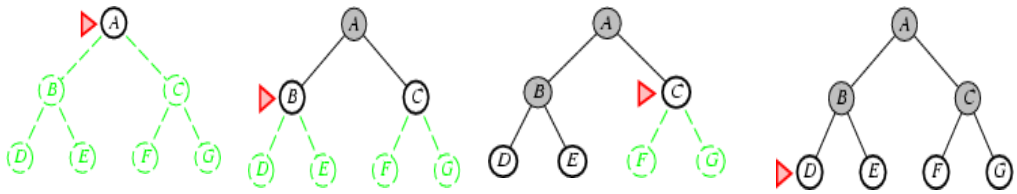


Figure: Breadth first search on a simple binary tree. At each state, the node to be expanded next is indicated by a marker.

Properties of breadth-first search

- ***Complete?*** ***Yes (if b is finite)***
- If the shallowest goal node is at some finite depth d, BFS will eventually find it after expanding all shallower nodes (b is a branching factor)
- ***Time?*** $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- ***Space?*** $O(b^{d+1})$ ***(keeps every node in memory)***
- We consider a hypothetical state space where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level, and so on. Now suppose that the solution is at depth d.
- ***Optimal?*** ***Yes (if cost = 1 per step)***
- BFS is optimal if the path cost is a nondecreasing funcion of the depth of the node.
- ***Space is the bigger problem (more than time)***

Uniform-cost search

BFS is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. Instead of expanding the shallowest node, Uniform-cost search expands the node n with the lowest path cost.

Implementation:

- *fringe* = queue ordered by path cost

- *Equivalent to breadth-first if step costs all equal*

- *Complete? Yes, if step cost >= ε*
- *Time? # of nodes with* g <= *cost of optimal solution,* $O(b^{ceiling(C^*/\varepsilon)})$ *where* C^* *is the cost of the optimal solution*
- *Space? # of nodes with* g <= *cost of optimal solution,* $O(b^{ceiling(C^*/\varepsilon)})$

– *Optimal? Yes – nodes expanded in increasing order of* g(n)

Depth-first search

– *Expand deepest unexpanded node*

– *Implementation:*

  – *fringe* = LIFO queue, i.e., put successors at front

Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element , and place the children at the front of the queue (in any order).
5. Return to step 2.

The progress of the search is illustrated in the following figure:

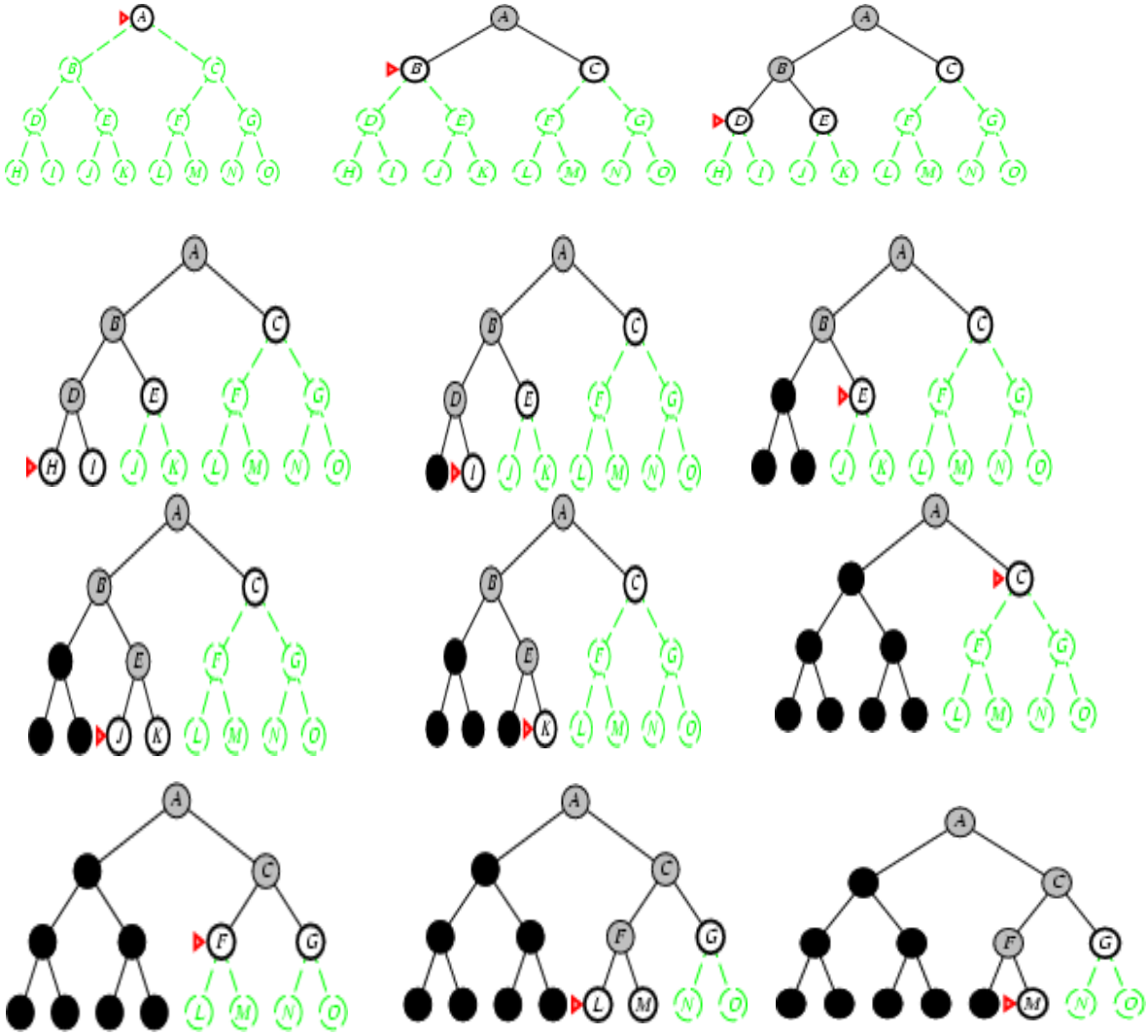


Figure: DFS on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no succesors and M is the only goal node.

Properties of depth-first search

- ***Complete?*** *No: fails in infinite-depth spaces, spaces with loops*
  - Modify to avoid repeated states along path

→ complete in finite spaces

- *Time?* $O(b^m)$*: terrible if* m *is much larger than* d
  - but if solutions are dense, may be much faster than breadth-first
- *Space?* O(bm), *i.e., linear space!*
- *Optimal? No*

Depth-limited search

The problem of unbounded trees can be alleviated by supplying DFS with a pre-determined depth limit.

= *depth-first search with depth limit* l, *i.e., nodes at depth* l *have no successors*

Depth-limited search will also be nonoptimal if we choose l<d. Its time complexity is $O(b^l)$ and its space complexity is O(bl).

▪***Recursive implementation:***

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

Depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

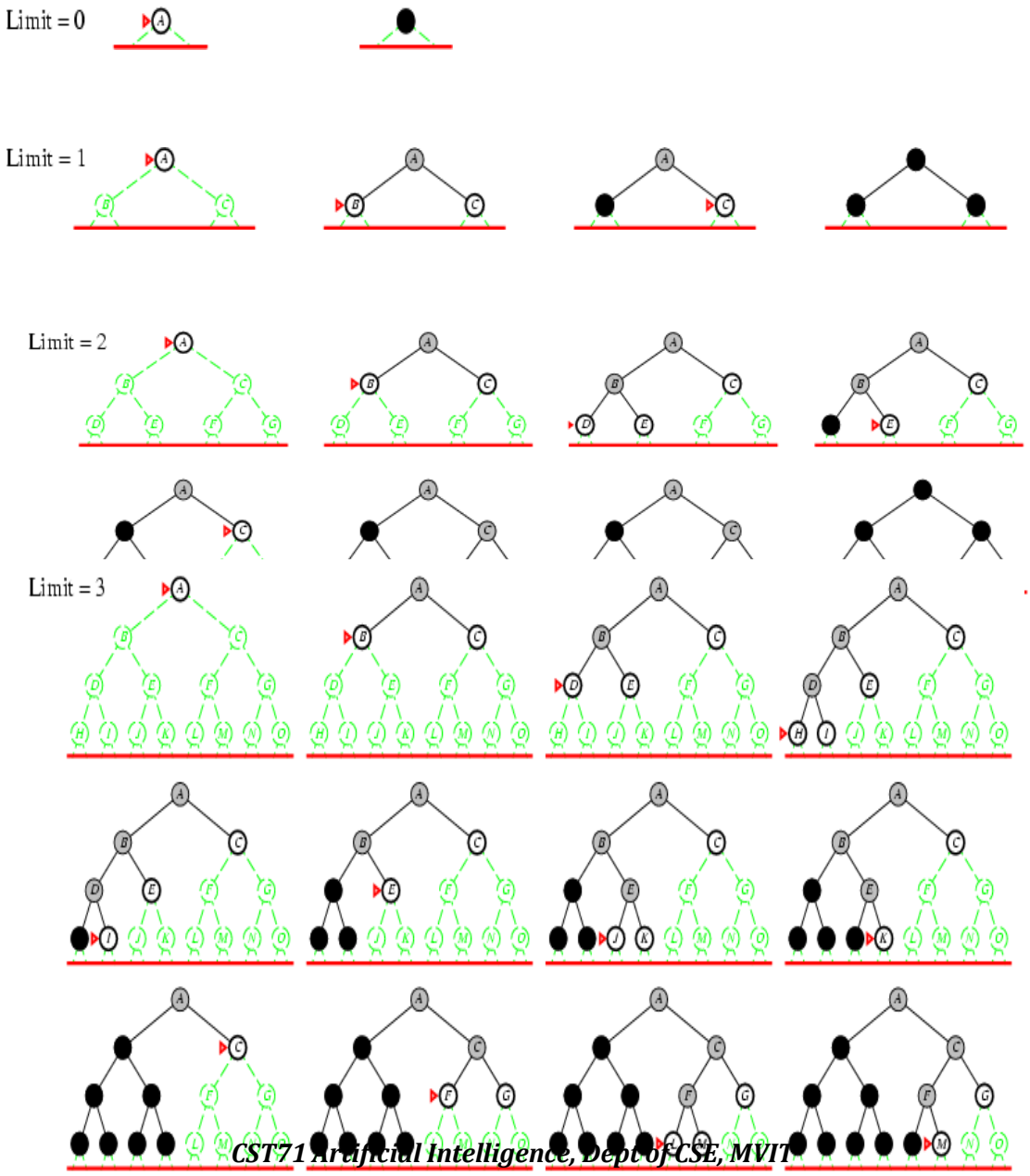
Iterative deepening search

Iterative deepening search is a strategy used in combination with DFS, that finds the best depth limit. It does this by gradually increasing the limit - first 0, then 1, then 2 and so on; until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. The algorithm is shown below:

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening combines the benefits of DFS and BFS. Like DFS, its memory requirements are very modest: O(bd). Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

The following figure shows four iterations of ITERATIVE-DEEPENING SEARCH on a binary search tree, where the solution is found on the fourth iteration.



▪*Number of nodes generated in a depth-limited search to depth* d *with branching factor* b*:*

$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

▪*Number of nodes generated in an iterative deepening search to depth* d *with branching factor* b*:*

$N_{IDS} = (d+1)b^0 + d\, b^{\wedge 1} + (d-1)b^{\wedge 2} + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

Properties of iterative deepening search

- *Complete?* *Yes*
- *Time?* $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- *Space?* O(bd)
- *Optimal?* *Yes, if step cost = 1*

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Bidirectional Search

The idea behind bi-directional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. Checking a node for membership in the other search tree can be done in constant time with a hash table, so the time complexity of bi-directional search is $O(b^{d/2})$.

Atleast one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is $O(b^{d/2})$ which is the weakness of the algorithm. The algorithm is complete and optimal if both searches are breadth-first;

7. HEURISTIC SEARCH TECHNIQUES - BEST-FIRST SEARCH

INFORMED SEARCH ALGORITHMS

Strategies that know whether one non-goal state is "more promising" than another are called **informed search or heuristic search strategies** Informed search strategy is the one that uses problem-specific knowledge beyond the definition of the problem itself.

- Best-first search
- Greedy best-first search
- A^* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

Review: Tree search

- A search strategy is defined by picking the order of node expansion

Best-first search

Best first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function f(n). The node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal. It can be implemented using a priority queue, a data structure that will maintain the fringe in ascending order of f – values.

Algorithm:

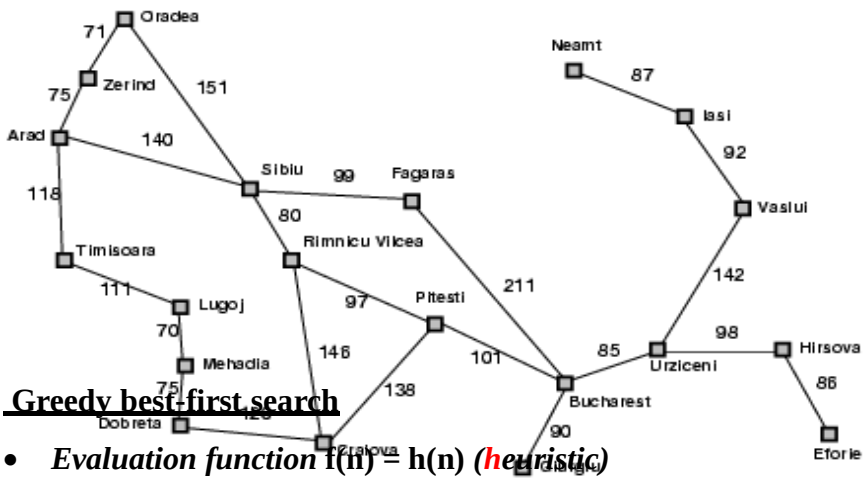
1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove the first element from the queue, expand it and compute the estimated goal distances for each child. Place the children on the queue(at either end) and arrange all queue elements in ascending order corresponding to goal distance from the front of the queue.
5. Return to step 2.

Best-first search uses different evaluation functions. A key component of these algorithms is a heuristic function, denoted h(n)

h(n)= estimated cost of the cheapest path from node n to a goal node.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest which is shown below:

Romania with step costs in km

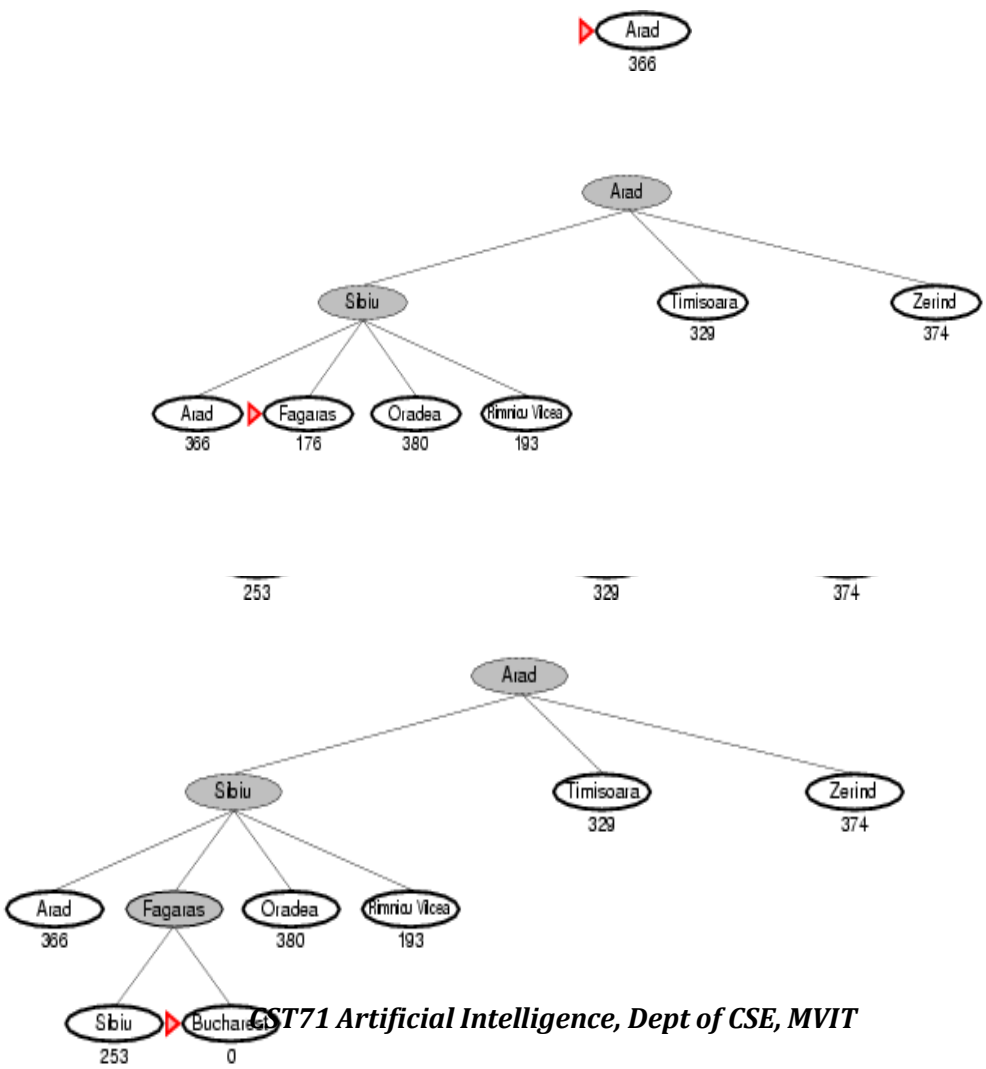


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- ***Evaluation function*** **f(n) = h(n)** ***(heuristic)***
- *= estimate of cost from* n *to* goal
- *e.g.,* $h_{SLD}(n)$ *= straight-line distance from* n *to Bucharest*
- *Greedy best-first search expands the node that appears to be closest to goal*

The progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest is shown in the following figure:



The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal. Greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence its search cost is minimal.

Properties of greedy best-first search

- *Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →*
- *Time?* $O(b^m)$, *but a good heuristic can give dramatic improvement*
- *Space?* $O(b^m)$ -- *keeps all nodes in memory*
- *Optimal? No*

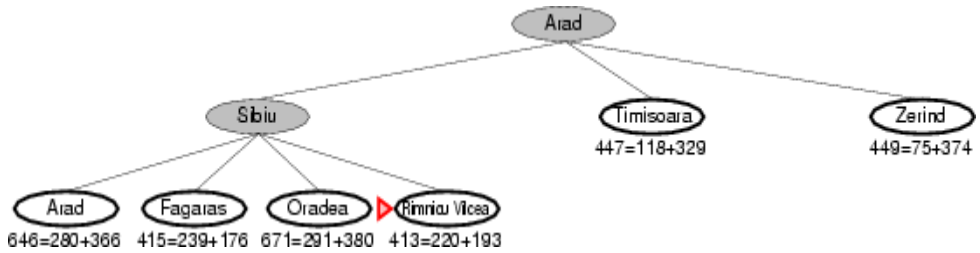
A* search: Minimizing the total estimated solution cost

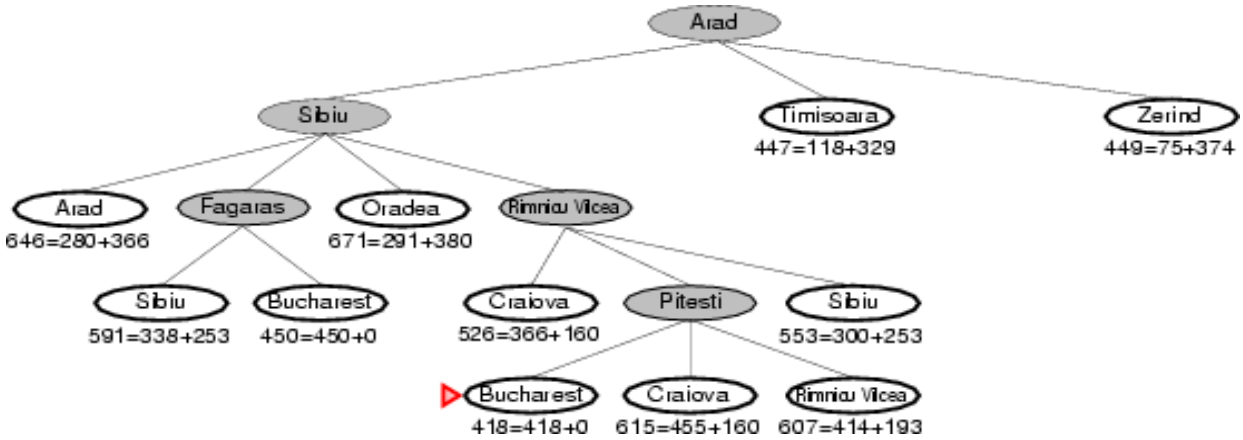
- *Idea: avoid expanding paths that are already expensive*
- *Evaluation function* $f(n) = g(n) + h(n)$
- $g(n)$ = *cost so far to reach* n
- $h(n)$ = *estimated cost from* n *to goal*
- $f(n)$ = *estimated total cost of path through* n *to goal*

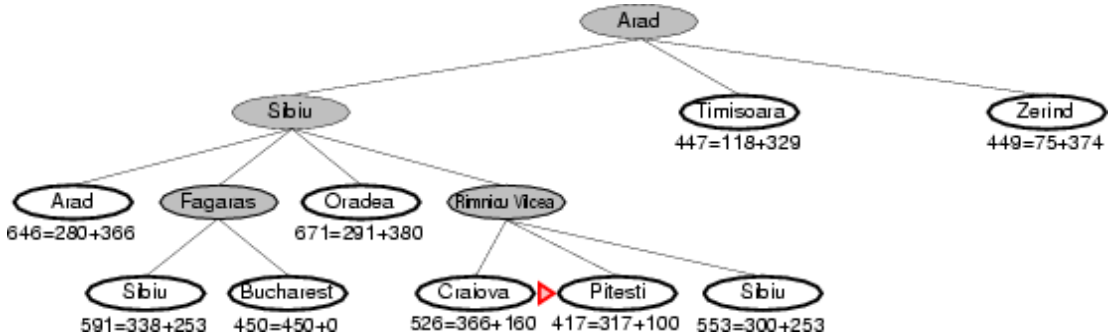
Algorithm:

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

The following figure shows an A* tree search for Bucharest.







Admissible heuristics

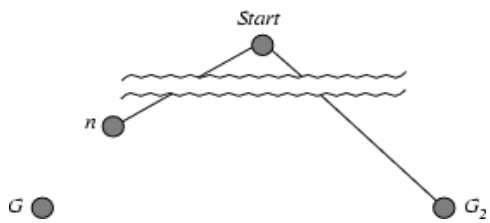
- A heuristic h(n) is admissible if for every node n,

$h(n) <= h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If h(n) is admissible, A^* using TREE-SEARCH is optimal

Optimality of A^* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

•Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

Properties of A*

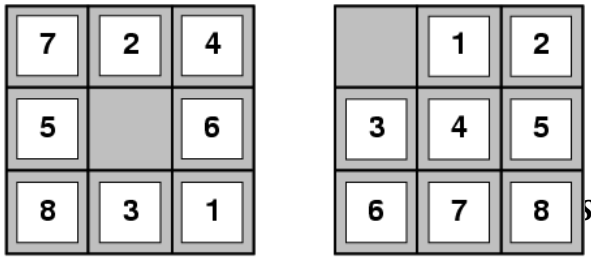
- Complete? Yes (unless there are infinitely many nodes with f <= f(G))
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

HEURISTIC FUNCTIONS

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions. This is sometimes called the city block distance or Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State Goal State

- $h_1(S) = ?$ 8, 8 tiles are out of position, so the start state would have $h_1=8$. h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

•$h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18 . h_2 is also admissible, because all any move can do is move one tile one step closer to the goal.

Relaxed problems

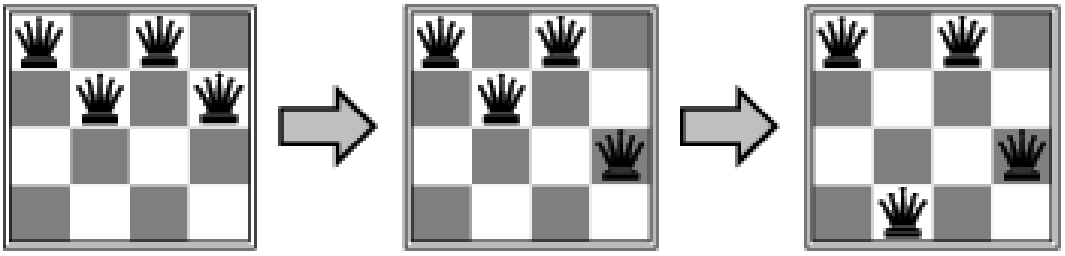
- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Local search algorithms

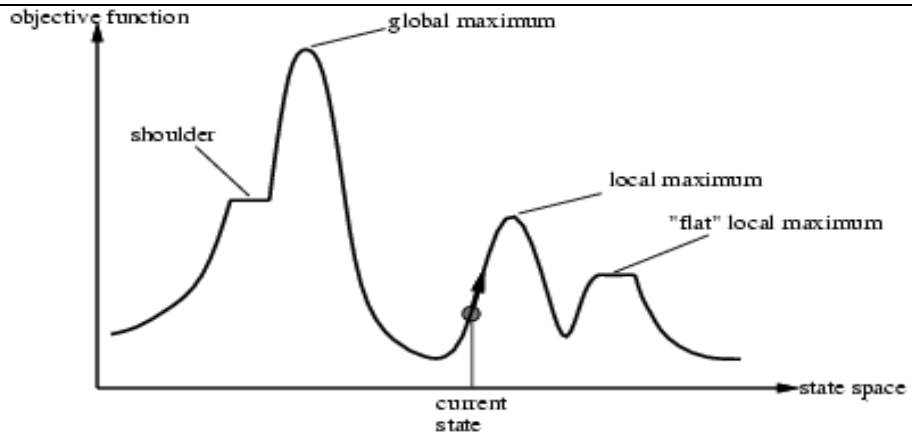
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: n-queens

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal



To understand local search, we will find it very useful to consider the state space landscape as shown in the following figure:



A landscape has both "location" and "elevation". If elevation corresponds to an objective function, then the aim is to find the highest peak - a global maximum. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

HILL-CLIMBING SEARCH

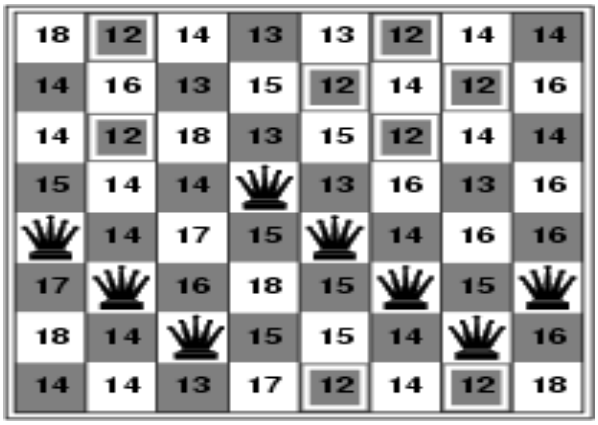
- ***"Like climbing Everest in thick fog with amnesia"***

The hill-climbing search algorithm is shown in the following function. It is simply a loop that continually moves in the direction of increasing value- that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

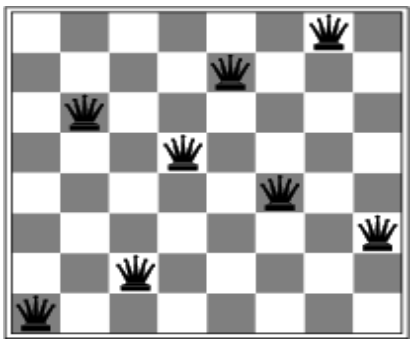
Problem: depending on initial state, can get stuck in local maxima

Hill-climbing search: 8-queens problem



- *h = number of pairs of queens that are attacking each other, either directly or indirectly*

•*h = 17 for the above state*



A local minimum in the 8-queens state space; the state has h=1 but every successor has a higher cost.

Hill climbing is sometimes called greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Hill climbing often gets stuck for the following reasons:

- **Local Maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.
- **Ridges**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux**: a plateau is an area of the state space landscape where the evaluation funcion is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

Simulated annealing search

- ***Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency***

A hill climbing algorithm that never makes “downhill” moves towards states with lower value is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk – that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient.Simulated annealing is the combination of hill climbing with a random walk.

The innermost loop of the simulated-annealing algorithm shown below is quite similar to hill climbing.

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

Instead of picking the best move, however, it picks a random move.

Properties of simulated annealing search

- One can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc

8. PROBLEM REDUCTION WITH AO* ALGORITHM.

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

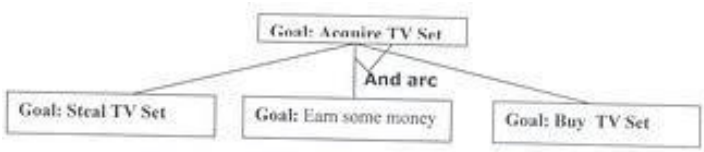


Figure shows AND - Or graph - an example.

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A* algorithm can not search AND - OR graphs efficiently. This can be understand from the give figure.

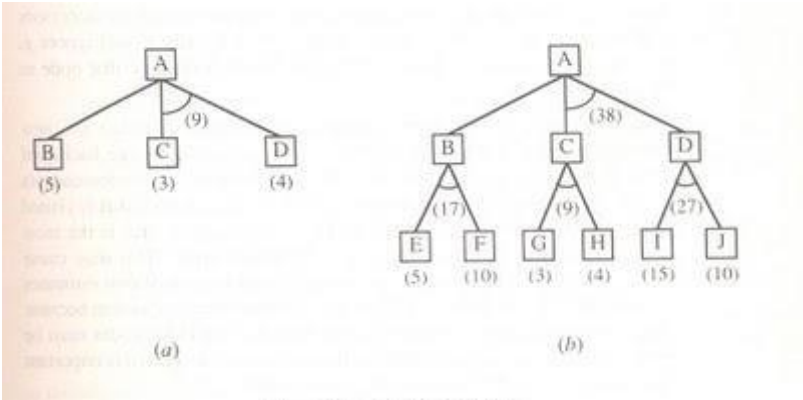


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

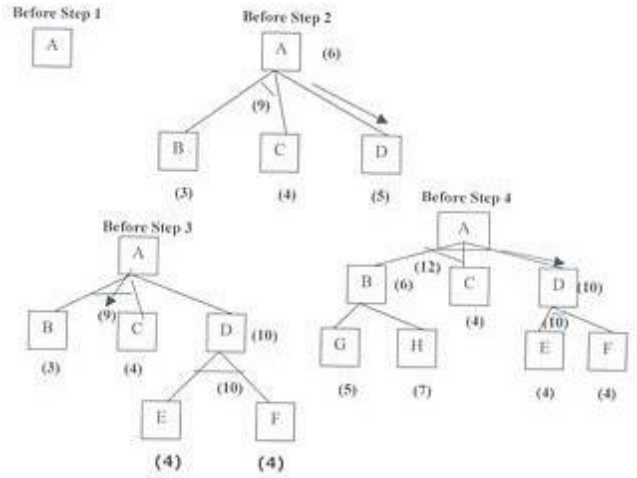
In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components.

With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.
3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:



Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better . it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes.

The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.

For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).
2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

(II) Generate the successors of NODE . if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one

called SUCCESSOR, that is not also an ancester of NODE do the following

(a) add SUCCESSOR to graph G

(b) if successor is not a terminal node, mark it solved and assign zero to its h ' value.

(c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following . let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;

(a) select a node from S call if CURRENT and remove it from S.

(b) compute h' of each of the arcs emerging from CURRENT , Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED.

(e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph . hence all the ancestors of CURRENT are added to S.

9. CONSTRAINT SATISFACTION

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables** **,**$X_1,X_2,\ldots X_n$,and a set of constraints $C_1,C_2,\ldots,C_m$. Each variable X_i has a nonempty **domain** D,of possible **values**. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.

A **State** of the problem is defined by an **assignment** of values to some or all of the variables,$\{X_i = v_i,X_j = v_j,\ldots\}$. An assignment that does not violate any constraints is called a

consistent or **legal assignment.** A complete assignment is one in which every variable is mentioned,and a **solution** to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an **objective function**.

Example for Constraint Satisfaction Problem :

Figure shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red,green,or blue in such a way that the neighboring regions have the same color. To formulate this as CSP ,we define the variable to be the regions :WA,NT,Q,NSW,V,SA, and T.

The domain of each variable is the set {red,green,blue}.The constraints require neighboring regions to have distinct colors;for example,the allowable combinations for WA and NT are the pairs

{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.

The constraint can also be represented more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as

{ WA = red, NT = green,Q = red, NSW = green, V = red ,SA = blue,T = red}.

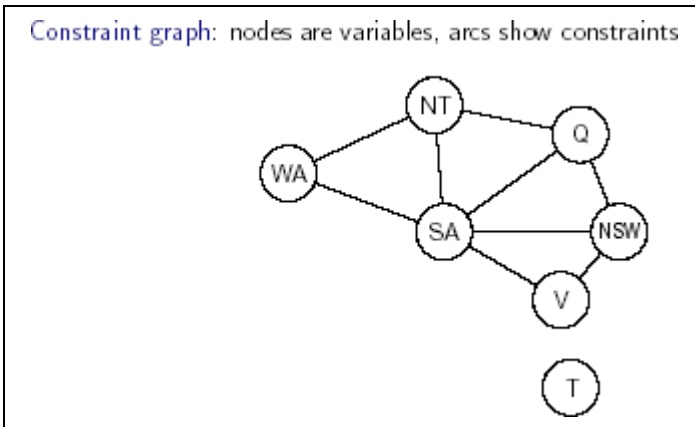


Figure: The map coloring problem represented as a constraint graph.

CSP can be viewed as a standard search problem as follows :

- **Initial state** : the empty assignment {},in which all variables are unassigned.
- **Successor function** : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
- **Goal test** : the current assignment is complete.
- **Path cost** : a constant cost(E.g.,1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.

Depth first search algorithms are popular for CSPs

VARIETIES OF CSPS

(i) Discrete variables

Finite domains

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain CSP,where the variables $Q_1,Q_2,\ldots..Q_8$ are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is $O(d^n)$ – that is,exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**,whose variables can be either *true* or *false*.

Infinite domains

Discrete variables can also have **infinite domains** – for example,the set of integers or the set of strings. With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as

$Startjob_1 + 5 <= Startjob_3$.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems,where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints :

(i) unary constraints involve a single variable.

Example : SA # green

(ii) Binary constraints involve paris of variables.

Example : SA # WA

(iii) Higher order constraints involve 3 or more variables.

Example : cryptarithmetic puzzles.

Backtracking Search for CSPs

The term backtracking search is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 2.17.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

Figure 2.17 A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search

10. MEANS ENDS ANALYSIS

Most of the search strategies either reason forward of backward however, often a mixture o 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 the arise when combining them together. Such a technique is called "Means - Ends Analysis".

The means -ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operate with a set of preconditions their application and difference functions that computes the difference between two state a(i) and s(j). A problem is solved using means - ends analysis by

1. Computing the current state s1 to a goal state s2 and computing their difference D12.
2. Satisfy the preconditions for some recommended operator op is selected, then to reduce the difference D12.

3. The operator OP is applied if possible. If not the current state is solved a goal is created and means- ends analysis is applied recursively to reduce the sub goal.

4. If the sub goal is solved state is restored and work resumed on the original problem.

means- ends analysis I useful for many human planning activities. Consider the example of planing for an office worker. Suppose we have a different table of three rules:

1. If in out current state we are hungry , and in our goal state we are not hungry , then either the "visit hotel" or "visit Canteen " operator is recommended.

2. If our current state we do not have money , and if in your goal state we have money, then the "Visit our bank" operator or the "Visit secretary" operator is recommended.

3. If our current state we do not know where something is , need in our goal state we do know, then either the "visit office enquiry" , "visit secretary" or "visit co worker " operator is recommended.

OTHER IMPORTANT EXAMPLE PROBLEMS IN UNIT:1

1. **Water jug problem:**

Problem statement:

You are given two jugs, 4 gallon one and 3 gallon one. Neither has nay measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water on 4 gallon jug?

Solution:

Defining and analyzing the problem:

- 1 galloon = 3.78541 litre
- The state space for the problem can be described as set of ordered pair of integer (x.y)

  Such that x= 0,1,2,3 or 4.

  y= 0,1,2 or 3.

  x= no of gallons of water in the 4 gallon jug.

y= no of gallons of water in the 3 gallon jug.

- The initial state = (0,0) and the Goal state = (2,n) for any value of n (where value of n). how many gallons need to be in 3 gallons jug is not specified in problem.

Assumptions:

- We need to take some assumptions not mentioned in the problem statement.
- We can fill a jug from pump that we can pour water out of a jug onto the ground.
- We can pour water from one jug to another. Since there are no measuring devices are available.
- To solve the water jug problem all we need is addition to the problem description given above is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is described on right side.

TABLE: WATER JUG PROBLEM

Steps	**Description**	**4 Gallon Jug**	**3 Gallon jug**
Step-1	Initially empty	0	0
Step-2	pour water on 3 gallon jug.	0	3
Step-3	pour water from jug 3 to jug 4	3	0 (empty jug)
Step-4	pump water to 3 gallon jug	3 (remaining 1 gallon)	3 (full)
Step-5	Pour water from 3 gallon to 4 gallon jug.	4 (full)	2
Step-6	empty 4 gallon water on ground	0	2
Step-7	Pour water from 3 gallon jug to 4 gallon jug.	2	0

Production rules for water jug problem.

1.	(x,y) →	(4,y)	fill the 4 gallon jug.
	Where x<4		
2.	(x,y) →	(x,3)	fill the 3 gallon jug.
	Where y<3		
3.	(x,y) →	(x-d,y)	pour some water out of 4 gallon jug.
	If x>0		
4.	(x,y) →	(x,y-d)	pour some water out from 3 gallon jug.
	(y>0)		
5.	(x,y) →	(0,y)	empty the 4 gallon jug on the ground
	If (x>0)		
6.	(x,y) →	(x,0)	empty the 3 gallon jug on the ground
	If (y>0		
7.	(x,y) →	(4,y-(4-x))	pour water from the 3 gallon jug into 4 gallon jug until 4 gallon jug is full.
	if x+y ≥4 and **y>0**		
8.	(x,y) →	(x-(3-y),3))	pour water from the 4 gallon jug into 3 gallon jug until 3 gallon jug is full.
	if x+y ≥3 and x>0		
9.	(x,y) →	(x+y,0)	pour all the water from 3 gallon into 4 gallon jug
	if x+y ≤ 4 and y>0		
10.	(x,y) →	(0,x+y)	pour all the water from 4 gallon into 3 gallon jug
	if x+y ≤ 3 and x>0		
11.	(0,2) →	(2,0)	pour the 2 gallons from the 3 gallons jug into 4 gallon jug.
12.	(2,y) →	(0,y)	empty the 2 gallons in the 4 gallons jug on the ground.

2. TRAVELLING SALESMAN PROBLEM

A salesman has a list of cities, each of which w 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 starts and finishes at any one of the cities.

Solution:

- Let G= (V,E) be a directed graph with each edge cost $c_{ij.}$
- C_{ij} is defined such that c_{ij}.>0 for all I and j and $c_{ij} = \infty$ if (i,j) not belongs to E.
- A tour of G is a directed simple cycle that includes every vertex in V.
- The cost of a tour is the sum of the cost of edges on the tour. The TVs problem is to find a tour of minimum cost.

  For example: consider a production environment in wchich several commodities are manufactured by same set of machines. The manufacture proceeds in cycles. In each production cycle, n different commodities are produced. When the machines are changed from production of commodity I to commodity j, a change over cost c_{ij} is incurred.

3. PUZZLE PROBLEM:

The 8 puzzle is a square tray in which are placed, eight square tiles. The remaining 9th square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position.

Solution:

Initial state

Step 1: Move 6th Tile to the empty space.

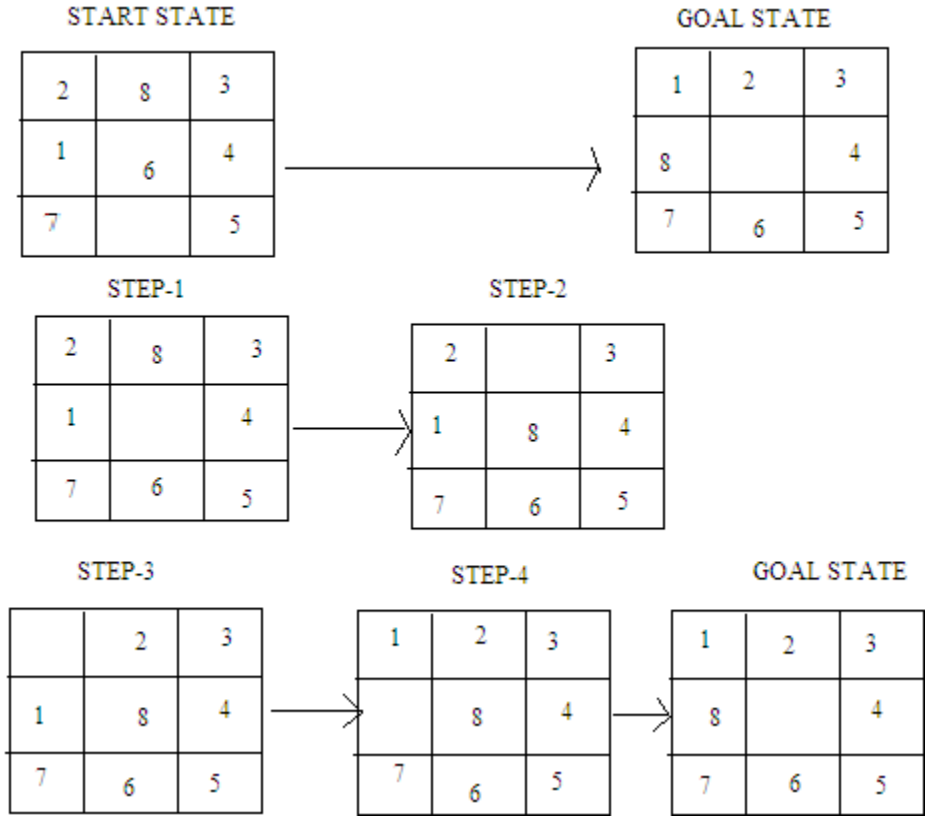
Step 2: Move 8th Tile to the empty space.

Step 3: Move 2th Tile to the empty space.

Step 4: Move 1th Tile to the empty space.

Step 5: Move 8th Tile to the empty space.

Step 6: Goal state is reached.



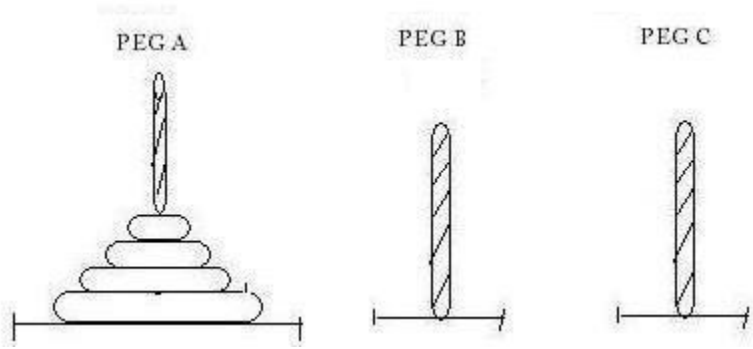
4. THE TOWER OF HANOI PROBLEM

The tower of Hanoi is one of example for recursion technique.

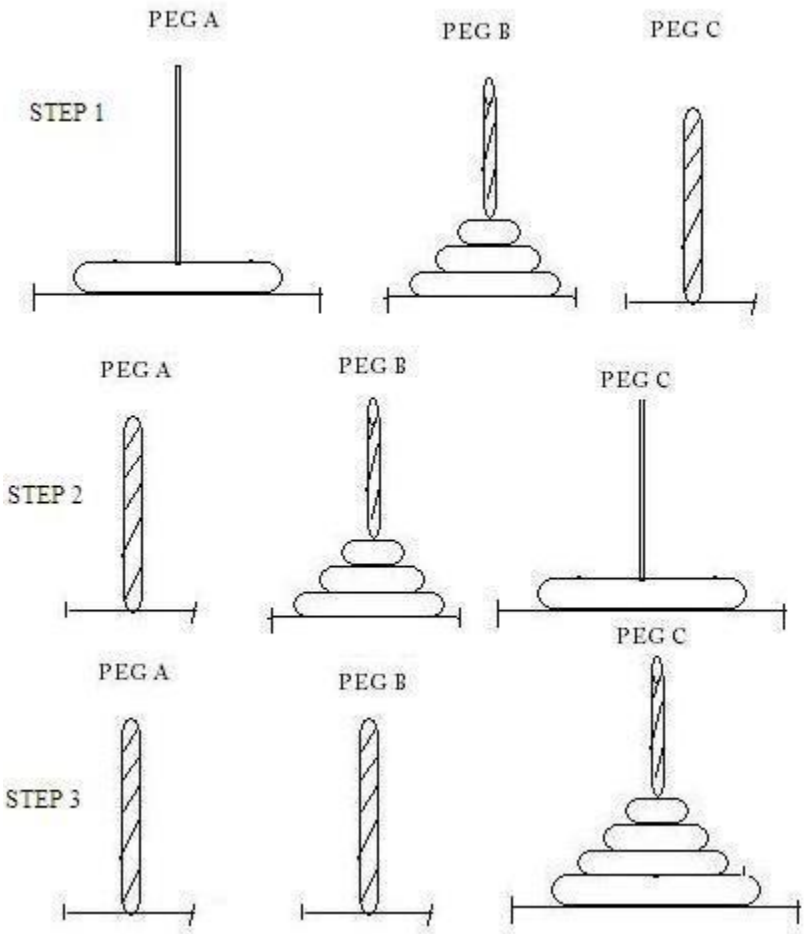
a) Three pegs A,B,C are exist. Five disks of differing diameter are placed on peg A so that a larger disk always below a smaller disk.

b) The object is to move the five disk to peg C, using peg B as auxiliary. Only the top disk on any peg moved to any other peg, and a larger disk may never rest on a smaller one.

PROBLEM:



Solution:



STEP 1: Moving A to B letting larger disk on A itself.

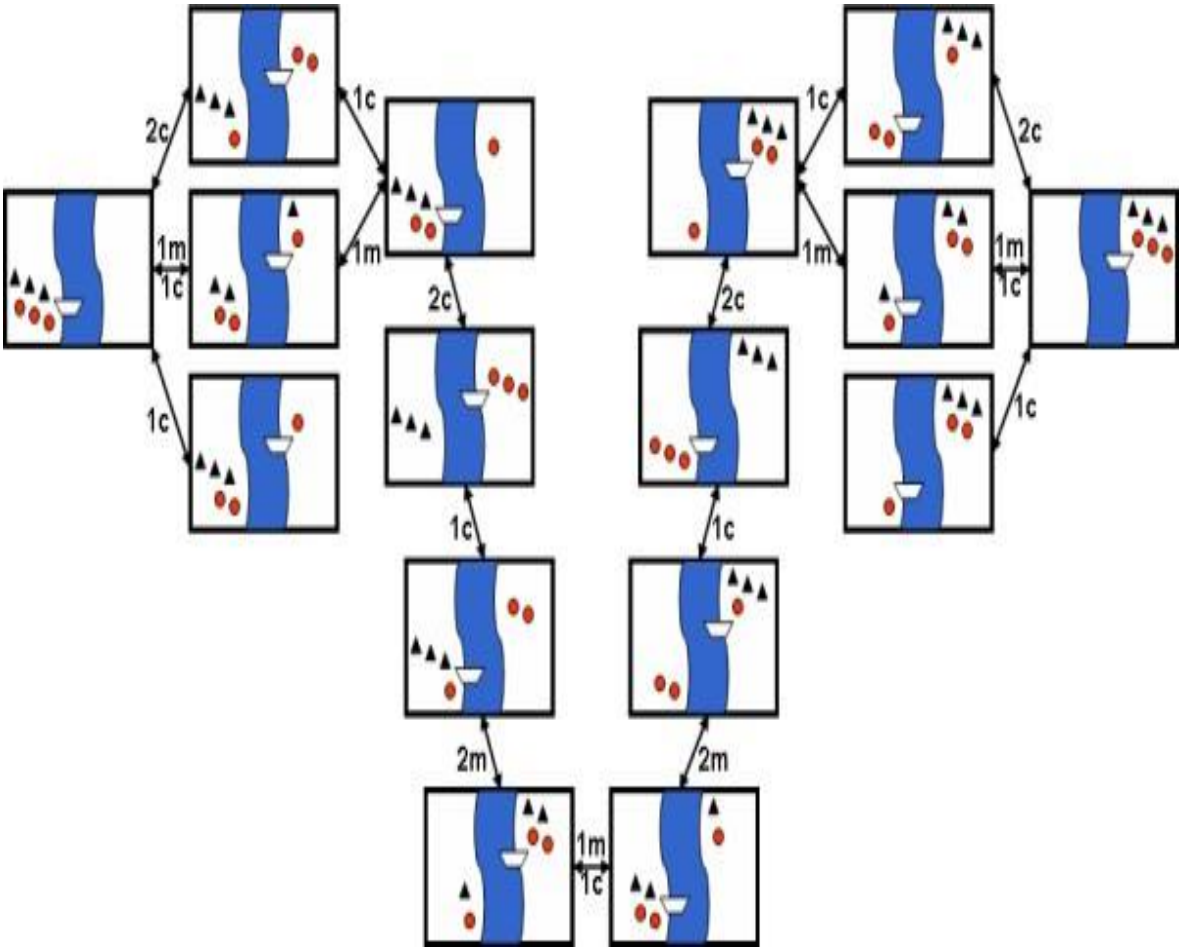
STEP 2: Move disk A to C.

STEP 3: Move disk from B to C we can reach GOAL STATE.

5. THE MISSIONARIES AND CANNIBALS PROBLEM:

Problem: Three missionaries and three cannibals find themselves on one side of a river. They have agreed that they would all like to get to 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 in such a way that the number of missionaries 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 the river without the missionaries risking being eaten?

Solution:



6. 8 QUEEN'S PROBLEM:

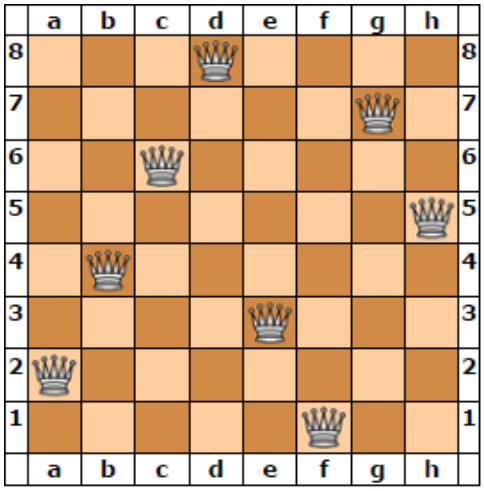
A classic combinatorial problem is to place eight queens on a 8x8 chess board so that no two attack, that is no two of them are on the same row, column, or diagonal.

Solution: To model this problem

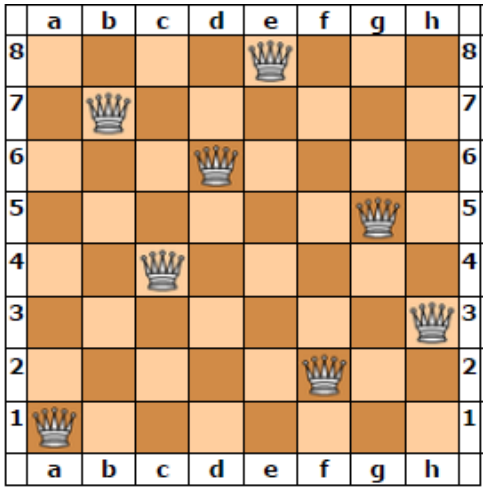
Assume that each queen ia in different column;

Assign a variable Ri (i=1 to N) to the queen in the ith column indicating the position of queen in the row.

Apply "no-threatening" constraints between each couple Ri and Rj of the queens and evaluate the algorithm.



Unique solution 1



Unique solution 2

Example:

The 8 queen puzzle has 92 distinct solutions. If the solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one. The puzzle has 12 unique solutions. Only two solutions are presented above.

UNIT II

Knowledge Representation: Approaches and issues in knowledge representation- Propositional Logic –Predicate logic-Forward and backward reasoning - Unification- Resolution- Weak slot-filler structure – Strong slot-filler structure- Knowledge- Based Agent

---

1. KNOWLEDGE REPRESENTATION

Let us first consider what kinds of knowledge might need to be represented in AI systems:

1. **Objects** -Facts about objects in our world domain. e.g. Guitars have strings, trumpets are brass instruments.
2. **Events** - Actions that occur in our world. e.g. Steve Vai played the guitar in Frank Zappa's Band.
3. **Performance** - A behavior like playing the guitar involves knowledge about how to do things.
4. **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.

a. Mapping between facts and representations

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

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

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

We can structure these entities at two levels

- **The knowledge level** - at which facts are described
- **The symbol level** - at which representations of objects are defined in terms of symbols that can be manipulated in programs

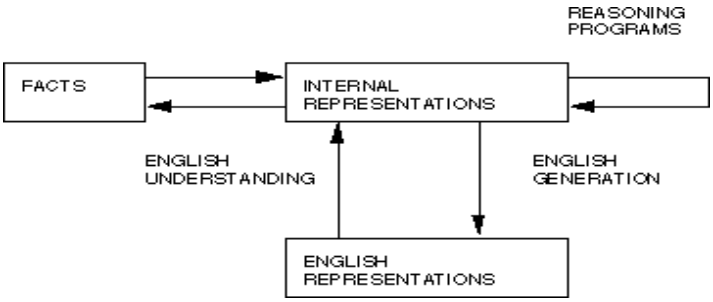


Figure: Mapping between facts and representations

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

$\forall x$: dog(x) $\rightarrow$ hasatail(x) We can then deduce:

hasatail(Spot)

Using an appropriate backward mapping function the English sentence Spot has a tail can be generated. The available functions are not always one to one but rather are many to many which is a characteristic of English representations.

The sentences All dogs have tails and every dog has a tail both say that each dog has a tail but the first could say that each dog has more than one tail try substituting teeth for tails. When an AI program manipulates the internal representation of facts these new representations should also be interpretable as new representations of facts.

b. <u>Intelligents agents should have following things</u>

- **Using Knowledge-** We have briefly mentioned where knowledge is used in AI systems. Let us consider a little further to what applications and how knowledge may be used.
- **Learning** – (Acquiring knowledge) This is more than simply adding new facts to a knowledge base. New data may have to be classified prior to storage for easy retrieval, etc. Interaction and inference with existing facts to avoid redundancy and replication in the knowledge and also so that facts can be updated.
- **Retrieval -** The representation scheme used can have a critical effect on the efficiency of the method. Humans are very good at it.
- **Reasoning** - Infer facts from existing

data.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.

The natural language reasoning requires inferring hidden state, namely, the intention of the speaker. When we say, "One of the wheel of the car is flat.", we know that it has three wheels left. Humans can cope with virtually infinite variety of utterances using a finite store of commonsense knowledge.

A logic consists of two parts, a language and a method of reasoning. The logical language, in turn, has two aspects, syntax and semantics. Thus, to specify or define a particular logic, one needs to specify three things:

- **Syntax:** The atomic symbols of the logical language, and the rules for constructing well-formed, non-atomic expressions (symbol structures) of the logic.
- **Semantics:** The meanings of the atomic symbols of the logic, and the rules for determining the meanings of non-atomic expressions of the logic. It specifies what facts in the world a sentence refers to. Hence, also specifies how you assign a truth value to a sentence based on its meaning in the world.
- **Facts** are claims about the world that are True or False, whereas a **representation** is an expression (sentence) in some language that can be encoded in a computer program and stands for the objects and relations in the world.

There are a number of logical systems with different syntax and semantics. We list below a few.

- Propositional logic
- All objects described are fixed or unique

  "John is a student" student(john) Here John refers to one unique person.
- First order predicate logic
- Objects described can be unique or variables to stand for a unique object

  "All students are poor" ForAll(S) [student(S) -> poor(S)]

  Here S can be replaced by many different unique students.

  This makes programs much more compact:

  eg. ForAll(A,B)[brother(A,B) -> brother (B,A)]

2. APPROACHES TO KNOWLEDGE REPRESENTATION?

Properties of a good system

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

- **Representational Adequacy**- the ability to represent the required knowledge.
- **Inferential Adequacy** - the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original
- **Inferential Efficiency** - the ability to direct the inferential mechanisms into the most productive directions by storing appropriate guides;
- **Acquisitional Efficiency** - the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

A. 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.

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

Figure: Simple Relational Knowledge

We can ask things like:

- Who is dead?
- Who plays Jazz/Trumpet etc.?

This sort of representation is popular in database systems.

B. Inheritable knowledge

Relational knowledge is made up of objects consisting of

- Attributes
- Corresponding associated values.

We extend the base more by allowing inference mechanisms:

- Property inheritance

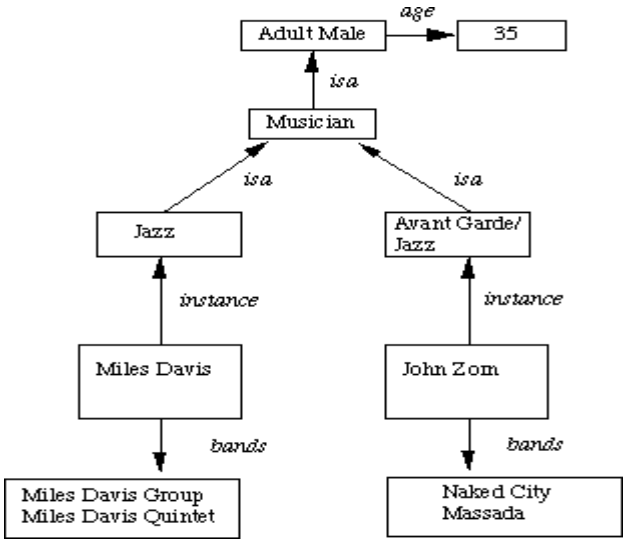
- Elements inherit values from being members of a class.
- Data must be organized into a hierarchy of classes.
- 

Fig. Property Inheritance Hierarchy

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

<u>Algorithm: Property Inheritance</u> (to retrieve a value for an attribute of an instance object)

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

C. <u>Inferential Knowledge</u>

Represent knowledge as formal logic:

All dogs have tails: $\forall x$: dog(x) $\rightarrow$hasatail(x)

Advantages:

- A set of strict rules.
  - Can be used to derive more facts.
  - Truths of new statements can be verified.
  - Guaranteed correctness.

- Many inference procedures available to in implement standard rules of logic.
- Popular in AI systems. e.g Automated theorem proving.

D. Procedural

KnowledgeBasic idea:

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

Advantages:

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

Disadvantages:

1. Completeness - not all cases may be represented.
2. Consistency - not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.
3. Modularity is sacrificed. Changes in knowledge base might have far-reaching effects.
4. Cumbersome control information.

3. ISSUES IN KNOWLEDGE REPRESENTATION

Overall issues

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

1. Are any attributes of objects so basic that they occur in almost every problem domain?
2. Are there any important relationships that exist among attributes of objects?
3. At what level should knowledge be represented? Is there a good set of primitives into which all knowledge can be broken down?
4. How should sets of objects be represented?
5. Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

We will see each of these questions briefly in the next five sections.

A. Important Attributes

Are there any attributes that occur in many different types of problem? There are two instance and isa and each is important because each supports property inheritance.

B. Relationships among Attributes:

The attributes that we use to describe objects are themselves entities that we represent. What properties do they have independent of the specific knowledge they encode? There are four such properties that deserve are mentioned below.

1. Inverses.
2. Existence in an isa hierarchy.
3. Techniques for reasoning about values.
4. Single valued attributes.

C. Inverses

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

band(John Zorn,Naked City)

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

Another representation is band = Naked City

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

D. Existence in an isa hierarchy:

Just as there are classes of objects and specialized subsets of those classes, there are attributes and specialization of attributes. Consider for example: the attribute height. In the case of attributes they support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

E. Techniques for reasoning about values:

Sometimes values of attributes are specified explicitly when a knowledge base is created. Several kinds of information can play a role in this reasoning including:

1. Information about the type of value- for (eg): the value of height must be a number measure in a unit of length.
2. Constraints on the value, often stated in terms of related entities- for (eg): the age of the person cannot be greater than the age of either of that person’s parents.

3. Rules for computing the values when it is needed.
4. Rules that describe actions that should be taken if a value ever becomes known.

F. Single valued attributes :

A specific but very useful kind of attribute is one that is guaranteed to take a unique value. For example: a baseball player can, at any one time, have only a single height and be a member of only one team.

G. Choosing the granularity of representation:

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

If Tom feeds a dog then it could become:

feeds(tom, dog)

If Tom gives the dog a bone like:

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

In any sense does giving an object food constitute feeding?

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

But we need to add certain inferential rules.

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

Clearly the separate levels of understanding require different levels of primitives and these need many rules to link together apparently similar primitives.

Obviously there is a potential storage problem and the underlying question must be what level of comprehension is needed.

H. Representing set of objects:

It is important to be able to represent sets of objects for several reasons. One is that there are some properties that are true of sets that are not true of the individual members of a set.

Example: Consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world." The only way to represent the facts described in these sentences is to attach assertions to the sets representing people, sheep, and English speakers, since, for example, no single English speaker

can be found all over the world. The other reason that it is important to be able to represent sets of objects is that if a property is true of all elements of a set, then it is more efficient to associate objects is that if a property is true of all elements of a set.

I. Finding the right structure as needed:

In order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems.

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situation.
- 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.

J. Selecting an initial structure

The selecting candidate knowledge structures to match a particular problem solving situation is a hard problem, there are several ways in which it can be done. Three important approaches are the following.

- Index the structures directly by the significant English words that can be used to describe them.
- Consider each major concept as a pointer to all of the structures in which it might be involved.
- Locate one major clue in the problem description and use it to select an initial structure.

K. Revising the choice when necessary

Once the candidate knowledge structure is detected, we must attempt to do a detailed match of it to the problem at hand. Depending on the representation we are using the details of the matching process will vary.

When the process runs into a snag, though, it is often not necessary to abandon the effort and start over. Rather there are a variety of things that can be done. The following things can be done:

- Select the fragments of the current structure that do correspond to the situation and match them against candidate alternatives.
- Make an excuse for the current structure's failure and continue to use it.
- Refer to specific stored links between structures to suggest new directions in which to explore.

4. PROPOSITIONAL LOGIC

Logic: The logic plays an important role in the design of almost all the systems in engineering and sciences. Designing the present days computer is complex task. This design involves two types of design namely

a. Hardware design
b. Software design

These are based on mathematical logic called formal logic.

Propositional logic: The propositional logic deals with individual propositions, which are viewed as atoms, i.e these cannot be further broken into smaller constituents. Though propositional logic is not powerful than predicate logic but it has great importance in number of applications, particularly in the design of computers at hardware level.

For building a propostional logic, first we describe the logic with the help of a formula called Well-Formed Formula (wff). The propositional logic contains variables such as p, q, r, s, t, p1, p2, p3, q1, q2, q3, r1, r2,r3 etc…

The other symbols of propositional logic are

$\neg p$	(read "not *p*")	the **negation** of *p*
$p \wedge q$	(read "*p* **and** *q*")	the **conjunction** of *p* and *q*
$p \vee q$	(read "*p* **or** *q*")	the **disjunction** of *p* and *q*
$p \rightarrow q$	(read "*p* **implies** *q*")	the **implication** of *q* from *p*
$p \leftarrow q$	(read "*p* **if** *q*")	the **implication** of *p* from *q*
$p \leftrightarrow q$	(read "*p* **if and only if** *q*" or "***p* is equivalent to** *q*")	

The examples of propositions are given below.

P= rama is student of second year.

Q= rama participates in tennis.

The following formulas can be constructed using above.

p∧q= rama is student of second year ***and*** *rama participates in tennis.*

p∨q= rama is student of second year ***or*** *rama participates in tennis.*

¬p∧q= rama is not student of second year ***and*** *rama participates in tennis.*

¬p→q= if rama is not student of second year ***then*** *rama participates in tennis.*

Note that we only talk about the truth value in an interpretation. Propositions may have different truth values in different interpretations.

---

p	*q*	*¬p*	*p ∧q*	*p∨q*	*p ←q*	*p →q*	*p ↔q*
True	*True*	*false*	*True*	*true*	*true*	*True*	*true*
True	*False*	*false*	*False*	*true*	*true*	*False*	*false*
False	*True*	*true*	*False*	*true*	*false*	*True*	*false*
False	*False*	*true*	*False*	*false*	*true*	*True*	*true*

Figure Truth table defining ¬, ∧, ∨, ←, →, and ↔

If an expression is true, for all the rows (i.e. for all possible values of variables in that expression) then it is called as tautology, and we write it as ⊨*u*.

The following are some of laws and its equivalences.

S.No	Equivalence	Name of the Equivalence
1.	*¬p(p ∧q)= ¬p∨¬q* *¬(p∨q)= ¬p∧¬q*	Demorgans Law
2.	p∧*Tp=p* *p∨Cp=p*	Identity laws
3.	p∧¬*p=Cp* *p∨¬p=Tp*	Inverse laws
4.	p∨*Tp=Tp* *p∧ Cp =Cp*	Domination Law
5.	p∨*p=p*	Idempotent laws

	$p \wedge p = p$	
6.	$p \rightarrow q = \neg p \vee q$	Implication laws
7.	$P \vee q = q \vee p$ $p \wedge q = q \wedge p$	Commutative laws
8.	$P \vee (q \vee r) = (p \vee q) \vee r)$ $p \wedge (q \wedge r) = (p \wedge q) \wedge r$	Associative laws
9.	$P \vee (q \wedge r) = (p \vee r) \wedge (p \vee r)$ $P \wedge (q \vee r) = (p \wedge r) \vee (p \wedge r)$	Distributive laws

Example1: show that (*$p \wedge q) \rightarrow (p \vee q)$ *is a Tautology, i.e interpretation of this sentence is always true.*

Solution: *it can be proved that above logical expression is a tautology using rules of logical equivalences.*

$(p \wedge q) \rightarrow (p \vee q) = \neg(p \wedge q) \vee (p \vee q)$ *(using implication laws)*

$= (\neg p \vee \neg q) \vee (p \vee q)$

$= (\neg p \vee p) \vee (\neg q \vee p)$ *(by rearrangement of terms)*

$= Tp \vee Tp$ *(using inverse law)*

$= T$

Example2: Show that (p$\vee q) \wedge \neg(\neg p \wedge q)$ *and p are logically equivalence.*

Solution: $(p \vee q) \wedge \neg(\neg p \wedge q)$

$= (p \vee q) \wedge (\neg\neg p \vee \neg q)$

$= (p \vee q) \wedge (p \vee \neg q)$

$= p \vee (q \wedge \neg q)$

$= p \vee F$

$= p$

Do the following problems for practice

1. **Find out using truth table whether implication is tautology.**

   a. $(p \wedge r) \rightarrow p$

   b. $(p \wedge q) \rightarrow (p \rightarrow q)$

   c. $((p \vee (\neg(q \wedge r))) \rightarrow ((p \leftrightarrow q) \vee r)$

2. **Show that tautology without using Truth table.**

   a. $(p \wedge (p \rightarrow q)) \rightarrow q$

   b. $(\neg p \wedge (p \vee q)) \rightarrow q$

3. **Verify whether following are tautology.**

   a. $(\neg p \wedge (p \rightarrow q)) \rightarrow \neg q$

   b. $(\neg q \wedge (p \vee q)) \rightarrow \neg p$

4. **Show that pairs of expressions are logically equivalent.**

   a. $\neg p \leftrightarrow q$ *and* $p \leftrightarrow \neg q$

   b. $\neg(p \wedge q)$ *and* $(\neg p) \vee (\neg q)$

   c. $\neg p \rightarrow \neg q$ *and* $q \rightarrow p$

Inference rules:

Inference rules are used to infer new knowledge in the form of propositions from the existing knowledge. The knowledge propositions are logical implications of the existing knowledge.

a. **Modus Ponens: This rule is also called rule of detachment. Symbolically it is written as [p**$\wedge (p \rightarrow q)] \rightarrow q$

Or

P

$\underline{p \rightarrow q}$

---

q (p and p implies q can be written as q)

b. **Modus Tollens: The inference rule of modus tollens is a logical implication specified by** $[(p \rightarrow q) \wedge \neg q] \rightarrow \neg p$

Which can be written as

$p \rightarrow q$

$\neg q$

$\neg p$ *(p implies q and negation q can be written as negation p)*

c. **Law of syllogism: This rule of inference is expressed by the logical implication:**

$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$

It can also be expressed in tabular form as:

$p \rightarrow q$

$q \rightarrow r$

$p \rightarrow r$ *(p implies q and q implies r can be written as p implies r)*

Example: Prove whether the following argument is valid, contradiction or satisfied.

"rajini is preparing food in kitchen. If rajini is preparing food in kitchen then she is not playing violin. If she is not playing violin, then she is not learning music. Therefore, rajini is not learning music."

Solution: The above statements can be specified in the form of logical implication if the following propositional symbols are assigned to statements:

p= rajini is preparing food in kitchen

q= rajini is playing violin

r= rajini is learning music

now, the argument can be expressed in the form of a propositional formula:

$p \wedge ((p \rightarrow \neg q) \wedge (\neg q \rightarrow \neg r)] \rightarrow \neg r$

the above wff can also be represented in the tabular form along with inference as follows

p

$p \rightarrow \neg q$

$\neg q \rightarrow \neg r$

$\neg r$ *(p and p implies negation q and negation q implies negation r can be as* $\neg r$*)*

To prove this logical implication, we carry out following steps.

i. p already specified as premise

ii. $p \rightarrow \neg q$ already specified as premise

iii. $\neg q \rightarrow \neg r$ already specified as premise

iv. $p \rightarrow \neg r$ *by law of syllogism using 2 and 3 above*

v. $\neg r$ *by modus ponnens rule, using 1 and 4*

d.Rule of Conjunction: This rule states that if p and q are individually true statements, then the composite p$\wedge q$ *is a true statements, i.e*

$$[p \wedge q] \rightarrow p \wedge q$$

or

p

q

$p \wedge q$ *(p and q can be wriiten as* $p \wedge q$*)*

e. Rule of Disjunctive Syllogism: It id defined as a logical implication given as follows.

$[(p \vee q) \wedge \neg p] \rightarrow q$

Or

$p \vee q$

$\neg p$

---

q ($p \vee q$ and negation p can be written as q)

f. Rule of Contradiction: it is defined as a logical implication,

$(\neg p \rightarrow Cp) \rightarrow p$

Or

$\neg p \rightarrow Cp$

---

p (it implies always p)

g. Rule of Conjunctive Simplification: This rule states that conjuction of p and q logically implies p, i.e

$(p \wedge q) \rightarrow p$

Or

$p \wedge q$

$$\frac{}{p \text{ (it implies } p)}$$

h. **Rule of Disjunctive Amplification: This rule states that** $p \vee q$ **can be inferred from p, and** $p \vee q$ **is logical consequence of p, it is expressed as**

$$p \rightarrow (p \wedge q)$$

this above can be expressed as,

$$\frac{p}{p \vee q}$$

i. **Rule of End elimination: This rule infers p from the wff** $p \wedge q$, *i.e*

$$\frac{(p \wedge q}{p}$$

j. **Rule of proof cases: it is stated in tabular form**

$$\frac{\begin{array}{l} p \rightarrow r \\ q \rightarrow r \end{array}}{(p \vee q) \rightarrow r}$$

Example: *Prove or disprove the following arguments:*

"if the auditorium was not available or there were examinations, then the music programme was postponed. If the music programme gets postponed, then a new date was announced. No new date was announced. Therefore, auditorium was available."

Solution: *Let us assume that following are symbols for the statements (propositions) in the above argument.*

P= auditorium was available

Q= there were examination

R= music programme was postponed

S= new date was announced

The statements can be expressed in the form of logical expressions given as follows.

$$(\neg p \vee q) \rightarrow r$$

$$r \rightarrow s$$

$$\neg s$$

---

p (all 3 premise can be written as p)

the logical implication for the above expression can be expressed as follows:

$$(((\neg p \vee q) \rightarrow r) \wedge (r \rightarrow s) \wedge \neg s) \rightarrow p$$

Validity of the above arguments can be proved as follows.

Steps	**Inference**	**Justification**
1	$(\neg p \vee q) \rightarrow r)$	already specified as premise
2	$r \rightarrow s$	already specified as premise
3	$\neg s$	already specified as premise
4	$\neg r$	*by mobus tollens using 2,3*
5	$\neg(\neg p \vee q)$	*by 1 and 4 and modus tollens.*
6	$(p \wedge \neg q)$	*by 5, and de morgans's rule*
7	p	*by 6, and rule of end elimination.*

Thus, it is proved through various inference rules, which logically follow the premises, the argument as a whole is valid statement.

5 . PREDICATE LOGIC

Basic idea: The word "**predicate**" means to declare or affirm concerning the subject of a preposition. **For example**, in the sentence, "**He was a king**", "**king**" is a predicate noun. Let us consider the following two statements represented as proposition.

P= rama is a student, and

Q= Krishna is a student.

Here, symbols p and q do not show anything common between them. However, the phrase "**is a student**" is Predicate, common in both sentences. In predicate logic these statements can be written as

(**First Level predicate**)

isstudent(Rama) and

isstudent(Krishna)

(Second Level Predicate)

student(Rama)

student(Krishna)

(Third Level Predicate)

s*(Rama),*

s*(Krishna),*

in addition it can be represented by

s(R),

s(K),

the above representations in predicate form shows that there is some common feature is s(R)and s(K), because both have common predicate, i.e student. If all the students in a class are to be represented using this form, we use a variable for student name. Therefore, the statements, "x is a student" can be represented in predicate form as s(x).

Predicate Formula: A general form of predicate statement is,

$$P(a_1, a_2, \ldots \ldots\ldots\ldots\ldots\ldots, a_n)$$

Where p is a predicate and $a_1, a_2, \ldots\ldots\ldots\ldots\ldots, a_n$ are terms. The predicate $p(a_1, a_2, \ldots, a_n)$ is called atomic formula. A well formed formula (wff) defined in propositional calculus is also applicable in predicate calculus

For all	$\forall$
There exists	$\exists$
Implies	$\rightarrow$
Not	$\neg$
Or	$\vee$
And	$\wedge$

Connectives can be used in the predicate similar to those in propositions.

Let us consider the sentences given below.

"Rama is a student ***and*** Rama plays cricket".

"Rama is a student ***or*** Rama plays cricket".

"Rama is a student ***implies that*** Rama plays cricket".

"Rama is ***not*** student".

These can be represented in the predicate forms in the same order as:

s(R) ∧*p(R,C)*

s(R) ∨*p(R,C)*

s(R) →*p(R,C)*

¬ s(R)

In the above predicates, p(R,C) stands for " Rama plays Cricket", where p is predicate for "plays", R for "Rama" is a subject and C for Cricket" is an object. P(R,C) is a two place predicate. Higher place predicates are also possible. Following are some of examples.

Rajan plays cricket and basketball = p(R,C,B).

Functions:

The parameters a1, a2,….., an in a predicate p, given below, can be constants or variables or functions.

$$\mathbf{P(a_1,a_2,\ldots\ldots,a_n)}$$

Consider the following sentences:

"Rajan is father of Rohit."

"Sheela is mother of Rohit."

"Rajan and Sheela are spouse."

Let the expressions – *fatherof(Rohit)*, and *motherof(Rohit)*, be functions and their values are *"Rajan" and "Sheela"* respectively. Using above expressions, the predicate.

Spouse(Rajan, Sheela),

Can be written as

spouse (fatherof(Rohit*), motherof*(Rohit)).

A function may have any number of objects, called arity of the functions. For example, if Rohit and Rajini are brother-sisters, then the functions.

Father of Rajni and Rohit, and

Mother of Rajni and Rohit,

Can be written as,

fatherof (Rajni, Rohit)=Rajan

motherof (Rajni, Rohit)=Sheela.

Example:

"2 plus 2 is 4." Can be written as function formula as ***plus*(2, 2)=4**

"50 divided by 10 is 5." Can be written as function formula as ***divided by*(50, 10)=5**

REPRESENTING SIMPLE FACTS IN LOGIC:

We briefly mentioned how logic can be used to represent simple facts in the last lecture. Here we will highlight major principles involved in knowledge representation. In particular predicate logic will be met in other knowledge representation schemes and reasoning methods. Symbols used the following standard logic symbols we use in this course are:

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an AI system might need. Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists. Suppose we want to represent the obvious fact stated by the classical sentence.

It is raining.

RAINING

It is sunny

SUNNY

It is raining, then it is not sunny.

RAINING $\rightarrow \neg$ SUNNY

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences.

1. Marcus was a man
2. Marcus was a pompeian.
3. All Pompeians were romans
4. Caesar was a ruler.
5. All romans were either loyal to caesar or hated him.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.

   Man(Marcus)

   This representation captures the critical fact of marcus being a man. It fails to capture some of the information in the english sentence, namely the notion of past tense.
2. Marcus was a Pompeian

   Pompeian(marcus)
3. All Pompeians were romans.

   $\forall$x: pompeians(x) $\rightarrow$ *Roman(x)*

4. Caesar was a ruler.

   ruler(Casear)

   Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name being referred to in a particular statement may require a fair amount of knowledge and reasoning.

5. All romans were either loyal to caesar or hated him.

$$\forall x{:}Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$$

In English the word "or" sometimes means the logical inclusive or and sometimes means the logical exclusive or (XOR). Here we have used the inclusive interpretation. Some people argue however that this English sentence is really stating an, exclusive or. To express that, we would have to write.

$$\forall x{:}\ roman(x) \rightarrow [(loyal\ to(x, Caesar) \vee hate(x, Caesar)) \wedge \neg(loyalto(x, Caesar) \wedge hate(x, Caesar))]$$

<u>Variable and Quantifiers</u>

To generalize the statement, "rama is student", it is written as "x is student", i.e s(X). if s(X) is true for a single case, then we say that the expression is satisfied.

Let us consider the following statements.

"x is human implies x is mortal."

"Socrates is human."

When represented in predicate form, these become:

$h(x) \rightarrow$ ***m(x), and***

h(S)

the above two wffs have some resemblance to the premises required for the inference rule of modus ponens. To generalize the implication, the variable x applicable for the entire human domain is quantified using quantying operator $\forall$ called universal quantifier. Above statements can be modified as follows after incorporating the effect of quantifiers.

"for all x, x is human implies that x is mortal", and

"Socrates is human"

Now, these are rewritten in symbolic form using quantifying operator $\forall$

$\forall x\ (h(x) \rightarrow$ ***m(x)), and***

h(S)

in this case the first statement ∀*x (h(x) →**m(x)) is true, when the statement is found to be true*** for entire range of x. because it says "for all x" or "for every x" or "for all possible values of x", h(x) →m(x) is true. Still, it is not possible to infer m(S), i.e "mortal Socrates". Because the statements still do not appear in the form such that the rule of modus ponens can be applied. The inference rule of universal instantiation, discussed in the next section, will help in resolving this problem.

SYNTAX AND SEMANTICS of FOL:

Terms: First order logic has sentence, but it also has terms which represents objects. Constant symbols, variables and function symbols are used to build terms and quantifiers and predicate symbols are used to build sentences.

SYNTAX of FOL in BNF (Backus – Naus Form):

Sentence → Atomic sentence
|Sentence connective sentence
|Quantifier variable,….sentence
|7 sentence
|(sentence).

Atomic sentence→ Predicate (Term,…)

Term → Term

Term → Function (Term…)
|constant
|variable

Connective → =>|A|V|⇔

Quantifier → ∀ | ∃

Constant → *A*| λcon| john|

<u>QUANTIFIERS:</u>

Quantifiers are used to express properties of entire collection of objects, rather than represent the object by names. FOL contains two standard quantifiers,

Universal quantifier (∀)

Existential quantifier (∃)

UNIVERSAL QUANTIFIERS:

General Notation: "$\forall X$P" where,

P – Logical expression, X – Variable, $\forall$ - For all

That is, P is true for all objects X in the Universe.

Examples:

All cats are mammals => $\forall$ X cat (X) → mammals(X)

That is, all the cats in the universe belongs to the type of mammals and hence the variable X may be replaced by any of the cat name (object, Name)

Examples:

Spot is a cat

Spot is a mammal

Cat (spot)

Mammal (spot)

Cat (spot) → mammal (spot)

Spot – Name of the cat.

Existential Quantifiers:

General Notification: $\exists$ X P, where

P – Logical Expression, X – Variable, $\exists$ - There exist

That is P is true for some object X in the universe.

Example:

Spot has a sister who is a cat.

$\exists$ X sister $(X, spot)$ → cat(X)

That is, the spot's sister is a cat, implies spot is also a cat and hence X may be replaced by, sister of spot, if it exists.

Example:

Felix is a cat.

Felix is a sister of spot

Cat (Felix)

Sister (Felix, spot)

Sister (Felix, spot) → cat (Felix).

NESTED QUANTIFIERS:

The sentences are represented using multiple quantifiers.

Example:

For all X and all Y, if x is the parent of Y then Y is the child of X.

∀X, Y parent (X, Y) → child (Y, X).

Everybody loves somebody

∀ *X* ∃ Y loves (X, Y)

There is someone who is loved by everyone

∃Y∀X loves (X, Y)

Connection between ∀ and ∃

The two Quantifiers (∀and ∃) are actually connected with each other through negation.

Example:

Everyone likes ice cream

∀X likes (X, ice cream) is equivalent to 7 ∃ X 7 likes (X, ice cream)

That is there is no one who does not like ice cream.

Ground Term or Clause: A term with no variable is called ground term.

Eg: cat (spot)

The De Morgan rules for quantified and unquantified sentences are as follows,

 # Quantified sentences:

∀X7P ≡ 7∃XP

7∀XP ≡ ∃X7P

∀XP ≡ 7∃ X 7P

∃XP ≡ 7∀ X 7P

#Unquantified sentence:

7(PAQ) ≡ 7P ∨ 7Q

7P A 7Q ≡ 7(P∨Q)

P A Q ≡ 7(7P ∨ 7Q)

P∨ Q ≡ 7(7P A 7Q)

Inference rules : All the inference rules applicable for propositional logic also apply for predicate logic. However, due to introduction of quantifiers, additional inference rules are there for the expressions using quantifiers. These are given below.

1. **Rule of universal instantiation**

   This rules states that if a universally quantified variable in a valid sentence is replaced by a term from the domain, then also the sentence is true. Thus is, if

   $$\forall x\ (h(x) \rightarrow m(x))$$

   Is true, and if x is replaced by "Socrates". And quantifier is removed then the statement,

   $$(h(S) \rightarrow m(S))$$

   Is still true. This rule is called as rule of universal instantiation and expressed as

   $$\frac{\forall x\ \ P(x)}{P(a)}$$

   (for all x p(x), can be written as P(a))

2. **Rule of universal generalization**

   If a statement p(a) is true for each element a of universe, then the universal quanitifier may be prefixed, and $\forall x\ P(x)$ *can be inferred from p(a), i.e.*

   $$\frac{P(a),\ \text{for all } a \in U}{\forall x\ \ P(x)}$$

   ((p(a) can be written as for all x, p(x))

3. **Rule of Existential Instantiation**

   If $\exists$x p(x) is true, and there is an element a in the universe of p, then we can infer p(a). i.e

   $$\frac{\exists x\ p(x)}{P(a)\ \text{for some } a \in U}$$

   (There existx p(x), can be written as P(a))

4. **Rule of Existential Generalization**

   If p(a) is true for some a in the universe, then it can be inferred that $\exists$x p(x) is true, i.e

   $$\frac{P(a),\ \text{for all } a \in U}{\exists x\ P(x)}$$

   ((p(a) can be written as there exist x, p(x))

<u>Example Problems:</u>

Prove the validity of the following statements

"All kings are men"..... (1)

"All men are falliable".....*(2)*

"Therefore, all kings falliable"

Solution:

The above statements could be rewritten without affecting the meanings they convey

"For all x, x is king implies that x is man"

"For all y, y is man implies that y is falliable."

"Therefore, for all z, z is king implies that z is falliable."

The above statements can be represented in the predicate form as follows.

$\forall x\ (k(x) \rightarrow m(x))$.....*from (1)*

$\forall y\ (m(y) \rightarrow f(y))$.....*from(2)*

$\forall z\ (k(z) \rightarrow f(z))$

We can arrive to formal proof for the above using following steps.

STEPS	Justification
1. $\forall x\ (k(x) \rightarrow m(x))$	*Given the premise*
2. $k(a) \rightarrow m(a)$	*By rule of universal instantiation on (1)*
3. $\forall y\ (m(y) \rightarrow f(y))$	*Given the premise*
4. $m(a) \rightarrow f(a)$	*By rule of universal instantiation on (3)*
5. $k(a) \rightarrow f(a)$	*By rule of syllogism using (2) and (4)*
6. $\forall z\ (k(z) \rightarrow f(z))$	*By rule of universal generalization on (5)*

Hence its proved

6. UNIFICATION

Basic idea: The require findings substitutions that make different logical expression. This provers is called unification and is a key component of all first order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists:

Syntax: Unify (P,Q) = θ where SUBSET (θ,P)= SUBSET (θ, q)

Here are the results of unification with four different sentences that might be in knowledge base.

UNIFY (knows (john,x), knows (john,jane)) = {x|jane}.....(1)

UNIFY (knows (john,x), knows (y,bill)) = {x|bil,y|john}.....(2)

UNIFY (knows (john,x), knows (y,mother(y))) = {y|john,x|mother(john)}……(3)

UNIFY (knows (john,x), knows (x,Elizabeth)) = fail ..... (4)

The last unification fails because x cannot take on the values john and Elizabeth at the sametime. Now remember that knows (x,Elizabeth) means "everyone knows Elizabeth", we should be able to infer that john knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. the problem can be avoided by standardizing apart one of the two sentences being unified, which means remaining its variable to avoid name clashes.

p	**q**	**θ**
Knows(John,x)	Knows(John,Jane)	{x/Jane}}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}}
Knows(John,x)	Knows(x,OJ)	{fail}

- Standardizing apart eliminates overlap of variables, e.g., Knows(z_{17},OJ)
  To unify *Knows(John,x)* and *Knows(y,z)*,
  θ = {y/John, x/z } or θ = {y/John, x/John, z/John}
- The first unifier is more general than the second.
  - There is a single most general unifier (MGU) that is unique up to renaming of variables.MGU = { y/John, x/z }

Unification algorithm:

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
   inputs: x, a variable, constant, list, or compound
           y, a variable, constant, list, or compound
           θ, the substitution built up so far

   if θ = failure then return failure
   else if x = y then return θ
   else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
   else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
   else if COMPOUND?(x) and COMPOUND?(y) then
       return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
   else if LIST?(x) and LIST?(y) then
       return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
   else return failure
```

1. 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.
2. In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.

For example: man(John) and ¬man(John) is a contradiction

While

man(John) and ¬man(Spot) is not

thus in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straight forward recursive procedure, called the unification algorithm that does just in simple way.

The basic idea of unification is very simple.: To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified regardless of their arguments. For example, the two literals.

Try assassinate(Marcus, Caesar)

hate(Marcus, Caesar)

Cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively.

RULES:

The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain any instances of the variable being matched.

Example:

P(x,x)..... (1)

P(y,z) .....(2)

The two instances of P match fine. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write that substitution as **x=y in (1)**

y/.x

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions). But now, if we simply continue and match x and z, we produce the substitution z/x. but we cannot substitute both y and z for x, so we have not produced a consistent substitution.

What we can need to do after finding the first substitution y/x is to make that substitution y/x is to make that substitution throughout the literals, giving

P(y,y)

P(y,z)

Now we can attempt to unify arguments y and z, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as **Y=z and x=y** unifications pass.

(z/y) (y/x)

Following standard notation for function compostion. In general the substitution (a1/a2, a3/a4,..) (b1/b2,b3/b4…)…means to apply all the substitutions of the right most list, then take the result and apply all the ones of the ones of the next list, and so forth, until all substitutions have been applied.

The object of unification procedure is to discover at least one substitution that causes two literals to match.

For example: the literals

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,Caesar/y,Caesar/z)

(marcus/x,polonius/y,polonius/z)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match.

ALGORITHM:

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 L1 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 L20
5. For I←1 to number of arguments inL1:
    a. Call unify with the /th argument of L1 and the i th argument of L2, putting result in S.
    b. If S contains FAIL then return {FAIL}.
    c. If S is not equal to NIL then;
        - Apply S to the remainder of both L1 and L2.
        - SUBST := APPEND(S, SUBST)
6. Return SUBST.

7. WHAT IS RESOLUTION DISCUSS BRIEFLY IN PROPOSITIONAL AND PREDICATE LOGIC?

Basic idea: Resolution produces proofs by refutation. In others words, to prove a statement resolution attempts to show that negation of the statements produces a contradiction with the known statements (i.e that is unsatisfiable).

The basis of resolution in propositional logic:

The resolution procedure is a iterative process at each step, two clauses called parent clauses are compared (resolved), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other.

Suppose there are two literals in the system

Winter ∨ *summer*

¬**Winter** ∨ *cold*

Recall that this means that both clauses must be true(i.e., the clauses, although they look independent, are really cojoined).

- We can observe precisely one of winter and ¬winter will be true at any point.
- If winter is true , then cold must be true to guarantee the truth of the second clause.
- If ¬winter is true, then summer must be true to gurantee the truth of the first clause.

- Thus we see that from these two clauses we can deduce

Summer ∨ *cold*

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example , winter.

If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses Winter

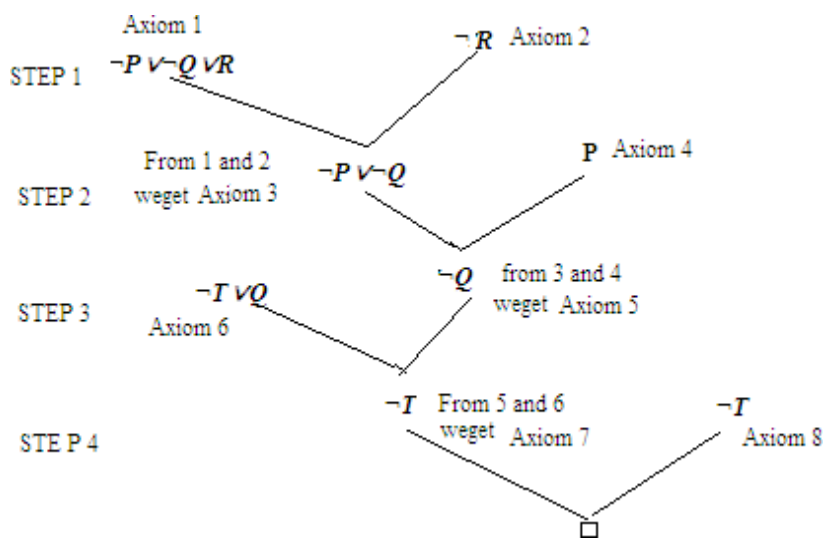
¬*W*inter

Will produce an empty clause.

1. If contradiction exist then eventually it will be found.
2. If no contradiction exists, it is possible that the procedure will never terminate.

Note: a→*b is equivalent to ¬a∨b (convert to clause form)*

Example:	**Given axioms**	**Converted to clause form**	**steps**
	P	**P**	**1**
	(P∧*Q*) →*R*	***¬P∨¬Q∨R***	***2***
	(S∨T) →Q	***¬S∨¬Q***	***3***
		¬T∨Q	***4***
	T	***T***	***5***



Step1: axiom1 and axiom2 are conjoined.

Step2: *R and ¬R are true finally we get Axiom3.*

Step3: Axiom 3 and Axiom 4 are conjoined where P and ¬P are found to be true, finally weget axiom5

Step 4: from Axiom 5 and Axiom6 weget axiom 7 where Q and ¬Q are found to be true.

Step 5: Contradiction found.

Resolution in Predicate Logic:

In order to use resolution for expressions in predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause.

For example:

Suppose we want to resolve two clauses;

1. *Man(Marcus)*
2. *¬Man(x_1) ∨ mortal (x_1)*

- The literal man/Marcus can be unified with the literal man{x\} with substituting **x_1=marcus** i.e Marcus/x_1.
- While substituting **x_1=marcus**, *¬Man(Marcus) is false. Where we cannot simply cancel out the two man literals as we did in propositional logic and generate the resolvent mortal(x_1)*
- *Clause 2 says that for a given x1, either ¬Man(x_1) or mortal (x_1). So for it to be true, we can now conclude only that mortal(Marcus) must be true.*

Algorithm: resolution

Step1: convert all the propositions of F to clause form.

Step2: Negate P & convert the result to clause form. Add it to set of clauses obtained in step1.

Step3: Repeat until either a contradiction is found or no progress can be made.

a. Select two clauses. Call the parent clauses.
b. Resolve them together. The resulting clause, called the resolvent.

   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 ¬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.

c. If the resolvent is empty clause then a contradiction has been found. , then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

8. WEAK SLOT FILLER STRUCTURES

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.

So called because:

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

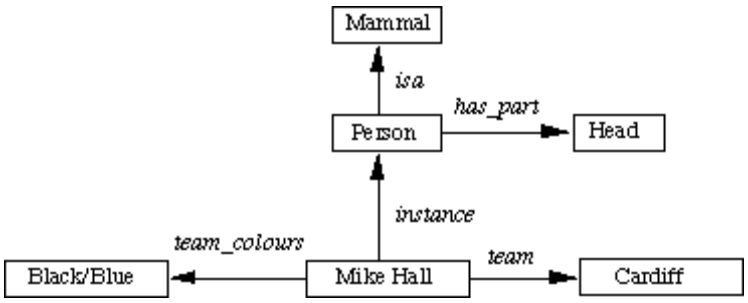
We will study two types:

- Semantic Nets.
- Frames.

Semantic Nets

The major idea is that:

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



Intersection search

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.

Representing Non binary predicates

Semantic nets are a natural way to represent relationships that would appear as grpund instances of binary predicates in predicate logic. For example some of the arcs from the above figure, could be represented as logic as

isa(Person, Mammal)

instance (Pee-Wee-Reese, Person)

team(Pee-Wee-Reese, Brooklyn-Dodgers)

uniform-color (Pee-Wee-Reese, Blue)

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

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

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

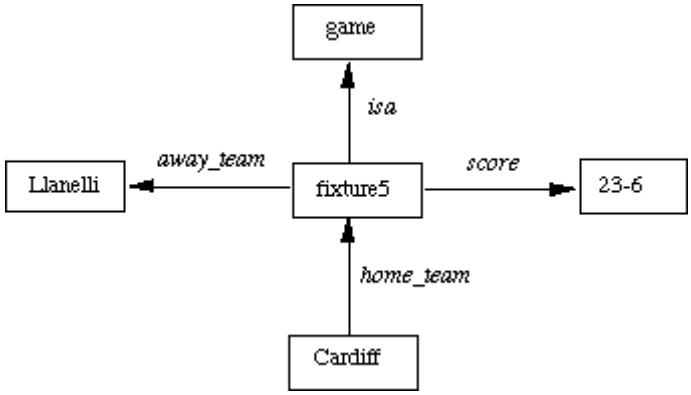


Fig. A Semantic Network for *n*-Place Predicate

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

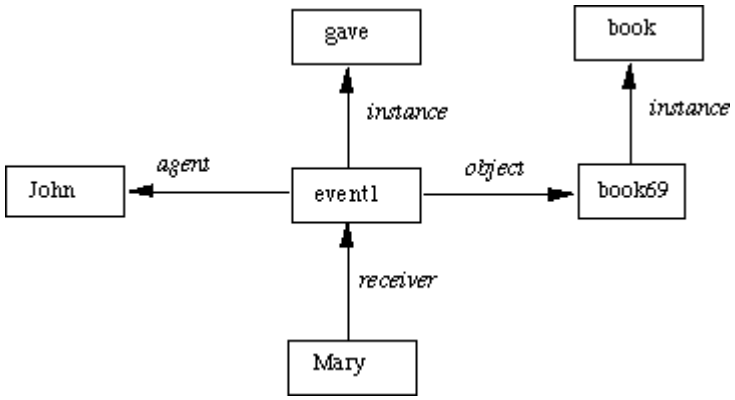


Fig. A Semantic Network for a Sentence

Making some important distinctions

Basic inference mechanism: *follow links between nodes*.

Two methods to do this:

Intersection search

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

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

Inheritance

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

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

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

in the following Semantic net:

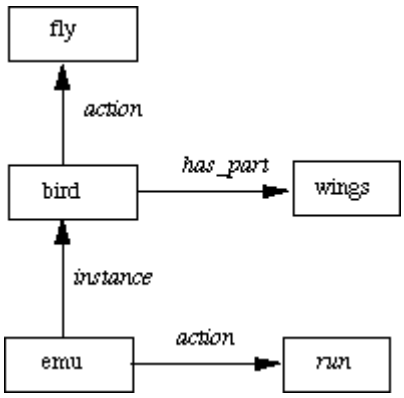


Fig. A Semantic Network for a Default Reasoning

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

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

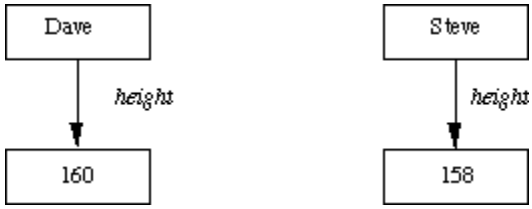


Fig. Two heights

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

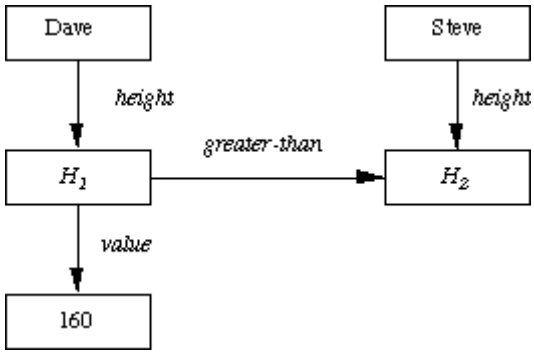


Fig. Comparison of two heights

Partitioned Semantic Nets:

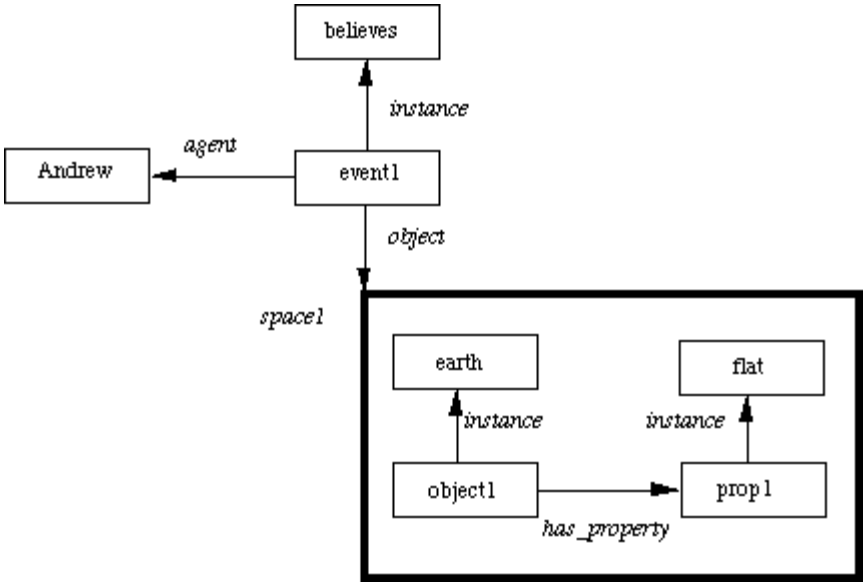
Partitioned Networks *Partitioned* Semantic Networks allow for:

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

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

Consider the following: *Andrew believes that the earth is flat.* We can encode the proposition *the earth is flat* in a *space* and within it have nodes and arcs the represent the fact (Fig. 15). We can

the have nodes and arcs to link this *space* the the rest of the network to represent Andrew's



belief.

Fig. Partitioned network

The Evolution into frames:

- The idea of a semantic net started out simply as a way to represent labeled connections among entities.
- But as we have just seen, as we expand the range of problem solving tasks that the representation must support, the representation itself necessarily begin to become more complex.
- In particular, it becomes useful to assign more structure to nodes as well as to links.

<u>Frames</u>

A frame is a collection of attribute (usually called slots) and associated values (and possibly constraints on values) that describes some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view (as it did in the vision system proposal in which the term frame was first introduced).

A single frame taken alone is rarely useful, instead we build frame systems out collections of frames that are connected to each other by virtue of the fact that the value of an explore ways that frame systems can be used to encode knowledge and support reasoning.

Frames as Sets and Instances:

Frames can also be regarded as an extension to Semantic nets. Indeed it is not clear where the distinction between a semantic net and a frame ends. Semantic nets initially we used to represent

labelled connections between objects. As tasks became more complex the representation needs to be more structured. The more structured the system it becomes more beneficial to use frames. A *frame* is a collection of attributes or slots and associated values that describe some real world entity. Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning. Set theory provides a good basis for understanding frame systems. Each frame represents.

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

Consider the example first discussed in Semantics Nets

Person

Isa : *Mammal*

Cardinality : . . .

Adult-Male

Isa : *Person*

Cardinality : . . .

Rugby-Player

Isa : *Adult-Male*

Cardinality : . . .

Height :

Weight :

Position :

Team :

Team-Colours :

Back

Isa : *Rugby-Player*

Cardinality : . . .

Tries :

Mike-Hall

Instance : *Back*

Height : *6-0*

Position : *Centre*

Team : *Cardiff-RFC*

Team-Colours : *Black/Blue*

Rugby-Team

Isa : *Team*

Cardinality : . . .

Team-size : *15*

Coach :

Cardiff-RFC

instance: *Rugby-Team*

Team-size: *15*

Coach: *Terry Holmes*

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:

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

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

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

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

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

Figure: A Metaclass frame system

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

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

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

Slots as Objects

How can we to represent the following properties in frames?

- Attributes such as *weight*, *age* be attached and make sense.
- Constraints on values such as *age* being less than a hundred
- Default values
- Rules for inheritance of values such as children inheriting parent's names
- Rules for computing values
- Many values for a slot.

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

NOTE the following:

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

Interpreting frames

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

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

Algorithm: Property Inheritance

To retrieve a value V for slot S of an instance F do.

1. Set CANDIDATES to empty.
2. Do breadth first or depth first search up the isa hierarchy from F, following all instance and isa links. At each step, see if a value for S or one f its generalizations is stored.
    a. If a value is found, add it to CANDIDATES and terminate that branch of the search.
    b. If no value is found but there are instance or isa links upward, follow them.
    c. Otherwise, terminate the branch.
3. for each element C of CANDIDATES do:
    a. see if there is any other element of CANDIDATES that was derived from a class closer to F than the class from which C came.
    b. If there is, then, remove C from CANDIDATES.
4. check the cardinality of CANDIDATES:
    a. if it is 0, then report that no value was found.
    b. If it is 1, then return the single element of CANDIDATES as V.
    c. If it is greater than 1, report a contradiction.

Frame Languages:

The idea of a frame system as a way to represent declarative knowledge has been encapsulated in a series of frame oriented knowledge representation languages, whose features have evolved and been driven by an increased understating of the sort of representation issues.

Example: KRL, FRL, RLL, KL-ONE, Brachman and Schmolze, KRYPTON,NIKL.

9. STRONG SLOT FILLER STRUCTURES

Introduction

Strong Slot and Filler Structures typically*:*

- Represent links between objects according to more **rigid** rules.
- Specific notions of what types of object and relations between them are provided.
- Represent knowledge about common situations.

Conceptual Dependency (CD)

Conceptual Dependency originally developed to represent knowledge acquired from natural language input.

The goals of this theory are:

- To help in the drawing of inference from sentences.
- To be independent of the words used in the original input.
- That is to say: *For any 2 (or more) sentences that are identical in meaning there should be only one representation of that meaning.*

It has been used by many programs that portend to understand English (*MARGIE, SAM, PAM*). CD developed by Schank *et al.* as were the previous examples.

CD provides:

- a structure into which nodes representing information can be placed
- a specific set of primitives
- at a given level of granularity.

Sentences are represented as a series of diagrams depicting actions using both abstract and real physical situations.

- The agent and the objects are represented
- The actions are built up from a set of primitive acts which can be modified by tense.

Examples of Primitive Acts are:

ATRANS

-- Transfer of an abstract relationship. *e.g. give*.

PTRANS

-- Transfer of the physical location of an object. *e.g. go.*

PROPEL

-- Application of a physical force to an object. *e.g. push*.

MTRANS

-- Transfer of mental information. *e.g. tell.*

MBUILD

-- Construct new information from old. *e.g. decide*.

SPEAK

-- Utter a sound. *e.g. say*.

ATTEND

-- Focus a sense on a stimulus. *e.g. listen, watch*.

MOVE

-- Movement of a body part by owner. *e.g. punch, kick*.

GRASP

-- Actor grasping an object. *e.g. clutch.*

INGEST

-- Actor ingesting an object. *e.g. eat*.

EXPEL

-- Actor getting rid of an object from body. *e.g. ????*.

Six primitive conceptual categories provide *building blocks* which are the set of allowable dependencies in the concepts in a sentence:

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.

Schank *et al.* developed all of the above.

Scripts

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 specialised roles.

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 components of a script include:

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

-- Slots representing objects involved in events.

Roles

-- Persons involved in the 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.

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*
- Get away 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.

There are 3 scenes: obtaining the gun, robbing the bank and the getaway.

Some additional points to note on Scripts:

- If a particular script is to be applied it must be activated and the activating depends on its significance.
- If a topic is mentioned in passing then a pointer to that script could be held.
- If the topic is important then the script should be opened.
- The danger lies in having too many active scripts much as one might have too many windows open on the screen or too many recursive calls in a program.
- Provided events follow a known trail we can use scripts to represent the actions involved and use them to answer detailed questions.
- Different trails may be allowed for different outcomes of Scripts (*e.g.* The bank robbery goes wrong).

The full Script could be described in the following figure

E

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	

Fig. Simplified Bank Robbing Script

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

What is CYC?

- An ambitious attempt to form a very large knowledge base aimed at capturing commonsense reasoning.

- Initial goals to capture knowledge from a hundred randomly selected articles in the En*CYC*lopedia Britannica.
- Both Implicit and Explicit knowledge encoded.
- Emphasis on study of underlying information (assumed by the authors but not needed to tell to the readers.

Example: Suppose we read that *Wellington learned of Napoleon's death*

Then we (humans) can conclude *Napoleon never new that Wellington had died*.

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.

Why build large knowledge bases:

Brittleness - Specialised 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. How is CYC coded?

- Special CYCL language:
  - LISP like.
  - Frame based
  - Multiple inheritance
  - Slots are fully fledged objects.
  - Generalised inheritance -- any link not just *isa* and *instance*.

UNIT III

Reasoning under uncertainty: Logics of non-monotonic reasoning-Implementation- Basic probability notation - Bayes rule – Certainty factors and rule based systems-Bayesian networks – Dempster - Shafer Theory - Fuzzy Logic

1. REASONING UNDER UNCERTAINTY

UNCERTAINTY

Let action At = leave for airport t minutes before flight

Will At get me there on time?

Problems:

1) partial observability (road state, other drivers' plans, etc.)

2) noisy sensors (KCBS tra_c reports)

3) uncertainty in action outcomes (at tire, etc.)

4) immense complexity of modelling and predicting tra_c

Hence a purely logical approach either

1) risks falsehood: A25 will get me there on time"

or 2) leads to conclusions that are too weak for decision making:

"A25 will get me there on time if there's no accident on the bridge and it doesn't rain and my tires remain intact etc etc."

Methods for handling uncertainty

Default or nonmonotonic logic:

Assume my car does not have a at tire

Assume A25 works unless contradicted by evidence

Issues: What assumptions are reasonable? How to handle contradiction?

Rules with fudge factors:

$$A_{25} \mapsto_{0.3} AtAirportOnTime$$
$$Sprinkler \mapsto_{0.99} WetGrass$$
$$WetGrass \mapsto_{0.7} Rain$$

Issues: Problems with combination, e.g., Sprinkler causes Rain??

2. NON-MONOTONIC REASONING .

A **non-monotonic logic** is a formal logic whose consequence relation is not monotonic. Most studied formal logics have a monotonic consequence relation, meaning that adding a formula to a theory never produces a reduction of its set of consequences. Intuitively, monotonicity indicates that learning a new piece of knowledge cannot reduce the set of what is known.

A monotonic logic cannot handle various reasoning tasks such as reasoning by default (consequences may be derived only because of lack of evidence of the contrary), abductive reasoning (consequences are only deduced as most likely explanations), some important approaches to reasoning about knowledge

Default reasoning

An example of a default assumption is that the typical bird flies. As a result, if a given animal is known to be a bird, and nothing else is known, it can be assumed to be able to fly. The default assumption must however be retracted if it is later learned that the considered animal is a penguin. This example shows that a logic that models default reasoning should not be monotonic.

Logics formalizing default reasoning can be roughly divided in two categories: logics able to deal with arbitrary default assumptions (default logic, defeasible logic/defeasible reasoning/argument (logic), and answer set programming) and logics that formalize the specific default assumption that facts that are not known to be true can be assumed false by default (closed world assumption and circumscription).

Abductive reasoning

Abductive reasoning is the process of deriving the most likely explanations of the known facts. An abductive logic should not be monotonic because the most likely explanations are not necessarily correct.

For example, the most likely explanation for seeing wet grass is that it rained; however, this explanation has to be retracted when learning that the real cause of the grass being wet was a sprinkler. Since the old explanation (it rained) is retracted because of the addition of a piece of knowledge (a sprinkler was active), any logic that models explanations is non-monotonic.

Reasoning about knowledge

If a logic includes formulae that mean that something is not known, this logic should not be monotonic. Indeed, learning something that was previously not known leads to the removal of

the formula specifying that this piece of knowledge is not known. This second change (a removal caused by an addition) violates the condition of monotonicity. A logic for reasoning about knowledge is the autoepistemic logic.

Belief revision

Belief revision is the process of changing beliefs to accommodate a new belief that might be inconsistent with the old ones. In the assumption that the new belief is correct, some of the old ones have to be retracted in order to maintain consistency. This retraction in response to an addition of a new belief makes any logic for belief revision to be non-monotonic. The belief revision approach is alternative to paraconsistent logics, which tolerate inconsistency rather than attempting to remove it.

3. IMPLEMENTATION ISSUES IN NON MONOTONIC REASONING:

The logical frameworks are not enough for implementing non monotonic reasoning in problem solving programs. These are the some weakness for logical systems. The four important problems that arise in real systems are us follows.

- The first is how to derive exactly those monotonic conclusions that are relevant to solving the problem at hand while not wasting not wasting time on those that, while they may be licensed by the logic.
- The second problem is how to update our knowledge incrementally as problem solving progresses.
- The third problem is that in nonmonotonic reasoning systems, it often happens that more than one interpretation of the known facts is licensed by the available inference rules.
- The final problem is that, in general, these theories are not computationally effective.

Implementation – Depth first search

The implementation of DFS is directed by

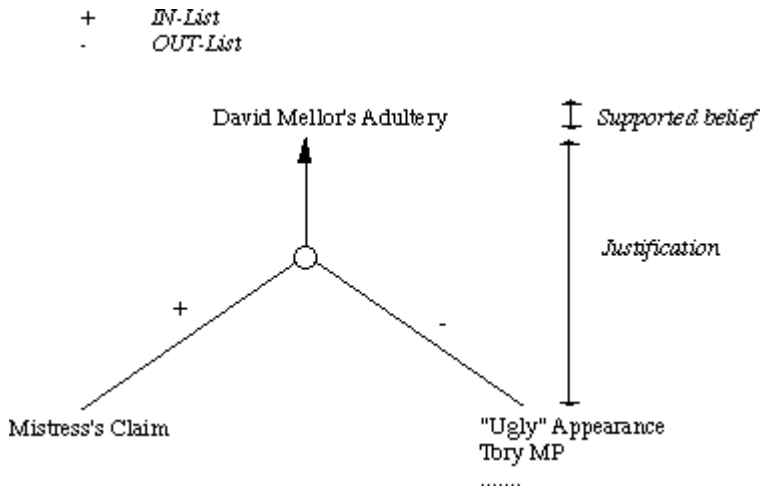
- Dependency directed Backtracking
- Justification based truth maintenance systems
- Logic based truth maintenance systems.

Dependency directed Backtracking:

The following scenarios are likely to occur often in non monotonic reasoning (DFS).

- Analyze the fact, F which cannot be derived monotonic from what we already know but which can be derived by making some assumption A which seems plausible. So we make assumption A, derive F, and then derive some additional facts G and H from F.
- We later derive some other facts M and N, but they are completely derive some additional facts G and H from F. We later derive some other facts M and N, but they are completely independent of A and F.
- A little while later, a new fact comes in that invalidates A. We need to rescind our proof of F, and also our proofs of G and H since they depended on F.
- But what about M and N? They didn't depend on F, so there is no logical need to invalidate them. But if we use a conventional backtracking scheme, we have to back up past conclusions in the order in which they derived them.
- To get around this problem, we need to backup past M and N, thus undoing them in order to get back to F,G,H and A. To get around this problem, we need a slightly different notion of backtracking, one that is based on logical dependencies rather than the chronological order in which decisions were made. We call this new method dependency directed backtracking.

Justification based truth maintenance systems



- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- 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 labeled 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.

The main key reasoning operations that a JTMS performed are as follows:

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

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 simultaneously. An LTMS would throw a contradiction here.
- If this happens network has to be reconstructed.

4. BASIC PROBABILITY NOTATION

An important goal for many problem solving systems is to collect evidence as the system goes along to modify its behavior on basis of evidence.

To model this behavior, we need a statistical theory of evidence. Bayesian theory is such a theory.

The fundamental notion of Bayesian statistics is than of conditional probability.

P(H\E) = the Probability that hypothesis H is true given evidence E.

P(E\H) = the Probability that we will observe evidence E given that hypothesis i is true.

P(H) = the priori probability that hypothesis i is true in absence of any specific evidence. These probabilities are called prior probabilities or priors.

K = the number of possible hypothesis.

Baye's Theorem then states that

$$P(H\backslash E) = \frac{P(E\backslash H)\, P(H)}{\Sigma_{n=1}\, P(E\backslash H_n)\, P(H_n)}$$

EXAMPLE:

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.

1. If we know the probability of each minerals in prior and its physical characteristics then we can use baye's formula to compute.
2. From based upon the evidence we collect how likely it is various minerals are present in particular place can be identified.

Key to use Baye's Theorem:

P(A\B)= conditional probability of A given we have only evidence of B.

Example: Solving Medical Diagnosis problem

S: Patient has Spots

M: Patient has Measles

F: Patient has High Fever

1. Presence of spots serves as evidence in favor of Measles. It also serves evidence for fever, measles cause fever.
2. Either spot or fever alone causes evidence in favor of measles.
3. If both are present we need to take both into account in determining total weight of evidence.

$$P(H\backslash E,e) = P(H\backslash E)\ \frac{P(e/E,H)}{P(e/E)}$$

BAYES' RULE

Product rule $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$

$\Rightarrow$ Bayes' rule $P(a|b) = \frac{P(b|a)P(a)}{P(b)}$

or in distribution form

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha \mathbf{P}(X|Y)\mathbf{P}(Y)$$

Useful for assessing diagnostic probability from causal probability:

$$P(Cause|Effect) = \frac{P(Effect|Cause)P(Cause)}{P(Effect)}$$

E.g., let M be meningitis, S be stiff neck:

$$P(m|s) = \frac{P(s|m)P(m)}{P(s)} = \frac{0.8 \times 0.0001}{0.1} = 0.0008$$

Note: posterior probability of meningitis still very small!

Probability

Given the available evidence, A25 will get me there on time with probability 0:04

(Fuzzy logic handles degree of truth NOT uncertainty e.g., WetGrass is true to degree 0:2)

Probabilistic assertions summarize effects of

laziness: failure to enumerate exceptions, qualifications, etc.

ignorance: lack of relevant facts, initial conditions, etc.

6. CERTAINTY FACTOR AND RULE BASED SYSTEM

1. It is one of practical way of compromising pure Bayesian system.
2. The approach we discuss was pioneered in the MYCIN system, which attempt to recommend appropriate therapies for patient with bacterial infections.
3. It interacts with the physician to acquire the clinical data it needs.
4. MYCIN is an example of an expert system since it normally done by a expert system.
5. MYCIN uses rules to reason clinical data from its goal of finding significant disease causing organisms.
6. Once it finds the organisms, it then attempt to select a therapy by which the disease (s) may be treated.

CERTAINTY FACTOR:

A Certainty factor (CF [h,e]) is defined in terms of 2 components

1. MB[h,e] --- a Measure of Belief (between 0 and 1) of belief in hypothesis h given the evidence e. MB measures the extent to which the evidence supports the hypothesis. It is zero if the evidence fails to support the hypothesis.
2. MD[h,e] --- a Measure of DisBelief (between 0 and 1) of Disbelief in hypothesis h given the evidence e. MD measures the extent to which the evidence supports the negation of the hypothesis. It is zero if the evidence supports the hypothesis.

From these two measures we can define the certainty factor

CF[h,e]= MB[h,e]-MD[h,e] →eq no 1

<u>Combining Uncertainty rules:</u>

1. MYCIN reflect the experts' assessments of the strength of evidence in support of hypothesis.
2. MYCIN reasons however there CF's need to be combined to reflect the operations of multiple pieces of evidence and multiple rules are applied to the problem.

   The measure of belief and disbelief is given by S1 and S2

MB(h, S1∧S2){ MB(h,S1) + MB(h, S2) [1−MB(h,S1) ……Equation no 2

MD(h, S1∧S2){ MD(h,S1) + MD(h, S2) [1−MD(h,S1)…….. Equation no 3

Suppose we make an initial observation that confirms our belief in h with **MB(h, S1)= 0.3**

MD[h, s1]=0, MB[h,s2]=0.2 and CF[h,s1]=0.3

Substituting these in **equation no 2** weget

MB(h, S1∧**S2)** = { 0.3 +0.2 (1-0.3)

= 0.3 + 0.2 (0.7)

= 0.44….....Equation no 4

MD(h, S1∧ **S2)= 0.0** .....Equation no 5

Substituting eq4 and eq5 in eq1 we get **CF[h,e]= MB[h,e]-MD[h,e]** = 0.44 -0

Certainty Factor (h,e)= 0.44..... equation no 6

The formula MYCIN uses for the MB of the conjunction and disjunction of 2 hypothesis.

By using Bayes theorem:

$$MB[h,e] = \begin{cases} \dfrac{\max[p(h/e), p(h)] - p(h)}{P(h/e)\ 1-p(h)} \end{cases}$$

From eq 4, 5 and 6 can be written as single rule rather that 3 separate rules.

MYCIN independence assumption can make moment 3 separate rules **CF each was 0.6**

MB [h, S1∧**S2] = 0.6+ 0.6 (1-0.6)**

= 0.6 + 0.6(0.4)

= 0.84

MB [h, (S1∧**S2)∧S3] = 0.84+ 0.6 (1-0.84)**

= 0.84 + 0.6(0.16)

= 0.936

Let us consider a concrete example

S: Sprinkler was on last night

W: grass is wet

R: it rained last night

We can write MYCIN rules that describe predictive relation among 3 events

a) If sprinkle was on last night then grass is wet in morning = evidence is 09%
b) If the grass is wet in morning then it is rained last night = 0.9-0.1=0.8%

MB(W,S) =0.8

MD(R,W)= 0.8* 0.9

= 0.72

6. BAYESIAN NETWORKS

A simple, graphical notation for conditional independence assertions and hence for compact specification of full joint distributions

Syntax:

a set of nodes, one per variable

a directed, acyclic graph (link _ \directly inuences")

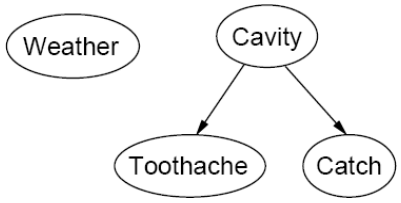
a conditional distribution for each node given its parents:

$$\mathbf{P}(X_i | Parents(X_i))$$

In the simplest case, conditional distribution represented as a conditional probability table (CPT) giving the distribution over Xi for each combination of parent values

Example

Topology of network encodes conditional independence assertions:



Weather is independent of the other variables

Toothache and Catch are conditionally independent given Cavity

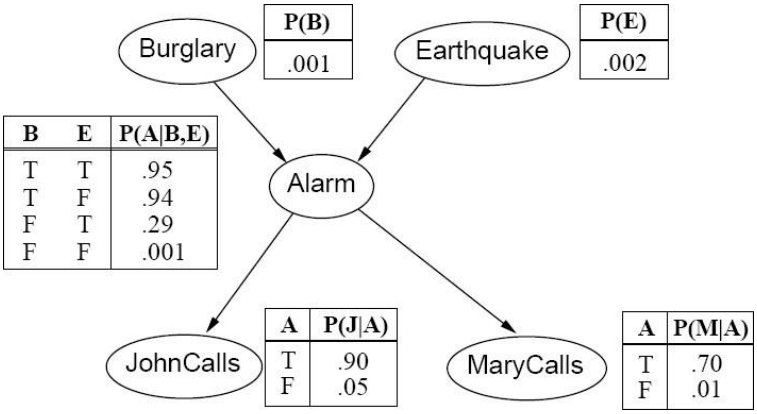
Example

I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call. Sometimes it's set of by minor earthquakes. Is there a burglar?

Variables: Burglar, Earthquake, Alarm, JohnCalls, MaryCalls

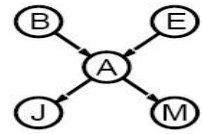
Network topology reacts \causal" knowledge:

- A burglar can set the alarm off
- An earthquake can set the alarm off
- The alarm can cause Mary to call
- The alarm can cause John to call



Compactness

A CPT for Boolean X_i with k Boolean parents has 2^k rows for the combinations of parent values



Each row requires one number p for $X_i = true$
(the number for $X_i = false$ is just $1 - p$)

If each variable has no more than k parents,
the complete network requires $O(n \cdot 2^k)$ numbers

I.e., grows linearly with n, vs. $O(2^n)$ for the full joint distribution

For burglary net, $1 + 1 + 4 + 2 + 2 = 10$ numbers (vs. $2^5 - 1 = 31$)

Constructing Bayesian networks

Need a method such that a series of locally testable assertions of conditional independence guarantees the required global semantics

1. Choose an ordering of variables $X_1, \ldots, X_n$
2. For $i = 1$ to n
   add X_i to the network
   select parents from $X_1, \ldots, X_{i-1}$ such that
   $\mathbf{P}(X_i | Parents(X_i)) = \mathbf{P}(X_i | X_1, \ldots, X_{i-1})$

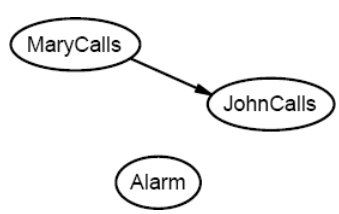
This choice of parents guarantees the global semantics:

$$\begin{aligned} \mathbf{P}(X_1, \ldots, X_n) &= \Pi_{i=1}^{n} \mathbf{P}(X_i | X_1, \ldots, X_{i-1}) \quad \text{(chain rule)} \\ &= \Pi_{i=1}^{n} \mathbf{P}(X_i | Parents(X_i)) \quad \text{(by construction)} \end{aligned}$$

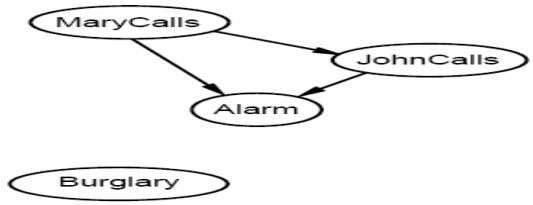
Example

Suppose we choose the ordering M, J, A, B, E

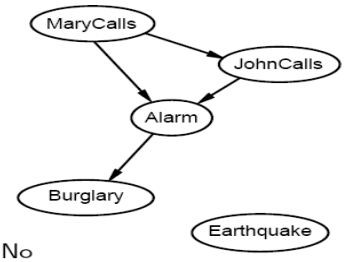
$P(J|M) = P(J)$?



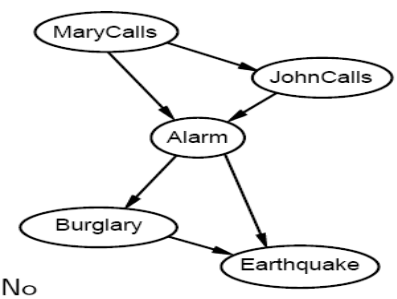
$P(J|M) = P(J)$? No
$P(A|J, M) = P(A|J)$? $P(A|J, M) = P(A)$?



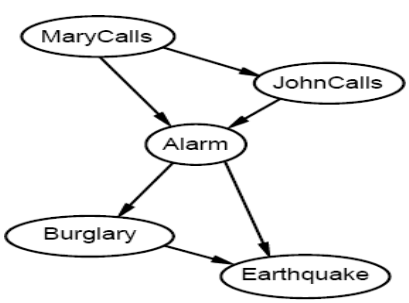
$P(J|M) = P(J)$? No
$P(A|J, M) = P(A|J)$? $P(A|J, M) = P(A)$? No
$P(B|A, J, M) = P(B|A)$?
$P(B|A, J, M) = P(B)$?



$P(J|M) = P(J)$? No
$P(A|J, M) = P(A|J)$? $P(A|J, M) = P(A)$? No
$P(B|A, J, M) = P(B|A)$? Yes
$P(B|A, J, M) = P(B)$? No
$P(E|B, A, J, M) = P(E|A)$?
$P(E|B, A, J, M) = P(E|A, B)$?



$P(J|M) = P(J)$? No
$P(A|J, M) = P(A|J)$? $P(A|J, M) = P(A)$? No
$P(B|A, J, M) = P(B|A)$? Yes
$P(B|A, J, M) = P(B)$? No
$P(E|B, A, J, M) = P(E|A)$? No
$P(E|B, A, J, M) = P(E|A, B)$? Yes



Deciding conditional independence is hard in noncausal directions (Causal models and conditional independence seem hardwired for humans!)

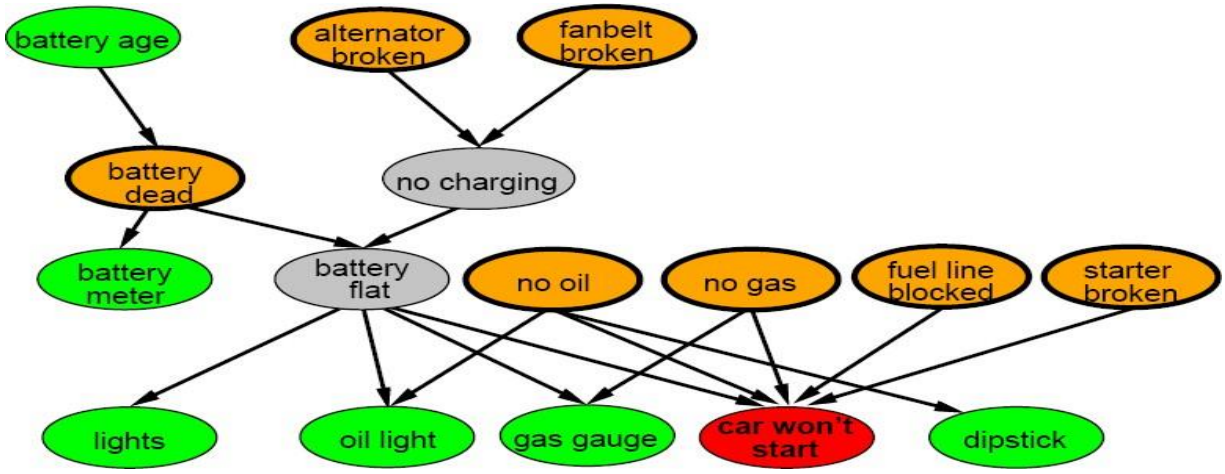
Assessing conditional probabilities is hard in noncausal directions Network is less compact: 1 + 2 + 4 + 2 + 4=13 numbers needed

Example: Car diagnosis

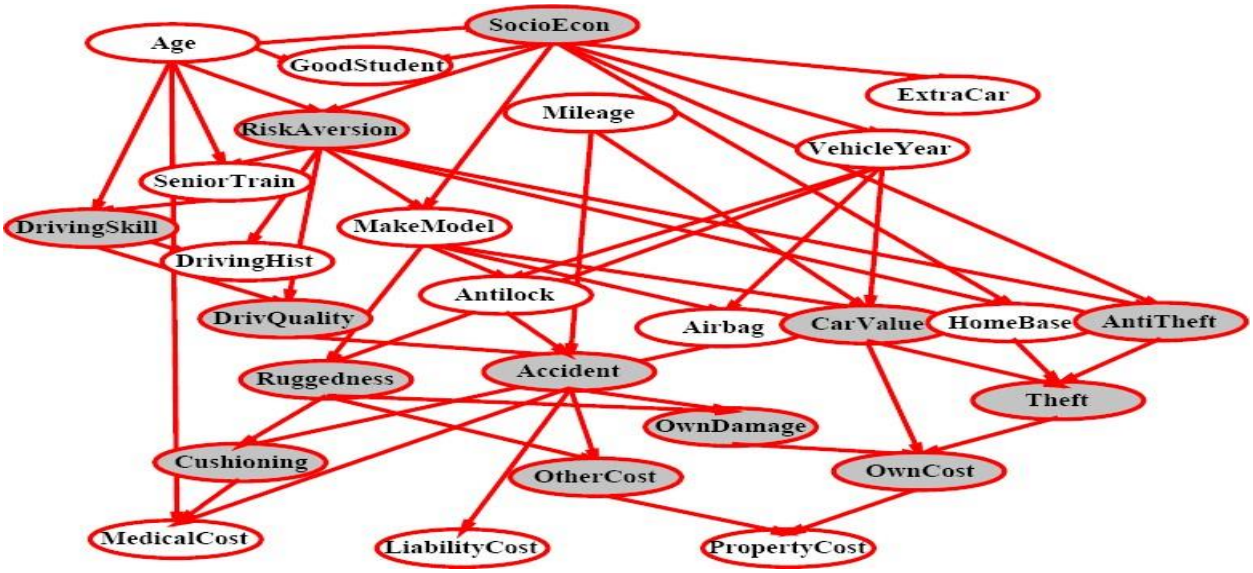
Initial evidence: car won't start

Testable variables (green), \broken, so _x it" variables (orange)

Hidden variables (gray) ensure sparse structure, reduce parameters



Example: Car insurance



<u>Inference in Bayesian networks Inference tasks</u>

Simple queries: compute posterior marginal $\mathbf{P}(X_i|\mathbf{E}=\mathbf{e})$
e.g., $P(NoGas|Gauge=empty, Lights=on, Starts=false)$

Conjunctive queries: $\mathbf{P}(X_i, X_j|\mathbf{E}=\mathbf{e}) = \mathbf{P}(X_i|\mathbf{E}=\mathbf{e})\mathbf{P}(X_j|X_i, \mathbf{E}=\mathbf{e})$

Optimal decisions: decision networks include utility information;
probabilistic inference required for $P(outcome|action, evidence)$

Value of information: which evidence to seek next?

Sensitivity analysis: which probability values are most critical?

Explanation: why do I need a new starter motor?

7. DEMPSTER - SHAFER

- The **Dempster–Shafer theory (DST)** is a mathematical theory of evidence. It allows one to combine evidence from different sources and arrive at a degree of belief (represented by a belief function) that takes into account all the available evidence. The theory was first developed by Arthur P. Dempster and Glenn Shafer.
- In a narrow sense, the term **Dempster–Shafer theory** refers to the original conception of the theory by Dempster and Shafer. However, it is more common to use the term in the wider sense of the same general approach, as adapted to specific kinds of situations. In particular, many authors have proposed different rules for combining evidence, often with a view to handling conflicts in evidence better.
- Dempster–Shafer theory is a generalization of the Bayesian theory of subjective probability; whereas the latter requires probabilities for each question of interest, belief functions base degrees of belief (or confidence, or trust) for one question on the probabilities for a related question.

These degrees of belief may or may not have the mathematical properties of probabilities; how much they differ depends on how closely the two questions are related.[4] Put another way, it is a way of representing epistemic plausibilities but it can yield answers that contradict those arrived at using probability theory.

Dempster–Shafer theory is based on two ideas:

1. Obtaining degrees of belief for one question from subjective probabilities for a related question.
2. Dempster's rule for combining such degrees of belief when they are based on independent items of evidence.

In essence, the degree of belief in a proposition depends primarily upon the number of answers (to the related questions) containing the proposition, and the subjective probability of each answer. Also contributing are the rules of combination that reflect general assumptions about the data.

In this formalism a **degree of belief** (also referred to as a **mass**) is represented as a **belief function** rather than a Bayesian probability distribution. Probability values are assigned to *sets* of possibilities rather than single events: their appeal rests on the fact they naturally encode evidence in favor of propositions.

Dempster–Shafer theory assigns its masses to all of the non-empty subsets of the entities that compose a system.[*clarification needed*]

Belief and plausibility:

Shafer's framework allows for belief about propositions to be represented as intervals, bounded by two values, *belief* (or *support*) and *plausibility*:

belief ≤ *plausibility*.

Belief in a hypothesis is constituted by the sum of the masses of all sets enclosed by it (*i.e.* the sum of the masses of all subsets of the hypothesis).[*clarification needed*]

It is the amount of belief that directly supports a given hypothesis at least in part, forming a lower bound. Belief (usually denoted *Bel*) measures the strength of the evidence in favor of a set of propositions. It ranges from 0 (indicating no evidence) to 1 (denoting certainty).

Plausibility is 1 minus the sum of the masses of all sets whose intersection with the hypothesis is empty. It is an upper bound on the possibility that the hypothesis could be true, *i.e.* it "could possibly be the true state of the system" up to that value, because there is only so much evidence that contradicts that hypothesis.

Plausibility (denoted by *Pl*) is defined to be

Pl(s)=1-Bel(~s) ..... Equation 1

It also ranges from 0 to 1 and measures the extent to which evidence in favor of ~*s* leaves room for belief in *s*.

For example: Consider a simplified Diagnosis problem to cause FEVER

All : allergy

Flu: flu

Cold: cold

Pneu: pneumonia

Θ might consist of the set {All, Flu, Cold, Pneu}. Each contains 0.2 % evidence to cause fever. (i.e) **All = 0.2, Flu= 0.2, Cold= 0.2, Pneu=0.2** Our goal is to attach some measure of belief to elements of Θ.

Let us see how m works for our diagnosis problem. Assume that we have no information about how to choose among four hypothses when we start the diagnosis task. Then we define m as:

{ Θ} (1.0) (i.e 100 percent evidence to cause fever)……Eq 2

Fever might be such a piece of evidence. We update m as follows:

{ Flu, Cold, Pneu} (0.2+0.2+0.2=0.6).....Eq 3

Where Flu, Cold, Pneu serves 60 % evidence to cause fever and remaining 40% is assigned to value **Θ.**

{ Θ} (0.4).....eq 4

At this point we assigned to the set {flu, cold, Pneu} the appropriate belief. The remainder of the belief still resides in the larger set Θ. Thus Bel(p) is our overall belief that the correct answer lies somewhere in the set p.

We are given two Belief functions m1 and m2. suppose m1 corresponds to our belief after observing fever. From equation 3 and 4.

{ Flu, Cold, Pneu} (0.6)

{ Θ} (0.4)

Suppose m2 corresponds to our belief after observing allergy in addition weget

Allergy =0.2 therefore (0.6 + 0.2 =0.8) {All, Flu, Cold} (0.8)….....eq no 5

The we can compute the combinationm3 using the following table (in which we further abbreviations disease names

M1 (x) * M2(x)..... eq no 6

		M2(x)			
M1 (x)		**{A,FC}**	**(0.8)**	**Θ**	**(0.2)**
{F,C,P}	**(0.6)**	*{F,C}*	*(0.48)*	*{F,C,P}*	*(0.12)*
Θ	**(0.4)**	*{A,F,C}*	*(0.32)*	Θ	*(0.08)*

As a result of applying m1 and m2, we produced a new piece of evidence over fever.

{Flu, Cold} (0.48)..... eq no 7

{All, Flu, Cold} (0.32) .....eq no 8

{Flu, Cold, Pneu} (0.12).....eq no 9

Θ (0.08)..... eq no 10

Now let m3 corresponds to our belief given just the evidence for allergy

From eq no 5 allergy= 0.8+ 0.1 =0.9 remaining 0.1% serves evidence for Θ.

{All} (0.9).....eq no 11

Θ (0.1).....eq no 12

We can apply the numerator of the combination rule to produce (where * denotes the empty set)

		M2(x)			
M1 (x)		**{A }**	**(0.9)**	**Θ**	**(0.1)**
{F,C}	**(0.48)**	*φ*	*(0.432)*	*{F,C}*	*(0.048)*
{A,F,C}	**(0.32)**	*{A,F,C}*	*(0.288)*	*{A,F,C}*	*(0.032)*
{F,C,P}	**(0.12)**	*φ*	*(0.108)*	*(F,C,P)*	*(0.012)*
Θ	**(0.08)**	*{A}*	*(0.072)*	**Θ**	*(0.008)*

Pl(s)=1-Bel(~s) {i.e} *φ= 0.54* **= 1- 0.54 = 0.46**

In this example the percentage value of m5 that was initially assigned to the empty set was large (over half)

UNIT IV

Planning and Learning: Planning with state space search - conditional planning-continuous planning - Multi-Agent planning. Forms of learning - inductive learning - Reinforcement Learning - learning decision trees - Neural Net learning and Genetic learning

---

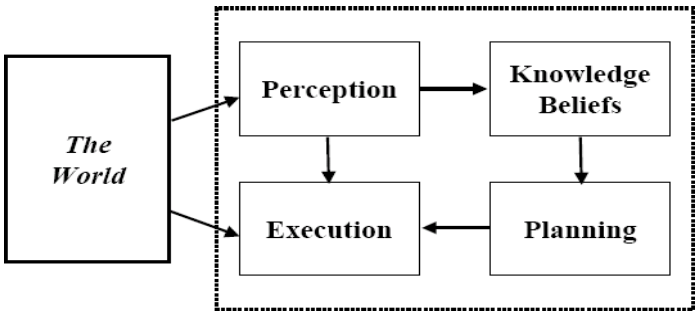
1.What is Planning in AI?Explain the Planning done by an agent?

Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of goals to be achieved

An Agent Architecture



Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)

- Search often proceeds through *plan space* rather than *state space* (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Typical assumptions

- Atomic time: Each action is indivisible
- No concurrent actions are allowed (though actions do not need to be ordered with respect to each other in the plan)
- Deterministic actions: The result of actions are completely determined—there is no uncertainty in their effects
- Agent is the sole cause of change in the world
- Agent is omniscient: Has complete knowledge of the state of the world
- Closed World Assumption: everything known to be true in the world is included in the state description. Anything not listed is false.

<u>Blocks world</u>

The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(a)

ontable(c)

on(b,a)

handempty

clear(b)

clear(c)

General Problem Solver

- The General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon)
- GPS generated actions that reduced the difference between some state and a goal state
- GPS used Means-Ends Analysis
  - Compare what is given or known with what is desired and select a reasonable thing to do next
  - Use a table of differences to identify procedures to reduce types of differences
- GPS was a state space planner: it operated in the domain of state space problems specified by an initial state, some goal states, and a set of operations

Situation calculus planning

- Intuition: Represent the planning problem using first-order logic
  - Situation calculus lets us reason about changes in the world
  - Use theorem proving to ‒prove‖ that a particular sequence of actions, when applied to the situation characterizing the world state, will lead to a desired result

Situation calculus

- **Initial state**: a logical sentence about (situation) S_0

  At(Home, S_0) ^ ~Have(Milk, S_0) ^ ~ Have(Bananas, S_0) ^ ~Have(Drill, S_0)
- **Goal state**:

  (∃s) At(Home,s) ^ Have(Milk,s) ^ Have(Bananas,s) ^ Have(Drill,s)
- **Operators** are descriptions of how the world changes as a result of the agent's actions:

  ∀(a,s) Have(Milk,Result(a,s)) <=> ((a=Buy(Milk) ^ At(Grocery,s)) ∨ (Have(Milk, s) ^ a~=Drop(Milk)))
- Result(a,s) names the situation resulting from executing action a in situation s.
- Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:

  (∀s) Result'([],s) = s

  (∀a,p,s) Result'([a|p]s) = Result'(p,Result(a,s))

Basic representations for planning

- Classic approach first used in the **STRIPS** planner circa 1970
- States represented as a conjunction of ground literals
  - at(Home) ^ ~have(Milk) ^ ~have(bananas) ...
- Goals are conjunctions of literals, but may have variables which are assumed to be existentially quantified
  - at(?x) ^ have(Milk) ^ have(bananas) ...
- Do not need to fully specify state
  - Non-specified either don't-care or assumed false
  - Represent many cases in small storage
  - Often only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

<u>Operator/action representation</u>

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied

<u>Example:</u>

Op[Action: Go(there),
Precond: At(here) ^ Path(here,there),
Effect: At(there) ^ ~At(here)]

- All variables are universally quantified
- Situation variables are implicit
  - preconditions must be true in the state immediately before operator is applied; effects are true immediately after

Blocks world operators

Here are the classic basic operations for the blocks world:

- stack(X,Y): put block X on block Y
- unstack(X,Y): remove block X from block Y
- pickup(X): pickup block X
- putdown(X): put block X on the table

Each will be represented by

- a list of preconditions
- a list of new facts to be added (add-effects)
- a list of facts to be removed (delete-effects)
- optionally, a set of (simple) variable constraints

For example:

```
preconditions(stack(X,Y), [holding(X),clear(Y)])
deletes(stack(X,Y), [holding(X),clear(Y)]).
adds(stack(X,Y), [handempty,on(X,Y),clear(X)])
constraints(stack(X,Y), [X\==Y,Y\==table,X\==table])
```

<u>STRIPS planning</u>

- STRIPS maintain two additional data structures:
  - **State List** - all currently true predicates.
  - **Goal Stack** - a push down stack of goals to be solved, with current goal on top of stack.
- If current goal is not satisfied by present state, examine add lists of operators, and push operator and preconditions list on stack. (Sub goals)
- When a current goal is satisfied, POP it from stack.
- When an operator is on top stack, record the application of that operator on the plan sequence and use the operator's add and delete lists to update the current state.

Typical BW planning problem

Initial state:

clear(a)

clear(b)

clear(c)

ontable(a)

ontable(b)

ontable(c)

handempty

Goal interaction

- Simple planning algorithms assume that the goals to be achieved are independent
  - Each can be solved separately and then the solutions concatenated
- This planning problem, called the ‗Sussman Anomaly,‖ is the classic example of the goal interaction problem:
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS could not handle this, although minor modifications can get it to do simple cases

State-space planning

- We initially have a space of situations (where you are, what you have, etc.)
- The plan is a solution found by ‗searching‖ through the situations to get to the goal
- A **progression planner** searches forward from initial state to goal state
- A **regression planner** searches backward from the goal
  - This works if operators have enough information to go both ways
  - Ideally this leads to reduced branching –you are only considering things that are relevant to the goal

Plan-space planning

- An alternative is to **search through the space of *plans***, rather than situations.

- Start from a **partial plan** which is expanded and refined until a complete plan that solves the problem is generated.
- **Refinement operators** add constraints to the partial plan and modification operators for other changes.
- We can still use STRIPS-style operators:

  Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)

  Op(ACTION: RightSock, EFFECT: RightSockOn)

  Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)

  Op(ACTION: LeftSock, EFFECT: leftSockOn)

could result in a partial plan of

[RightShoe, LeftShoe]

2.<u>What is learning and its representation in AI explain?</u>

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 <u>computers</u> that can be taught rather than programmed.

<u>*Learning*</u> 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

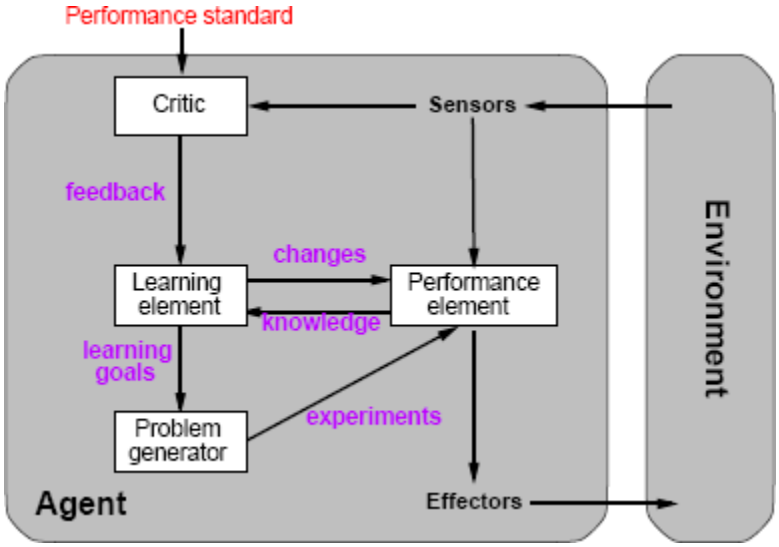
- Here one learns knowledge without the aid of a teacher.

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.

General model:

- **Learning is essential for unknown environments, i.e., when designer lacks omniscience. Learning is useful as a system construction method, i.e., expose the agent to reality rather than trying to write it down.**
- **Learning modifies the agent's decision mechanisms to improve performance**



-

Learning element

- **Design of a learning element is affected by**
- – **Which components of the performance element are to be learned**
- – **What feedback is available to learn these components**
- – **What representation is used for the components**
- **Type of feedback:**
- – **Supervised learning: correct answers for each example**
- – **Unsupervised learning: correct answers not given**
- – **Reinforcement learning: occasional rewards**

3.Explain Inductive Learning

Inductive learning is a kind of learning in which, given a set of examples an agent tries to estimate or create an evaluation function. Most inductive learning is supervised learning, in which examples provided with classifications. (The alternative is *clustering*.) More formally, an

example is a pair *(x, f(x))*, where *x* is the input and *f(x)* is the output of the function applied to *x*. The task of *pure inductive inference* (or *induction*) is, given a set of examples of *f*, to find a hypothesis *h* that approximates *f*.

- Given an example, A pair <x, f(x)> where x is the input and f(x) is the result
- Generate a hypothesis, A function h(x) that approximates f(x)
- That will generalize well, Correctly predict values for unseen samples

How do we do this?

_Must determine a hypothesis space...

- The set of all hypotheses we are willing to consider
- _e.g. All functions of degree less than 10

That is realizable, Contains the true function

Inductive learning method

Simplest form: learn a function from examples (tabula rasa)

f is the target function

An example is a pair x, f(x), e.g.,

Problem: find a(n) hypothesis h

such that h≈f

given a training set of examples

(This is a highly simpli_ed model of real learning:

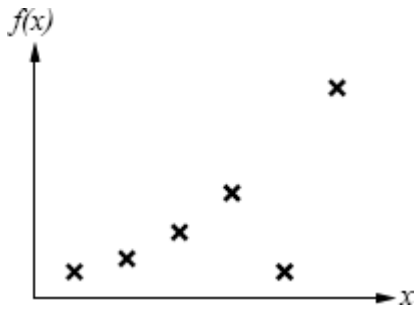
{ Ignores prior knowledge

{ Assumes a deterministic, observable \environment"

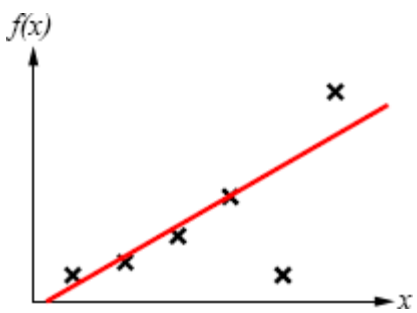
{ Assumes examples are given

{ Assumes that the agent wants to learn f|why?)

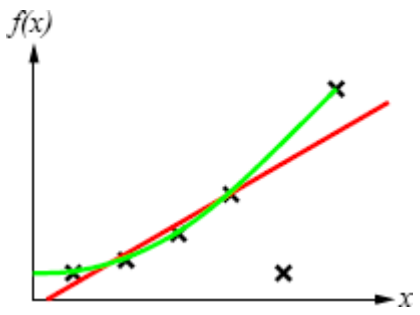
- **Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f* on all examples)**
- **E.g., curve fitting:**



-
- **Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f* on all examples)**
- **E.g., curve fitting:**



- **Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f* on all examples)**
- **E.g., curve fitting:**



- **Construct/adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f* on all examples)**
- **E.g., curve fitting:**

The most likely hypothesis is the simplest one consistent with the data."

Since there can be noise in the measurements, in practice we need to make

a tradeoff between simplicity of the hypothesis and how well it fits the data.

5.Explain Reinforcement Learning

As opposed to *supervised learning*, reinforcement learning takes place in an environment where the agent cannot directly compare the results of its action to a desired result. Instead, it is given some *reward* or *punishment* that relates to its actions. It may win or lose a game, or be told it has made a good move or a poor one. The job of reinforcement learning is to find a successful function using these rewards.

The reason reinforcement learning is harder than supervised learning is that the agent is never told what the right action is, only whether it is doing well or poorly, and in some cases (such as chess) it may only receive feedback after a long string of actions.

There are two basic kinds of information an agent can try to learn.

- **utility function** -- The agent learns the utility of being in various states, and chooses actions to maximize the expected utility of their outcomes. This requires the agent keep a model of the environment.

- **action-value** -- The agent learns an action-value function giving the expected utility of performing an action in a given state. This is called *Q-learning*. This is the *model-free* approach.

Passive Learning in a Known Environment

Assuming an environment consisting of a set of states, some terminal and some nonterminal, and a model that specifies the probabilities of transition from state to state, an agent learns passively by observing a set of *training sequences*, which consist of a set of state transitions followed by a reward.

The goal is to use the reward information to learn the expected utility of each of the nonterminal states. **An important simplifying assumption is that the utility of a sequence is the sum of the rewards accumulated in the states of the sequence**. That is, the utility function is *additive*.

A passive learning agent keeps an estimate *U* of the utility of each state, a table *N* of how many times each state was seen, and a table *M* of transition probabilities. There are a variety of ways the agent can update its table *U*.

Naive Updating

One simple updating method is the *least mean squares* (LMS) approach [Widrow and Hoff, 1960]. It assumes that the observed reward-to-go of a state in a sequence provides direct evidence of the actual reward-to-go. The approach is simply to keep the utility as a running average of the rewards based upon the number of times the state has been seen. This approach minimizes the mean square error with respect to the observed data.

This approach converges very slowly, because it ignores the fact that **the actual utility of a state is the probability-weighted average of its successors' utilities, plus its own reward**. LMS disregards these probabilities.

Adaptive Dynamic Programming

If the transition probabilities and the rewards of the states are known (which will usually happen after a reasonably small set of training examples), then the actual utilities can be computed directly as

U(i) = R(i) + SUMj MijU(j)

where *U(i)* is the utility of state *i*, *R* is its reward, and *Mij* is the probability of transition from state *i* to state *j*. This is identical to a single *value determination* in the policy iteration algorithm for Markov decision processes. *Adaptive dynamic programming* is any kind of reinforcement learning method that works by solving the utility equations using a dynamic programming algorithm. It is exact, but of course highly inefficient in large state spaces.

Temporal Difference Learning

[Richard Sutton] Temporal difference learning uses the difference in utility values between successive states to adjust them from one epoch to another. The key idea is to use the observed transitions to adjust the values of the observed states so that they agree with the ADP constraint equations. Practically, this means updating the utility of state *i* so that it agrees better with its successor *j*. This is done with the *temporal-difference* (TD) equation:

U(i) <- U(i) + a(R(i) + U(j) - U(i))

where *a* is a *learning rate* parameter.

Temporal difference learning is a way of approximating the ADP constraint equations without solving them for all possible states. The idea generally is to define conditions that hold over local transitions when the utility estimates are correct, and then create update rules that nudge the estimates toward this equation.

This approach will cause *U(i)* to converge to the correct value if the learning rate parameter decreases with the number of times a state has been visited [Dayan, 1992]. In general, as the number of training sequences tends to infinity, TD will converge on the same utilities as ADP.

Passive Learning in an Unknown Environment

Since neither temporal difference learning nor LMS actually use the model *M* of state transition probabilities, they will operate unchanged in an unknown environment. The ADP approach, however, updates its estimated model of an unknown environment after each step, and this model is used to revise the utility estimates.

Any method for learning stochastic functions can be used to learn the environment model; in particular, in a simple environment the transition probability *Mij* is just the percentage of times state *i* has transititoned to *j*.

The basic difference between TD and ADP is that TD adjusts a state to agree with the observed successor, while ADP makes a state agree with all successors that might occur, weighted by their probabilities. More importantly, ADP's adjustments may need to be propagated across all of the utility equations, while TD's affect only the current equation. TD is essentially a crude first approximation to ADP.

A middle-ground can be found by bounding or ordering the number of adjustments made in ADP, beyond the simple one made in TD. The *prioritized-sweeping* heuristic prefers only to make adjustments to states whose *likely* successors have just undergone *large* adjustments in their utility estimates. Such approximate ADP systems can be very nearly as efficient as ADP in terms of convergence, but operate much more quickly.

Active Learning in an Unknown Environment

The difference between active and passive agents is that passive agents learn a fixed policy, while the active agent must decide what action to take and how it will affect its rewards. To represent an active agent, the environment model *M* is extended to give the probability of a transition from a state *i* to a state *j*, *given an action a*. Utility is modified to be the reward of the state plus the maximum utility expected depending upon the agent's action:

U(i) = R(i) + maxa x SUMj MaijU(j)

An ADP agent is extended to learn transition probabilities given actions; this is simply another dimension in its transition table. A TD agent must similarly be extended to have a model of the environment.

Learning Action-Value Functions

An action-value function assigns an expected utility to the result of performing a given action in a given state. If *Q(a, i)* is the value of doing action *a* in state *i*, then

U(i) = maxa Q(a, i)

The equations for Q-learning are similar to those for state-based learning agents. The difference is that Q-learning agents do not need models of the world. The equilibrium equation, which can be used directly (as with ADP agents) is

Q(a, i) = R(i) + SUMj Maij maxa' Q(a', j)

The temporal difference version does not require that a model be learned; its update equation is

Q(a, i) <- Q(a, i) + a(R(i) + maxa' Q(a', j) - Q(a, i))

Applications of Reinforcement Learning

The first significant reinforcement learning system was used in Arthur Samuel's checker-playing program. It used a weighted linear function to evaluate positions, though it did not use observed rewards in its learning process.

TD-gammon [Tesauro, 1992] has an evaluation function represented by a fully-connected neural network with one hidden layer of 80 nodes; with the inclusion of some precomputed board features in its input, it was able to reach world-class play after about 300,000 training games.

A case of reinforcement learning in robotics is the famous *cart-pole* balancing problem. The problem is to control the position of the cart (along a single axis) so as to keep a pole balanced on top of it upright, while staying within the limits of the track length. Actions are usually to jerk left or right, the so-called *bang-bang control* approach. The first work on this problem was the

BOXES system [Michie and Chambers, 1968], in which state space was partitioned into boxes, and reinforcement propogated into the boxes. More recent simulated work using neural networks [Furuta et al., 1984] simulated the *triple-inverted-pendulum* problem, in which three poles balance one atop another on a cart

6. Discuss Neural Net learning and Genetic learning

Neural Networks

More specifically, a neural network consists of a set of *nodes* (or *units*), *links* that connect one node to another, and *weights* associated with each link. Some nodes receive inputs via links; others directly from the environment, and some nodes send outputs out of the network. Learning usually occurs by adjusting the weights on the links.

Each unit has a set of weighted inputs, an *activation level*, and a way to compute its activation level at the next time step. This is done by applying an *activation function* to the weighted sum of the node's inputs. Generally, the weighted sum (also called the *input function*) is a strictly linear sum, while the activation function may be nonlinear. If the value of the activation function is above a *threshold*, the node "fires."

Generally, all nodes share the same activation function and threshold value, and only the topology and weights change.

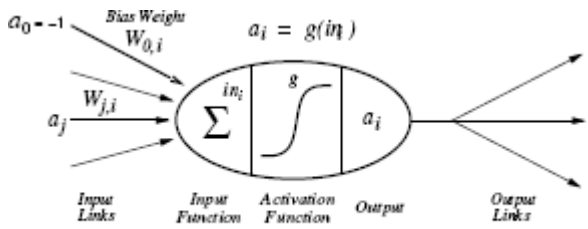


Figure 1: Simple neuron model and basic unit of neural networks

- g is a non-linear function which takes as input a weighted sum of the input link signals (as well as an intrinsic bias weight) and outputs a certain signal strength.
- g is commonly a threshold function or sigmoid function.

Network Structures

The two fundamental types of network structure are *feed-forward* and *recurrent*. A feed-forward network is a directed acyclic graph; information flows in one direction only, and there are no cycles. Such networks cannot represent internal state. Usually, neural networks are also *layered*, meaning that nodes are organized into groups of layers, and links only go from nodes to nodes in adjacent layers.

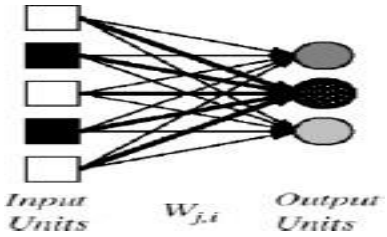
Recurrent networks allow loops, and as a result can represent state, though they are much more complex to analyze. *Hopfield networks* and *Boltzmann machines* are examples of recurrent networks; Hopfield networks are the best understood. All connections in Hopfield networks are bidirectional with symmetric weights, all units have outputs of 1 or -1, and the activation function is the sign function. Also, all nodes in a Hopfield network are both input and output nodes. Interestingly, it has been shown that a Hopfield network can reliably recognize *0.138N* training examples, where *N* is the number of units in the network.

Boltzmann machines allow non-input/output units, and they use a stochastic evaluation function that is based upon the sum of the total weighted input. Boltzmann machines are formally equivalent to a certain kind of belief network evaluated with a stochastic simulation algorithm.

One problem in building neural networks is deciding on the initial topology, e.g., how many nodes there are and how they are connected. Genetic algorithms have been used to explore this problem, but it is a large search space and this is a computationally intensive approach. The

optimal brain damage method uses information theory to determine whether weights can be removed from the network without loss of performance, and possibly improving it. The alternative of making the network larger has been tested with the *tiling algorithm* [Mezard and Nadal, 1989] which takes an approach similar to induction on decision trees; it expands a unit by adding new ones to cover instances it misclassified. *Cross-validation* techniques can be used to determine when the network size is right.

Perceptrons



Perceptrons are single-layer, feed-forward networks that were first studied in the 1950's. They are only capable of learning *linearly separable* functions. That is, if we view *F* features as defining an *F*-dimensional space, the network can recognize any class that involves placing a single hyperplane between the instances of two classes. So, for example, they can easily represent **AND**, **OR**, or **NOT**, but cannot represent **XOR**.

Perceptrons learn by updating the weights on their links in response to the difference between their output value and the correct output value. The updating rule (due to Frank Rosenblatt, 1960) is as follows. Define *Err* as the difference between the correct output and actual output. Then the learning rule for each weight is

Wj <- Wj + A x Ij x Err

where *A* is a constant called the *learning rate*.

Of course, this was too good to last, and in *Perceptrons* [Minsky and Papert, 1969] it was observed how limited linearly separable functions were. Work on perceptrons withered, and neural networks didn't come into vogue again until the 1980's, when multi-layer networks became the focus.

```
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
  inputs: examples, a set of examples, each with input x = x1, ..., xn and output y
          network, a perceptron with weights Wj, j = 0 ... n, and activation function g

  repeat
     for each e in examples do
        in ← Σ_{j=0}^{n} Wj xj[e]
        Err ← y[e] − g(in)
        Wj ← Wj + α × Err × g'(in) × xj[e]
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Multi-Layer Feed-Forward Networks

The standard method for learning in multi-layer feed-forward networks is *back-propagation* [Bryson and Ho, 1969]. Such networks have an input layer, and output layer, and one or more *hidden layers* in between. The difficulty is to divide the blame for an erroneous output among the nodes in the hidden layers.

The back-propagation rule is similar to the perceptron learning rule. If *Erri* is the error at the output node, then the weight update for the link from unit *j* to unit *i* (the output node) is

Wj,i <- Wj,i + A x aj x Erri x g'(ini)

where *g'* is the derivative of the activation function, and *aj* is the activation of the unit *j*. (Note that this means the activation function must have a derivative, so the sigmoid function is usually used rather than the step function.) Define *Di* as *Erri x g'(ini)*.

This updates the weights leading to the output node. To update the weights on the interior links, we use the idea that the hidden node *j* is responsible for part of the error in each of the nodes to which it connects. Thus the error at the output is divided according to the strength of the connection between the output node and the hidden node, and propogated backward to previous layers. Specifically,

$D_j = g'(in_j) \times SUM_i W_{j,i} D_i$

Thus the updating rule for internal nodes is

$W_{j,i} <- W_{j,i} + A \times a_j \times D_i$.

Lastly, the weight updating rule for the weights from the input layer to the hidden layer is is

$W_{k,j} <- W_{k,j} + A \times I_k \times D_j$

where *k* is the input node and *j* the hidden node, and *Ik* is the input value of *k*.

A neural network requires *2n/n* hidden units to represent all Boolean functions of *n* inputs. For *m* training examples and *W* weights, each epoch in the learning process takes *O(mW)* time; but in the worst case, the number of epochs can be exponential in the number of inputs.

In general, if the number of hidden nodes is too large, the network may learn only the training examples, while if the number is too small it may never converge on a set of weights consistent with the training examples.

Multi-layer feed-forward networks can represent any continuous function with a single hidden layer, and any function with two hidden layers [Cybenko, 1988, 1989].

Applications of Neural Networks

John Denker remarked that "neural networks are the second best way of doing just about anything." They provide passable performance on a wide variety of problems that are difficult to solve well using other methods.

NETtalk [Sejnowski and Rosenberg, 1987] was designed to learn how to pronounce written text. Input was a seven-character centered on the target character, and output was a set of Booleans controlling the form of the sound to be produced. It learned 95% accuracy on its training set, but had only 78% accuracy on the test set. Not spectacularly good, but important because it impressed many people with the potential of neural networks.

Other applications include a ZIP code recognition [Le Cun et al., 1989] system that achieves 99% accuracy on handwritten codes, and driving [Pomerleau, 1993] in the ALVINN system at CMU. ALVINN controls the NavLab vehicles, and translates inputs from a video image into

steering control directions. ALVINN performs exceptionally well on the particular road-type it learns, but poorly on other terrain types. The extended MANIAC system [Jochem et al., 1993] has multiple ALVINN subnets combined to handle different road types.

7. What is conditional planning?Explain.

Conditional planning is a way to deal with uncertainty by checking what is actually happening in the environment at predetermined points in the plan. Conditional planning is simplest to explain for fully observable environments, so we will begin with that case. The partially observable case is more difficult, but more interesting.

Conditional planning in fully observable environments

Full observability means that the agent always knows the current state. If the environment is nondeterministic, however, the agent will not be able to predict the *outcome* of its actions. A conditional planning agent handles nondeterminism by building into the plan (at planning time) conditional steps that will check the state of the environment (at execution time) to decide what to do next. The problem, then, is how to construct these conditional plans.

The first thing we need to do is augment the STRIPS language to allow for nondeterminism. To do this, we allow actions to have **disjunctive effects**, meaning that the action can have two or more different outcomes whenever it is executed. For example, suppose that moving *Left* sometimes fails. Then the normal action description

$$Action(Left, \text{PRECOND}{:}AtR, \text{EFFECT}{:}AtL \wedge \neg AtR)$$

must be modified to include a disjunctive effect:

$$Action(Left, \text{PRECOND}{:}AtR, \text{EFFECT}{:}AtL \vee AtR)\ .$$

We will also find it useful to allow actions to have **conditional effects**, wherein the effect of the action depends on the state in which it is executed. Conditional effects appear in the EFFECT slot of an action, and have the syntax "**when** <*condition*>: <*effect*>." For example, to model the *Suck* action, we would write

$$Action(Suck, \text{PRECOND}{:}, \text{EFFECT}{:}(\textbf{when}\ AtL{:}\ CleanL) \wedge (\textbf{when}\ AtR{:}\ CleanR))].$$

Conditional effects do not introduce indeterminacy, but they can help to model it. For example, suppose we have a devious vacuum cleaner that sometimes dumps dirt on the destination square when it moves, but only if that square is clean. This can be modeled with a description such as

$$Action(Left, \text{PRECOND}{:}AtR, \text{EFFECT}{:}AtL \vee (AtL \wedge \textbf{when}\ CleanL{:}\ \neg CleanL)),$$

which is both disjunctive and conditional.[6] To create conditional plans, we need **conditional steps**. We will write these using the syntax "**if** <*test*> **then** *plan_A* **else** *plan_B*," where

<*test*> is a Boolean function of the state variables. For example, a conditional step for the vacuum world might be, "**if** $AtL \wedge CleanL$ **then** *Right* **else** *Suck*." The execution of such a step proceeds in the obvious way. By nesting conditional steps, plans become trees.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(INITIAL-STATE[problem], problem, [])
```

```
function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if GOAL-TEST[problem](state) then return the empty plan
  if state is on path then return failure
  for each action, state_set in SUCCESSORS[problem](state) do
      plan ← AND-SEARCH(state_set, problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure
```

```
function AND-SEARCH(state_set, problem, path) returns a conditional plan, or failure
  for each s_i in state_set do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Conditional planning in partially observable environments

The preceding section dealt with fully observable environments, which have the advantage that conditional tests can ask any question at all and be sure of getting an answer. In the real world, partial observability is much more common. In the initial state of a partially observable planning problem, the agent knows only a certain amount about the actual state. The simplest way to model this situation is to say that the initial state belongs to a **state set**; the state set is a way of describing the agent's initial **belief state**.[8]

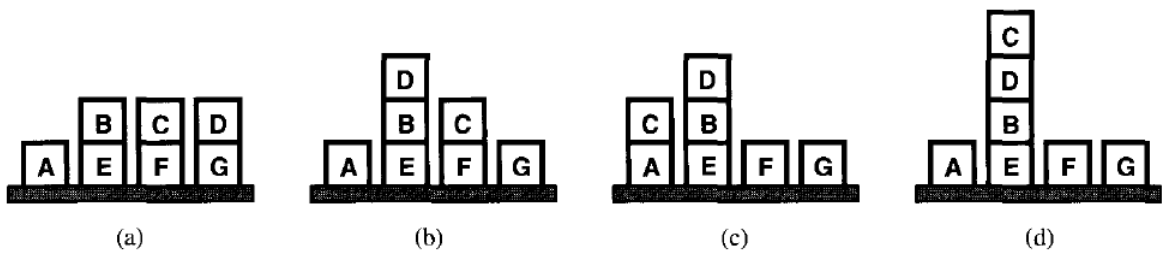
Suppose that a vacuum-world agent knows that it is in the right-hand square and that the square is clean, but it cannot sense the presence or absence of dirt in other squares. Then *as far as it knows* it could be in one of two states: the left-hand square might be either clean or dirty.

\

8. Describe continuous planning.

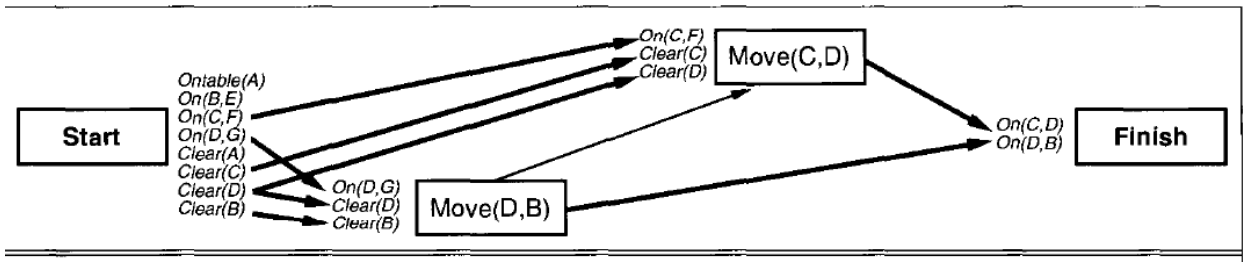
we design an agent that persists indefinitely in an environment. Thus it is not a "problem solver" that is given a single goal and then plans and acts until the goal is achieved; rather, it lives through a series of ever-changing goal formulation, planning, and acting phases. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process in a **continuous planning agent**.

The agent is thought of as always being *part of the way through* executing a plan—the grand plan of living its life. Its activities include executing some steps of the plan that are ready to be executed, refining the plan to satisfy open preconditions or resolve conflicts, and modifying the plan in the light of additional information obtained during execution. Obviously, when it first formulates a new goal, the agent will have no actions ready to execute, so it will spend a while generating a partial plan. It is quite possible, however, for the agent to begin execution before the plan is complete, especially when it has independent subgoals to achieve. The continuous planning agent monitors the world continuously, updating its world model from new percepts even if its deliberations are still continuing.



The sequence of states as the continuous planning agent tries to reach the goal state $On(C, D) \wedge On(D, B)$, as shown in (d). The start state is (a). At (b), another agent has interfered, putting D on B. At (c), the agent has executed $Move(C, D)$ but has failed, dropping C on A instead. It retries $Move(C, D)$, reaching the goal state (d).

Throughout the continuous planning process, $Start$ is always used as the label for the current state. The agent updates the state after each action.



The initial plan constructed by the continuous planning agent. The plan is indistinguishable, so far, from that produced by a normal partial-order planner.

The plan is now ready to be executed, but before the agent can take action, nature intervenes. An external agent (perhaps the agent's teacher getting impatient) moves D onto B and the world is now in the state shown in Figure 12.15(b). The agent perceives this, recognizes that $Clear(B)$ and $On(D, G)$ are no longer true in the current state, and updates its model of the current state accordingly. The causal links that were supplying the preconditions $Clear(B)$ and $On(D, G)$ for the $Move(D, B)$ action become invalid and must be removed from the plan. The new plan is shown in Figure 12.17. At all times, $Start$ represents the current state, so this $Start$ is different from the one in the previous figure. Notice that the plan is now incomplete: two of the preconditions for $Move(D, B)$ are open, and its precondition $On(D, y)$ is now uninstantiated, because there is no longer any reason to assume the that move will be from G.

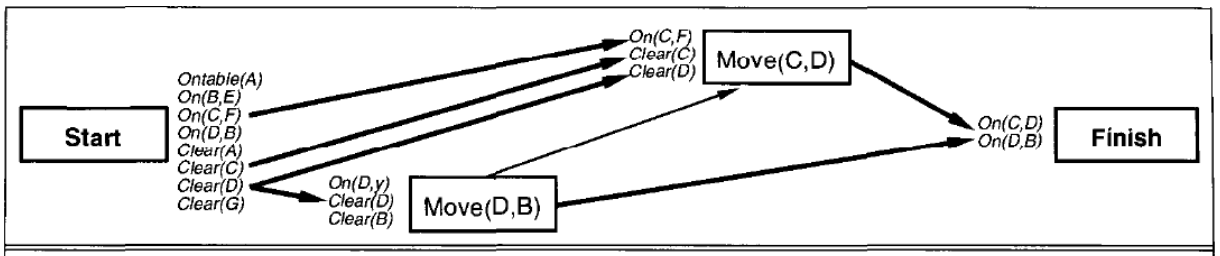


Figure 12.17 After someone else moves D onto B, the unsupported links supplying $Clear(B)$ and $On(D, G)$ are dropped, producing this plan.

Now the agent can take advantage of the "helpful" interference by noticing that the causal link $Move(D, B) \xrightarrow{On(D,B)} Finish$ can be replaced by a direct link from $Start$ to $Finish$. This process is called **extending** a causal link and is done whenever a condition can be supplied by an earlier step instead of a later one without causing a new conflict.

gorithm where the two flaws are open preconditions and causal conflicts. The continuous planning agent, on the other hand, addresses a much broader range of flaws:

- *Missing goal:* The agent can decide to add a new goal or goals to the $Finish$ state. (Under continuous planning, it might make more sense to change the name of $Finish$ to $Infinity$, and of $Start$ to $Current$, but we will stick with tradition.)
- *Open precondition:* Add a causal link to an open precondition, choosing either a new or an existing action (as in POP).
- *Causal Conflict:* Given a causal link $A \xrightarrow{p} B$ and an action C with effect $\neg p$, choose an ordering constraint or variable constraint to resolve the conflict (as in POP).

- *Unsupported link:* If there is a causal link $Start \xrightarrow{p} A$ where p is no longer true in $Start$, then remove the link. (This prevents us from executing an action whose preconditions are false.)
- *Redundant action:* If an action A supplies no causal links, remove it and its links. (This allows us to take advantage of serendipitous events.)
- *Unexecuted action:* If an action A (other than $Finish$) has its preconditions satisfied in $Start$, has no other actions (besides $Start$) ordered before it, and conflicts with no causal links, then remove A and its causal links and return it as the action to be executed.
- *Unnecessary historical goal:* If there are no open preconditions and no actions in the plan (so that all causal links go directly from $Start$ to $Finish$), then we have achieved the current goal set. Remove the goals and the links to them to allow for new goals.

UNIT V

Advanced Topics: Game Playing: Minimax search procedure-Adding alpha-beta cutoffs- Expert System: Representation-Expert System shells-Knowledge Acquisition. Robotics: Hardware-Robotic Perception-Planning-Application domains

Advanced Topics:

GAME PLAYING

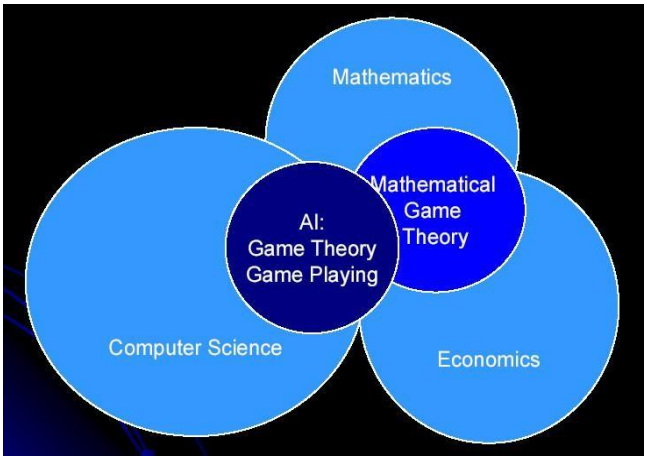
- The term *Game* means a sort of *conflict* in which n individuals or groups (known as players) participate.
- Game theory denotes games of *strategy*.
- John von Neumann is acknowledged as father of game theory. Neumann defined Game theory in 1928 and 1937 and established the mathematical framework for all subsequent theoretical developments.
- Game theory allows decision-makers (players) to cope with other decision-makers (players) who have different purposes in mind. In other words, players determine their own strategies in terms of the strategies and goals of their opponent. Games are integral attribute of human beings.
- Games engage the *intellectual faculties* of humans.
- If computers are to mimic people they should be able to play games.

Definition of Game

- A game has at least *two players*.
- Solitaire is not considered a game by game theory.
- The term 'solitaire' is used for single-player games of concentration.
- An instance of a game begins with a player choosing from a set of specified *(game rules)* alternatives. This choice is called a *move*.
- After first move, the new situation determines which player to make next move and alternatives available to that player.
    - In many board games, the next move is by other player.
    - In many multi-player card games, the player making next move depends on who dealt, who took last trick, won last hand, etc.
- Minimax - The least good of all good outcomes.
- Maximin - The least bad of all bad outcomes**.**

The primary game theory is the Mini-Max Theorem. This theorem says :

"If a Minimax of one player corresponds to a Maximin of the other player, then that outcome is the best both players can hope for."



MINIMAX SEARCH PROCEDURE

The minimax search procedure is a depth first, depth limited search procedure. The idea is to start at the current position and use the plausible move generator to generate the set of possible successor positions.

Consider two players, zero sum, non-random, perfect knowledge games.

Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, and Othello.

Formalizing Game

A general and a Tic-Tac-Toe game in particular.

Consider 2-Person, Zero-Sum, Perfect Information

Both players have access to complete information about the state of the game.

No information is hidden from either player.

Players alternately move.

Apply Iterative methods

Required because search space may be large for the games to search for a solution.

Do search before each move to select the next best move.

Adversary Methods

Required because alternate moves are made by an opponent , who is trying to win, are not controllable.

Static Evaluation Function f(n)

‡ Used to evaluate the "goodness" of a configuration of the game.

It estimates board quality leading to win for one player.

Example: Let the board associated with node n then

If f(n) = large +ve value means the board is good for me and bad for opponent.

If f(n) = large -ve value means the board is bad for me and good for opponent.

If f(n) near 0 Means the board is a neutral position.

If f(n) = +infinity means a winning position for me.

If f(n) = -infinity means a winning position for opponent.

- Games hold an inexplicable fascination for many people, and the notion that computers might play games has existed at least as long as computers.
- Charles Babbage the nineteenth century computer architect, thought about programming his analytical engine to play chess and later of building a machine to play tic tac toe.
- Two of the pioneers of the science of information and computing contributed to the fledgling computer game playing literature.
- Claude Shannon [1950] wrote a paper in which he described mechanisms that could be used in a program to play chess.
- A few years later, Alan Turing described a chess playing program, although he never built it.

There are two main reasons that games appeared to be a good domain in which to explore machine intelligence.

1. They provide a structured task in which it is very easy to measure success or failure.
2. They did not obviously require large amounts of knowledge.

For example in chess:

1. The average branching factor is around 35.
2. In a average game, each player might makes 50 moves.
3. So in order to examine complete game tree, we would have to examine 35^{100} positions.

<u>Adversarial Search</u>

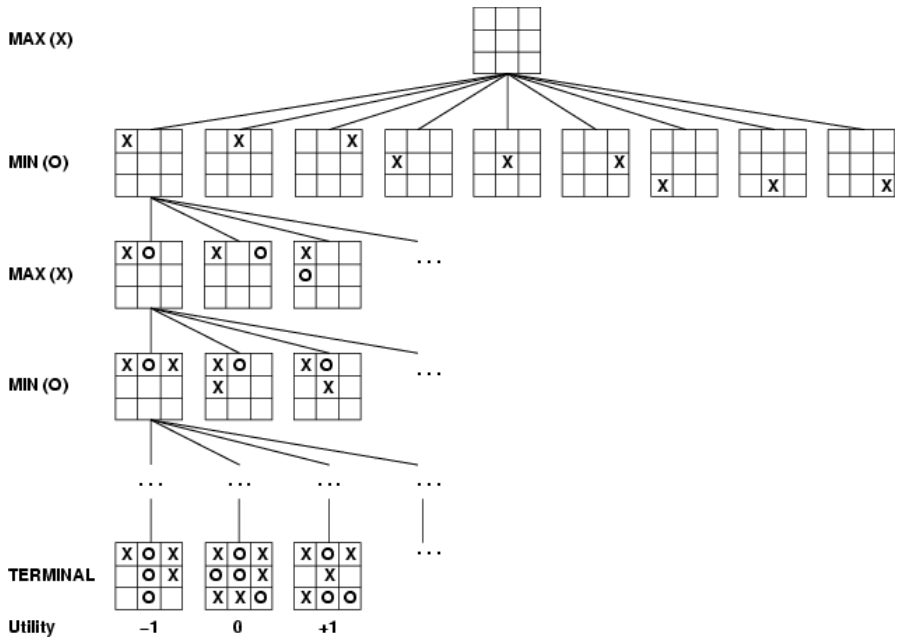
Competitive environments, in which the agents' goals are in conflict, give rise to adversarial search problems- often known as games. In AI, "games" are usually of a rather specialized kind – in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite.

A game can be formally defined as a kind of search problem with the following components:

- The initial state, which includes the board position and identifies the player to move.

- A successor function, which returns a list of (move, state)pairs, each indicating a legal move and the resulting state.
- A terminal test, which determines when the game is over. States where the game has ended are called terminal states.
- A utility function, which gives a numeric value for the terminal states.

The initial state and the legal moves for each side define the game tree for the game. The following figure shows part of the game tree for tic-tac-toe. From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. Game tree (2-player, deterministic, turns)



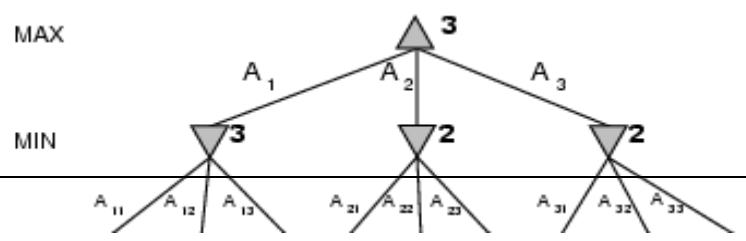
Minimax

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node, which we write as MINIMAX-VALUE(n). The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game.

•Perfect play for deterministic games

•Idea: choose move to position with highest minimax value = best achievable payoff against best play

•E.g., 2-ply game:



Here, the first MIN node, labeled B, has three successors with values 3,12, and 8, so its minimax value is 3. The minimax algorithm computes the minimax decision from the current state.

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
---------------------------------------------------------
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v
---------------------------------------------------------
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

<u>Properties of minimax</u>

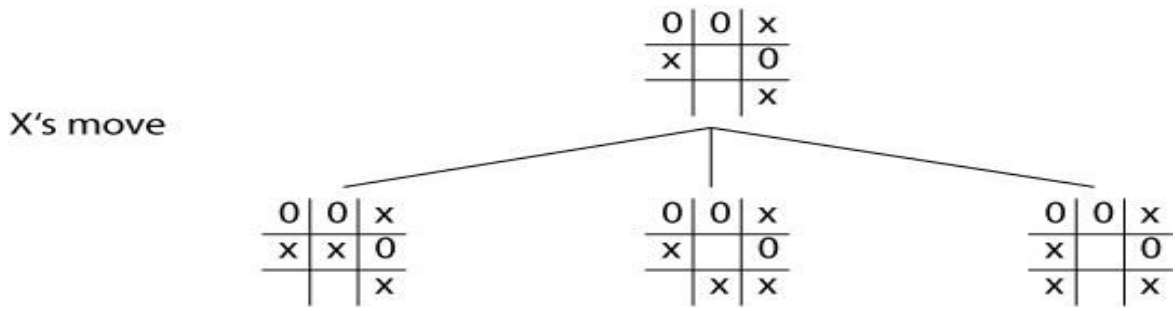
- <u>Complete?</u> Yes (if tree is finite)
- <u>Optimal?</u> Yes (against an optimal opponent)
- <u>Time complexity?</u> $O(b^m)$:the maximum depth of the tree is m, and there are b legal moves at each point
    - <u>Space complexity?</u> O(bm) (depth-first
- With "perfect ordering," time complexity = $O(b^{m/2})$

  → doubles depth of search
- exploration)
- For chess, b ≈35, m ≈100 for "reasonable" games

  → exact solution completely infeasible

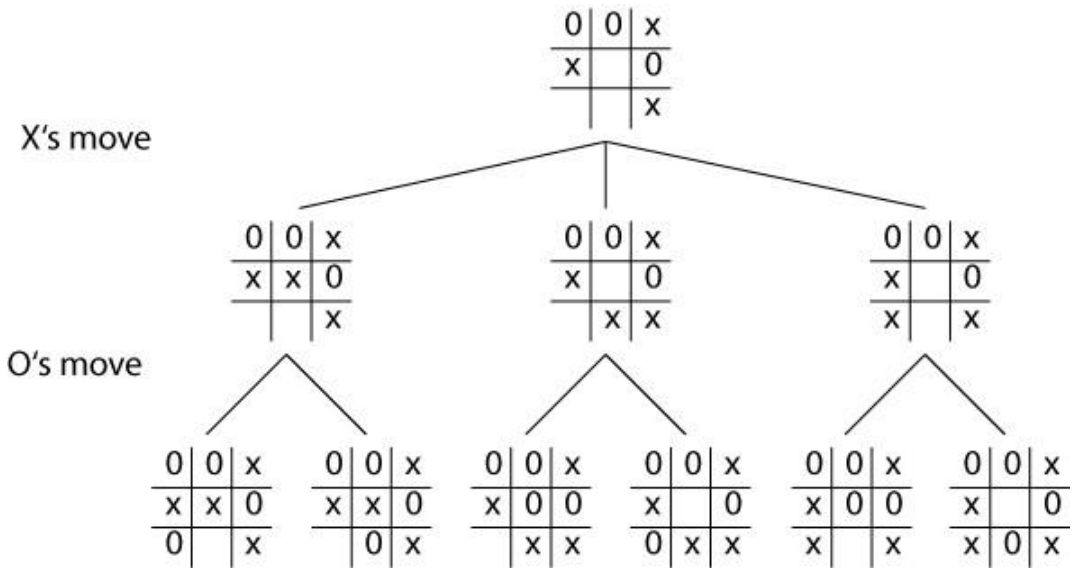
EXAMPLE: ***Game playing with Mini-Max***

<u>STEP:1</u>

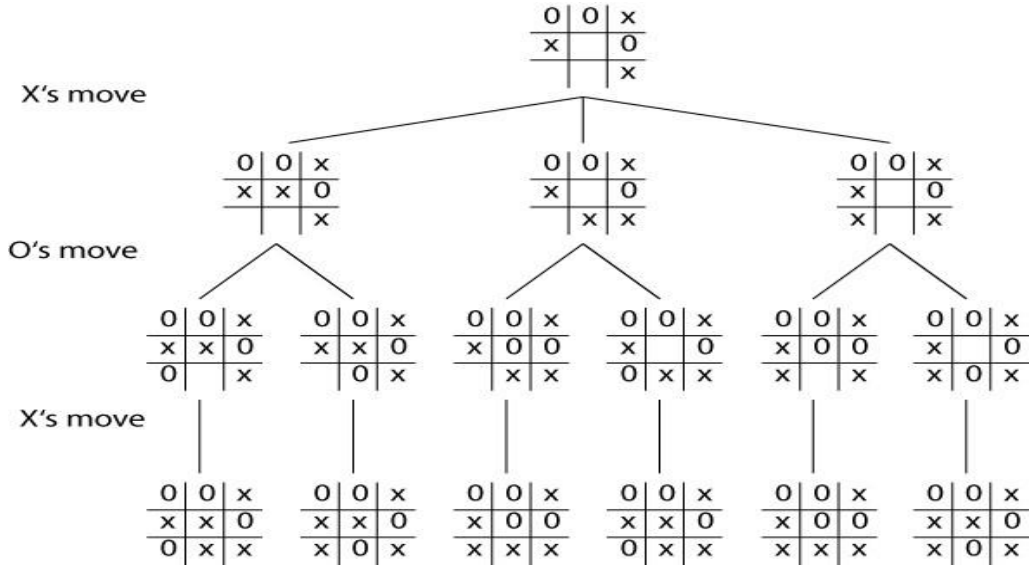


Game Playing with Mini-Max - Tic-Tac-Toe

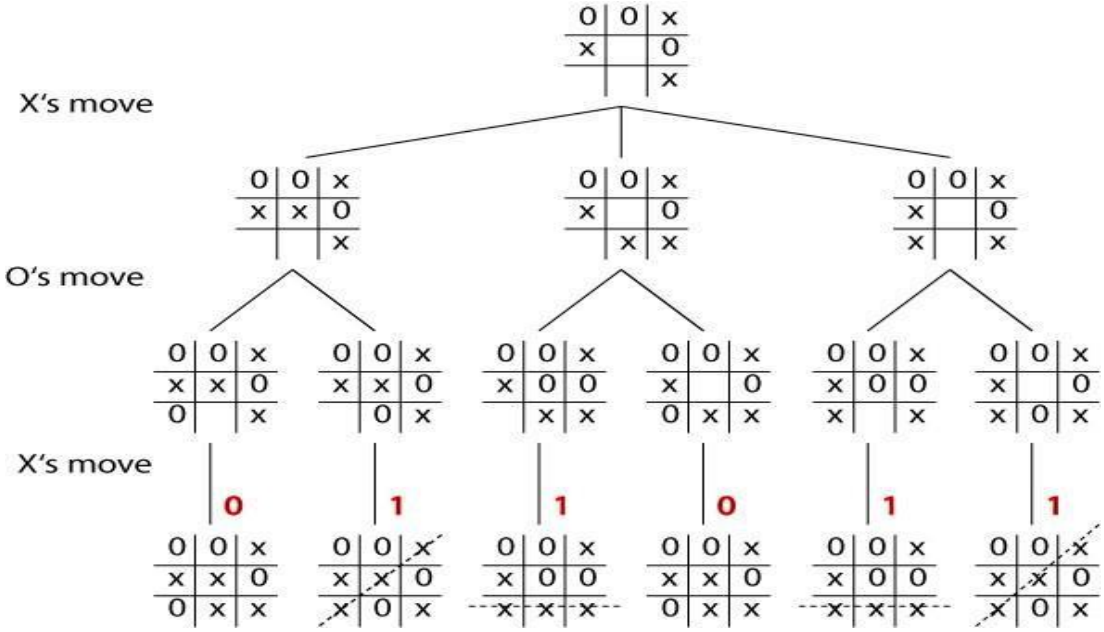
STEP:2



STEP:3



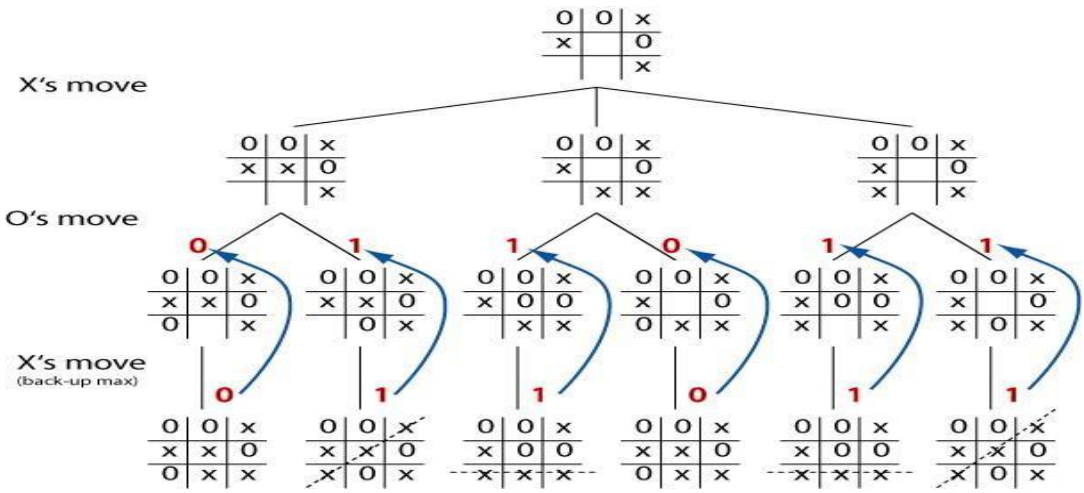
STEP:4



Static Evaluation:

'+1' for a win, '0' for a draw

Criteria '+1' for a Win, '0' for a Draw



Back-up the Evaluations:

Level by level, on the basis of opponent's turn

Up : One Level

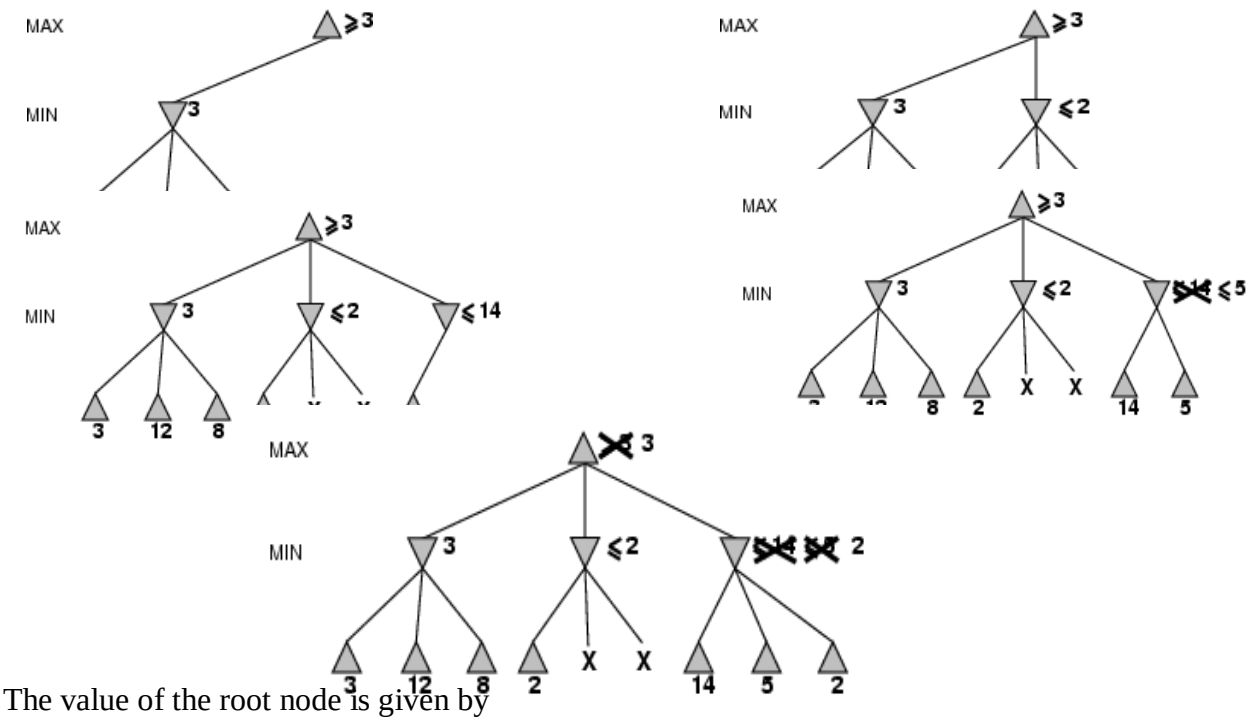
ALPHA-BETA PRUNING

The problem with minimax procedure is that the number of game states it has to examine is exponential in the number of moves. We can cut it in half using the technique called alpha-

beta pruning. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider again the two-ply game tree. The steps are explained in the following figure. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

α-β pruning example



The value of the root node is given by

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,x,y), min(14,5,2))

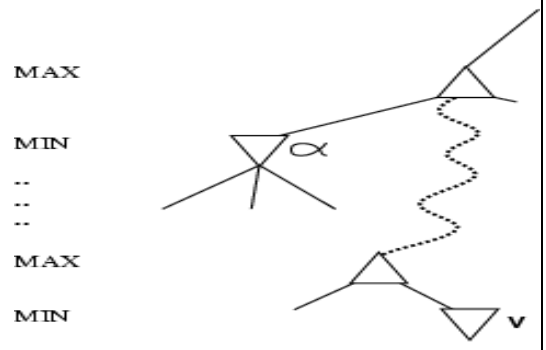
= max(3,min(2,x,y),2)

=max(3,z,2) where z<=2

=3

x and y: two unevaluated successors

z: minimum of x and y



<u>Properties of α-β</u>

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- A simple example of the value of reasoning about which computations are relevant (a form of meta reasoning)

<u>Why is it called α-β?</u>

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*
- β is the value of the best (i.e., lowest-value) choice found so far at any choice point along the path for *min*

  → If *v* is worse than α, *max* will avoid it prune that branch

Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN respectively. The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. The algorithm is given below:

The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
```

```
function MIN-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s, α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

2. EXPERT SYSTEM ITS REPRESENTATION

Expert Systems (ES), also called Knowledge-Based Systems (KBS) or simply Knowledge Systems (KS), are computer programs that use expertise to assist people in performing a wide variety of functions, including diagnosis, planning, scheduling and design. ESs are distinguished from conventional computer programs in two essential ways (Barr, Cohen et al. 1989):

1. Expert systems reason with domain-specific knowledge that is symbolic as well as numerical;
2. Expert systems use domain-specific methods that are heuristic (i.e., plausible) as well as algorithmic.

The technology of expert systems has had a far greater impact than even the expert systems business. Expert system technology has become widespread and deeply embedded. As expert system techniques have matured into a standard information technology, the most important recent trend is the increasing integration of this technology with conventional information processing, such as data processing or management information systems.

The Building Blocks of Expert Systems

Every expert system consists of two principal parts: the knowledge base; and the reasoning, or inference, engine.

- The *knowledge base* of expert systems contains both factual and heuristic knowledge. *Factual knowledge* is that knowledge of the task domain that is widely shared, typically found in textbooks or journals, and commonly agreed upon by those knowledgeable in the particular field.
- *Heuristic knowledge* is the less rigorous, more experiential, more judgmental knowledge of performance.
- In contrast to factual knowledge, heuristic knowledge is rarely discussed, and is largely individualistic.
- It is the knowledge of good practice, good judgment, and plausible reasoning in the field. It is the knowledge that underlies the "art of good guessing."
- *Knowledge representation* formalizes and organizes the knowledge. One widely used representation is the *production rule*, or simply *rule*.
- A rule consists of an IF part and a THEN part (also called a *condition* and an *action*). The IF part lists a set of conditions in some logical combination.
- The piece of knowledge represented by the production rule is relevant to the line of reasoning being developed if the IF part of the rule is satisfied; consequently, the THEN part can be concluded, or its problem-solving action taken.
- Expert systems whose knowledge is represented in rule form are called *rule-based systems*.
- Another widely used representation, called the *unit* (also known as *frame*, *schema*, or *list structure*) is based upon a more passive view of knowledge.
- The unit is an assemblage of associated symbolic knowledge about an entity to be represented. Typically, a unit consists of a list of properties of the entity and associated values for those properties.

Since every task domain consists of many entities that stand in various relations, the properties can also be used to specify relations, and the values of these properties are the names of other units that are linked according to the relations. One unit can also represent knowledge that is a "special case" of another unit, or some units can be "parts of" another unit.

The *problem-solving model*, or *paradigm*, organizes and controls the steps taken to solve the problem. One common but powerful paradigm involves chaining of IF-THEN rules to form a line of reasoning. If the chaining starts from a set of conditions and moves toward some conclusion, the method is called *forward chaining*. If the conclusion is known (for example, a goal to be achieved) but the path to that conclusion is not known, then reasoning backwards is called for, and the method is *backward chaining*. These problem-solving methods are built into program modules called *inference engines* or *inference procedures* that manipulate and use knowledge in the knowledge base to form a line of reasoning.

- The most important ingredient in any expert system is knowledge. The power of expert systems resides in the specific, high-quality knowledge they contain about task domains.
- AI researchers will continue to explore and add to the current repertoire of knowledge representation and reasoning methods. But in knowledge resides the power.
- Because of the importance of knowledge in expert systems and because the current knowledge acquisition method is slow and tedious, much of the future of expert systems depends on breaking the knowledge acquisition bottleneck and in codifying and representing a large knowledge infrastructure.

EXPERT SYSTEM SHELL

- A rule-based, expert system maintains a separation between its Knowledge-base and that part of the system that executes rules, often referred to as the ***expert system shell***.
- The system shell is indifferent to the rules it executes. This is an important distinction, because it means that the expert system shell can be applied to many different problem domains with little or no change.
- It also means that adding or modifying rules to an expert system can effect changes in program behavior without affecting the controlling component, the system shell.
- The language used to express a rule is closely related to the language ***subject matter experts*** use to describe problem solutions.
- When the subject matter expert composes a rule using this language, he is, at the same time, creating a written record of problem knowledge, which can then be shared with others.

- Thus, the creation of a rule kills two birds with one stone; the rule adds functionality or changes program behavior, and records essential information about the problem domain in a human-readable form. Knowledge captured and maintained by these systems ensures continuity of operations even as subject matter experts (i.e., mathematicians, accountants, physicians) retire or transfer.
- Furthermore, changes to the Knowledge-base can be made easily by subject matter experts without programmer intervention, thereby reducing the cost of software maintenance and helping to ensure that changes are made in the way they were intended.
- Rules are added to the knowledge-base by subject matter experts using text or graphical editors that are integral to the system shell. The simple process by which rules are added to the knowledge-base is depicted in Figure 1.

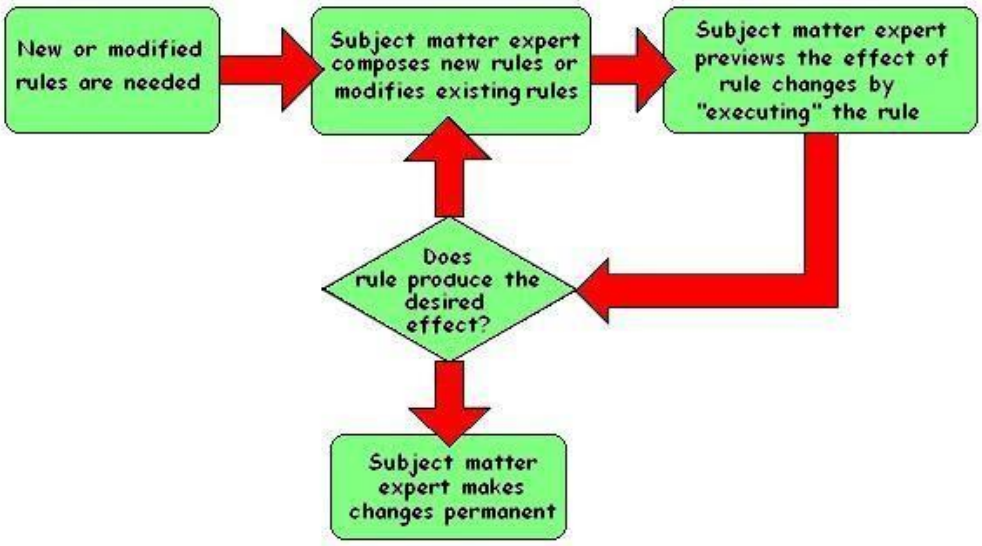


Figure 1 - Creating a knowledge-base

Finally, the expert system never forgets, can store and retrieve more knowledge than any single human being can remember, and makes no errors, provided the rules created by the subject matter experts accurately model the problem at hand.

EXPERT SYSTEM ARCHITECTURE

An expert system is, typically, composed of two major components, the Knowledge-base and the Expert System Shell.

The Knowledge-base is a collection of rules encoded as *metadata* in a file system, or more often in a relational database. The Expert System Shell is a problem-independent component housing facilities for creating, editing, and executing rules. A software architecture for an expert system is illustrated in Figure 2.

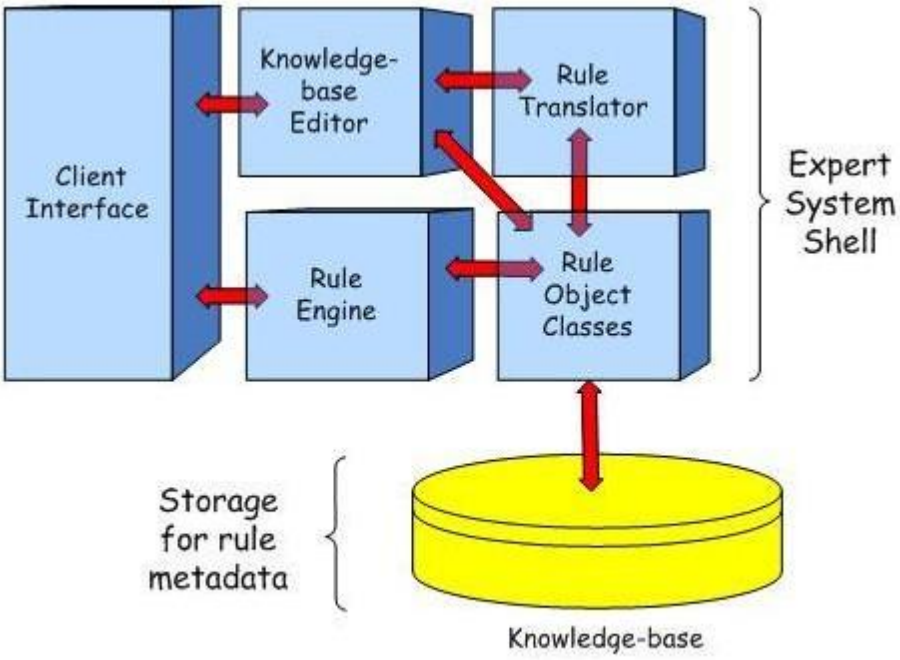


Figure 2 - Expert System Architecture

The **shell** portion includes software modules whose purpose it is to,

- Process requests for service from system users and application layer modules;
- Support the creation and modification of business rules by subject matter experts;
- Translate business rules, created by a subject matter experts, into machine-readable forms;
- Execute business rules; and
- Provide low-level support to expert system components (e.g., retrieve metadata from and save metadata to knowledge base, build Abstract Syntax Trees during rule translation of business rules, etc.).

3. ROBOTICS:

Definition

"A reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through various programmed motions for the performance of a variety of tasks"

A robot has these essential characteristics:

- **Sensing:** First of all your robot would have to be able to sense its surroundings. It would do this in ways that are not similar to the way that you sense your surroundings. Giving your robot sensors: light sensors (eyes), touch and pressure sensors (hands), chemical

sensors (nose), hearing and sonar sensors (ears), and taste sensors (tongue) will give your robot awareness of its environment.

- **Movement:** A robot needs to be able to move around its environment. Whether rolling on wheels, walking on legs or propelling by thrusters a robot needs to be able to move. To count as a robot either the whole robot moves, like the Sojourner or just parts of the robot moves, like the Canada Arm.
- **Energy:** A robot needs to be able to power itself. A robot might be solar powered, electrically powered, battery powered. The way your robot gets its energy will depend on what your robot needs to do.
- **Intelligence:** A robot needs some kind of "smarts." This is where programming enters the pictures. A programmer is the person who gives the robot its 'smarts.' The robot will have to have some way to receive the program so that it knows what it is to do.

ROBOTICS HARDWARE

1. **Sensors**

Sensors are the perceptual interface between robots and their environments. Passive sensors. Such as cameras are true observers of the environment: they capture signals that are generated by other sources in the environment. Active sensors, such as sonar, send energy into the environment.

Many mobile robots make use of range finders, which are sensors that measure the distance to nearby objects. One common type is the sonar sensor, also known as and ultrasonic transducer.

Some range sensors measure very short or very long distances. Close-range sensors include tactile sensors such as whiskers, bumb panels, and touch-sensitive skin. At the other end of the spectrum is the Global Positioning System (GPS), which measures the distance to satellites that emit pulsed signals.

2. **Effectors**

Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors,, it will help to talk about motion and shape in the abstract, using the concept of a degree of freedom(DOF).

The dynamic state of a robot includes one additional dimension for the rate of change of each kinematic dimension.

The arm in Figure 25.3(a) has exactly six degrees of freedom. Created by five revolute joints that generate rotational motion and one prismatic joint that generates sliding motion.

The car has 3 effective degrees of freedom but 2 controllable degress of freedom. We say a robot is nonholonomic if it has more effective DOF s than controllable DOFs and holonomic if the two numbers are the same.

For mobile robots, there exists a range of mechanisms for locomotion, including wheels,tracks and legs. Differential drive robots possess two independently actuated wheels.

ROBOTIC PERCEPTION

Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because in general the sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. As a rule of thumb, good internal representations have three properties:

- contain enough information for the robot to make the right decisions,
- structured such that can be updated efficiently
- Natural in the sense that internal variable correspond to natural state variables in the physical world.

Localization:

- Localization is a generic example of robot perception. It is the problem of determining where things are.
- Localization is one of the most pervasive perception problems in robotics, because knowledge about where things are is at the core of any successful physical interaction.
- For example , robot manipulators must know the location of objects they manipulate. Navigating robots must know where they are in order to find their way to goal locations.
- The localization problem comes in three flavours of increasing difficulty. If the initial pose of the object to be localized is known, localizations is a **tracking** problem.
- Tracking problems are characterized by bounded uncertainty. More difficult is the global localization problem, in which the initial location of the object is entirely unknown.
- Global localization problems turn into tracking problems once the object of interest has been localized, but they also involve phases where the robot has to manage very broad uncertainties.
- Finally, we can be mean to our robot and “KIDNAP” the object it is attempting to localize. Localization under such devious conditions is known as the Kidnapping problem.
- Kidnapping is often used to test the robustness of a localization technique under extreme conditions.

Mapping:

In the literature, the robot mapping problem is often referred to as **simultaneous localization and mapping**, abbreviated as SLAM. Not only must the robot construct a map, it must do so without knowing where it is. SLAM is one of the core problems in robotics.

PLANNING TO MOVE

In robotics, decisions ultimately involve motion of effectors.

- The **point-to-point motion problem** is to deliver the robot or its end-effectors to a designated target location.
- A greater challenge is the **complaint motion problem**, in which a robot moves while being physical contact with an obstacle .
- An example of compliant motion is a robot manipulator that screws in a light bulb , or a robot that pushes a box across a table top.
- We begin by finding a suitable representation in which motion planning problems can be described and solved.
- It turns out that the configuration space – the space of robot states defined by location orientation, and joint angles – is a better place to work than the original 3D space.
- The **path planning problem** is to find a path from one configuration to another in configuration space.

The literature on robot path planning distinguishes a range of different techniques specifically aimed at finding paths in high-dimensional continuous spaces.

The major families of approaches are known as **cell decomposition and skeletonization**. Each reduces the continuous path-planning problem to a discrete graph search problem by identifying some canonical states and paths within the free space.