ARTIFICIAL INTELLIGENCE

UNIT-I PROBLEM

SOLVING

Introduction – Agents – Problem formulation - Uninformed search strategies – Heuristics - Informed search strategies - Constraint satisfaction

What is artificial intelligence?

- **Artificial Intelligence** is the branch of computer science concerned with making computers behave like humans.
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.
- **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

Systems that think like humans "The exciting new effort to make computers think … machines with minds, in the full and literal sense."(Haugeland,1985)	**Systems that think rationally** "The study of mental faculties through the use of computer models." (Charniak and McDermont,1985)
Systems that act like humans The art of creating machines that performs functions that require intelligence when performed by people."(Kurzweil,1990)	**Systems that act rationally** "Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998)

Applications of Artificial Intelligence:

- **Autonomous planning and scheduling:**

  A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et* al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

- **Game playing:**

  IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

- **Autonomous control:**

  The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.

- **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.

- **Logistics Planning:**

  During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

- **Robotics:**

  Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia *et* al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

- **Language understanding and problem solving:**

  PROVERB (Littman *et al.,* 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

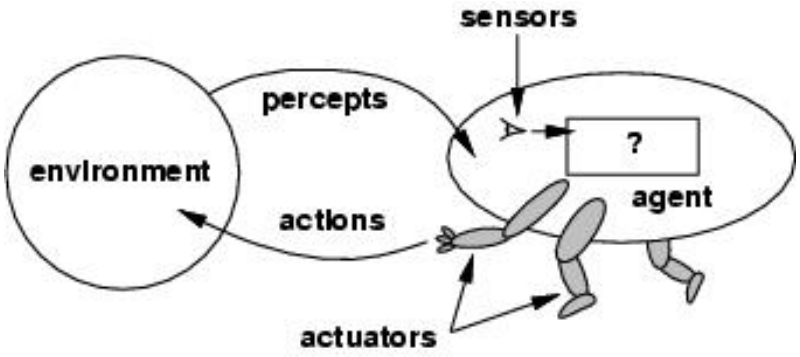
AGENTS:

Rationality concept can be used to develop a smallest of design principle for building successful agents; these systems are reasonably called as Intelligent.

Agents and environments:

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and SENSOR acting upon that environment through **actuators.** This simple idea is illustrated in Figure.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and 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.



Percept

We use the term **percept** to refer to the agent's perceptual inputs at any given instant.

Percept Sequence

An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

Agent function

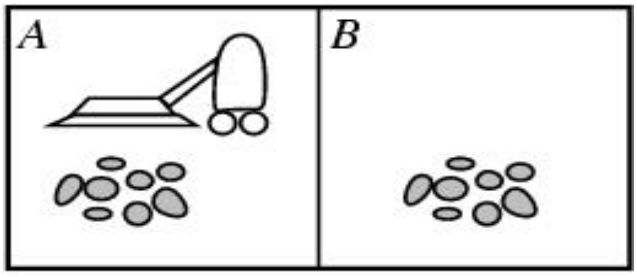
Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Agent program

- 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.

- To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure.
- This particular world has just two locations: squares A and B.
- The vacuum agent perceives which square it is in and 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,
- it move to the other square.
- A partial tabulation of this agent function is shown in Figure.



Agent function

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
…..	…..

Agent program

function Reflex-VACUUM-AGENT ([locations, status]) **returns an action**

if status = Dirty **then return** Suck
else if location = A **then return** Right
elseif location = B **then return** Left

Good Behavior: The concept of Rationality

- A **rational agent** is one that does the right thing-conceptually speaking; every entry in the table for the agent function is filled out correctly.
- Obviously, doing the right thing is better than doing the wrong thing.
- The right action is the one that will cause the agent to be most successful.

Performance measures

- **A performance measure** embodies the **criterion for success** of an agent's behavior.
- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states.
- If the sequence is desirable, then the agent has performed well.

Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.
- This leads to a **definition of a 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.

Omniscience, learning, and autonomy

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- Doing actions in order to modify future percepts-sometimes called **information gathering**-is an important part of rationality.
- Our definition requires a rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.
- To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy.
- A rational agent should be **autonomous**-it should learn what it can to compensate for partial or incorrect prior knowledge.

Task environments

- We must think about **task environments,** which are essentially the "**problems**" to which rational agents are the "**solutions**."

Specifying the task environment

- The rationality of the simple vacuum-cleaner agent, needs specification of
  - ✓ the performance measure
  - ✓ the environment
  - ✓ the agent's actuators
  - ✓ Sensors.

PEAS

- All these are grouped together under the heading of the **task environment.**
- We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.
- In designing an agent, the first step must always be to specify the task environment as fully as possible.
- The following table shows PEAS description of the task environment for an automated taxi.

Agent Type	**Performance Measure**	**Environments**	**Actuators**	**Sensors**
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads,other traffic,pedestrians, customers	Steering,accelerator, brake, Signal,horn,display	Cameras,sonar, Speedometer,GPS, Odometer,engine sensors,keyboards, accelerometer

- The following table shows PEAS description of the task environment for some other agent type.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Properties of task environments

- **Fully observable vs. partially observable**
- **Deterministic vs. stochastic**
- **Episodic vs. sequential**
- **Static vs. dynamic**
- **Discrete vs. continuous**
- **Single agent vs. multiagent**

Fully observable vs. **partially observable.**

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
- A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

Deterministic vs. **stochastic.**

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic;
- Otherwise, it is stochastic.

Episodic vs. **sequential**

- In an **episodic task environment**, the agent's experience is divided into atomic episodes.
- Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.
- For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;
- In **sequential environments**, on the other hand, the current decision Could affect all future decisions.
- Chess and taxi driving are sequential:

Discrete vs. **continuous.**

- The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.
- For example, a discrete-state environment such as a chess game has a finite number of distinct states.
- Chess also has a discrete set of percepts and actions.
- Taxi driving is a continuous- state and continuous-time problem:
- The speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
- Taxi-driving actions are also continuous (steering angles, etc.)

Single agent vs. **multiagent.**

- An agent solving a crossword puzzle by itself is clearly in a single-agent environment,
- Where as an agent playing chess is in a two-agent environment.
- Multiagent is further classified in to two ways

  - ✓ Competitive multiagent environment
  - ✓ Cooperative multiagent environment

Agent programs

- The job of Artificial Intelligence is to design the agent program that implements the agent function mapping percepts to actions
- The agent program will run in an architecture
- An architecture is a computing device with physical sensors and actuators
- Where Agent is combination of Program and Architecture

Agent = Program + Architecture

- An agent program takes the current percept as input while the agent function takes the entire percept history
- Current percept is taken as input to the agent program because nothing more is available from the environment
- The following TABLE-DRIVEN_AGENT program is invoked for each new percept and returns an action each time

Function TABLE-DRIVEN_AGENT (percept) **returns** an action

static: percepts, a sequence initially empty
table, a table of actions, indexed by percept sequence

append percept to the end of percepts
action ← LOOKUP(percepts, table)
return action

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional

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

Some Agent Types

- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**
  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.

Kinds of Agent Programs

- The following are the agent programs,
  - Simple reflex agents
  - Mode-based reflex agents
  - Goal-based reflex agents
  - Utility-based agents

Simple Reflex Agent

- The simplest kind of agent is the **simple reflex agent.**
- These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history.
- For example, the vacuum agent whose agent function is tabulated is given below,
- a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.
- Select action on the basis of *only the current* percept.E.g. the vacuum-agent
- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules*
- If dirty then suck

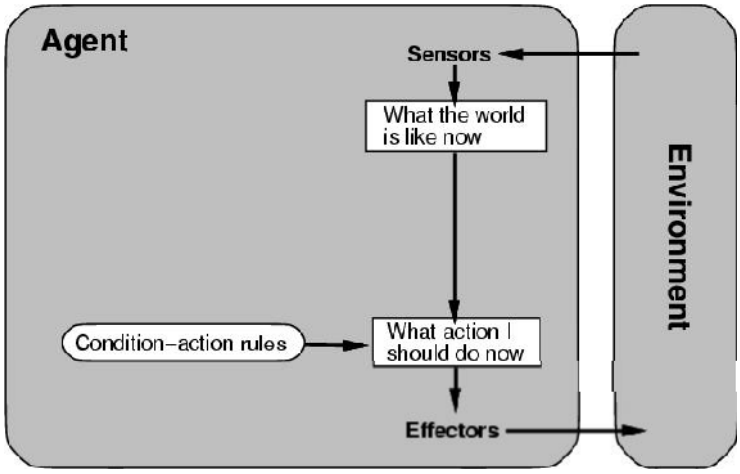
A Simple Reflex Agent: Schema

- Schematic diagram of a simple reflex agent.
- The following simple reflex agents, acts according to a rule whose condition matches the current state, as defined by the percept

function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
static: *rules*, a set of condition-action rules
state ← INTERPRET-

INPUT(*percept*) SVCET

rule ← RULE-MATCH(*state*, *rule*)
action ← RULE-ACTION[*rule*] return *action*



- The agent program for a simple reflex agent in the two-state vacuum environment.

function REFLEX-VACUUM-AGENT ([location, status]) return an action
if status == Dirty then return Suck
else if location == A then return Right
else if location == B then return Left

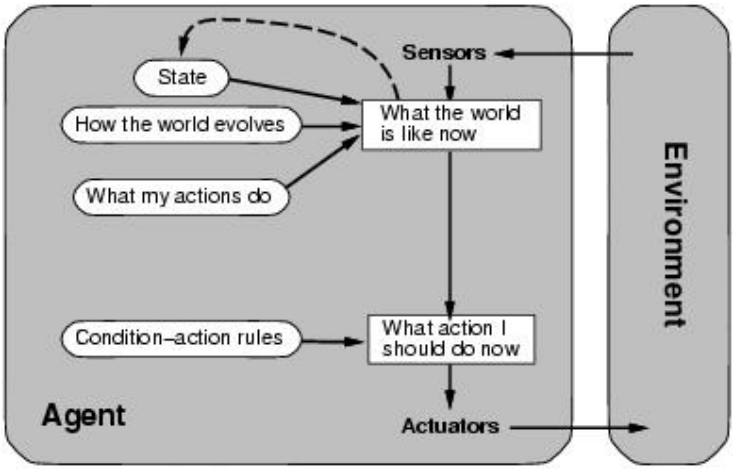
Characteristics

- Only works if the environment is fully observable.
- Lacking history, easily get stuck in infinite loops
- One solution is to randomize actions

Model-based reflex agents

- The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now.*
- That is, 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 as time goes by requires two kinds of knowledge to be encoded in the agent program.
- First, we need some information about how the world evolves independently of the agent
- For example, that an overtaking car generally will be closer behind than it was a moment ago.
- Second, we need some information about how the agent's own actions affect the world

- For example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.
- This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world.
- An agent that uses such a MODEL-BASED model is called a **model-based agent.**
- Schematic diagram of A model based reflex agent

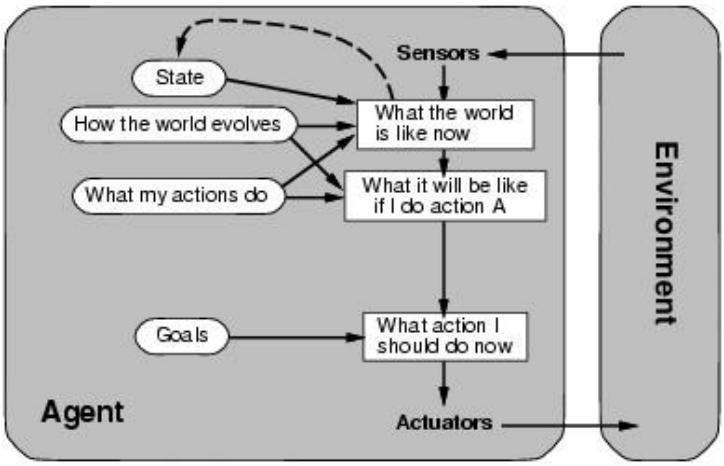


- Model based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** an action
static: *rules*, a set of condition-action rules
state, a description of the current world state
action, the most recent action.
state ← VPDATE-STATE(*state*, *action*, *percept*)
rule ← RVLE-MATCH(*state*, *rule*)
action ← RVLE-ACTION[*rule*] return *action*

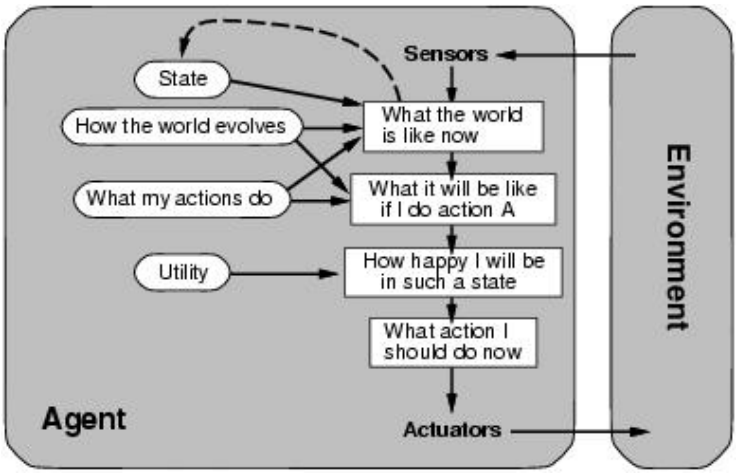
Goal-based agents

- 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.
- Schematic diagram of the goal-based agent's structure.



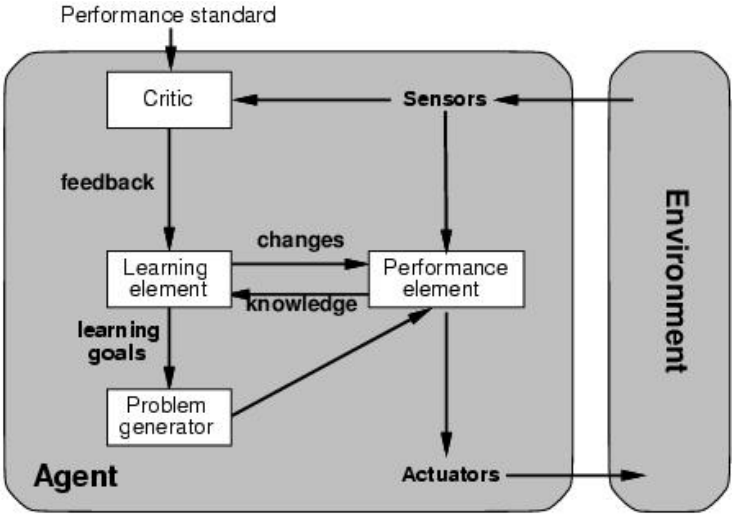
Utility-based agents

- Goals alone are not really enough to generate high-quality behavior in most environments.
- For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others.
- Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas 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. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.
- Schematic diagram of a utility-based agents
- It uses a model of the world, along with a utility function that measures its preferences among states of the world.
- Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.
- Certain goals can be reached in different ways
    - Some are better, have a higher utility
- Vtility function maps a (Sequence of) state(S) onto a real number.
- Improves on goal:
    - Selecting between conflicting goals
    - Select appropriately between several goals based on likelihood of Success



Learning Agent

Schematic diagram of Learning Agent



- All agents can improve their performance through learning.
- A learning agent can be divided into four conceptual components, as,
  - Learning element
  - Performance element
  - Critic
  - Problem generator
- The most important distinction is between the **learning element**, which is responsible for making improvements,
- The **performance element**, which is responsible for selecting external actions.
- The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on 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 last component of the learning agent is the **problem generator**.

- It is responsible for suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run.
- The problem generator's job is to suggest these **exploratory actions**. This is what scientists do when they carry out experiments.

Summary: Intelligent Agents

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- Task environment – **PEAS (P**erformance**, E**nvironment, **A**ctuators, **S**ensors**)**
- The most challenging environments are inaccessible, nondeterministic, dynamic, and continuous.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **agent program** maps from percept to action and updates internal state.
  - **Reflex agents** respond immediately to percepts.
    - simple reflex agents
    - model-based reflex agents
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based agents** maximize their own utility function.
- All agents can improve their performance through **learning.**

Problem Formulation

- An important aspect of intelligence is *goal-based* problem solving.
- The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal.
- Each action changes the *state* and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.
- **A well-defined problem can be described by:**
  - **Initial state**
  - **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
  - **State space** - all states reachable from initial by any sequence of actions
  - **Path -** sequence through state space
  - **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
  - **Goal test -** test to determine if at goal state
- What is **Search**?
- Search is the systematic examination of states to find path from the start/root state to the goal state.
- The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
- The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

Problem-solving agents

- A Problem solving agent is a goal-based agent.
- It decides what to do by finding sequence of actions that lead to desirable states.
- The agent can adopt a goal and aim at satisfying it.
- To illustrate the agent's behavior
- For example where our agent is in the city of Arad, which is in Romania. The agent has to adopt a goal of getting to Bucharest.
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
- The agent's task is to find out which sequence of actions will get to a goal state.
- Problem formulation is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem
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 formulation and problem formulation

A **problem** is defined by four items:
initial state e.g., "at Arad"
successor function S(x) = set of action-state pairs
e.g., S(Arad) = {[Arad -> Zerind;Zerind],….}
goal test, can be
explicit, e.g., x = at Bucharest"
implicit, e.g., NoDirt(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 >= 0

A **solution** is a sequence of actions leading from the initial state to a goal state.

Search

- An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value,and then choosing the best sequence.
- The process of looking for sequences actions from the current state to reach the goal state is called **search.**

- The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.**
- Once a solution is found,the **execution phase** consists of carrying out the recommended action.
- The following shows a simple "formulate,search,execute" design for the agent.
- Once solution has been executed, the agent will formulate a new goal.
- It first formulates a **goal** and a **problem**,searches for a sequence of actions that would solve a problem,and executes the actions one at a time.

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
inputs : *percept*, a percept
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 VPDATE-STATE(*state, percept*)
if seq is empty **then do**
goal ← FORMVLATE-GOAL(*state*)
problem ← FORMVLATE-PROBLEM(*state, goal*)
seq ← SEARCH(*problem*)
action ← FIRST(*seq*);
seq ← REST(seq)
return *action*

- The agent design assumes the Environment is

- **Static**: The entire process carried out without paying attention to changes that might be occurring in the environment.
- **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
- **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
- **Deterministic:** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

Well-defined problems and solutions

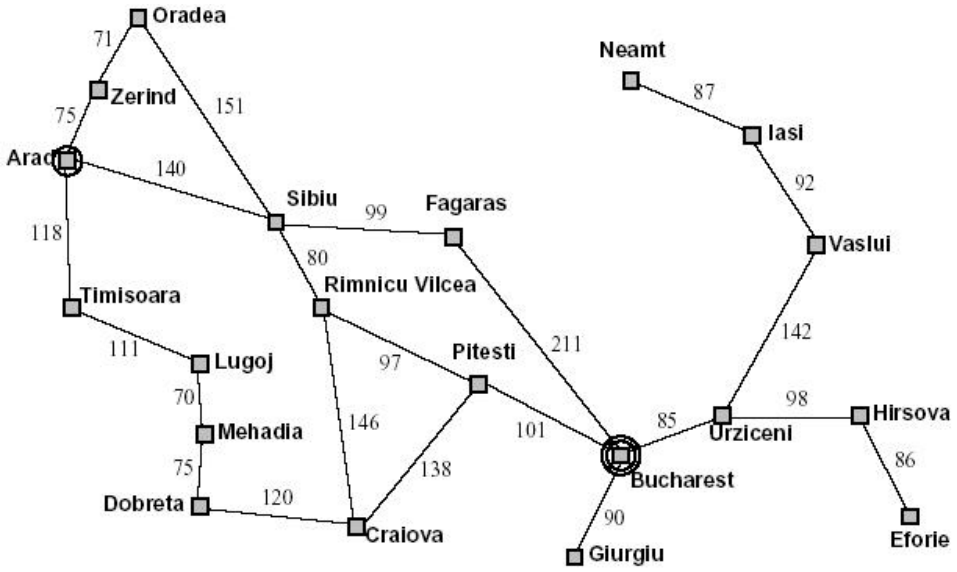
A **problem** can be formally defined by **four components**:

- The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent.
- Given a state x,SVCCESSOR-FN(x) returns a set of {action,successor} ordered pairs where each action is one of the legal actions in state x,and each successor is a state that can be reached from x by applying the action.

- For example, from the state In(Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space**: The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test** determines whether the given state is a goal state.
- A **path cost** function assigns numeric cost to each action.
- For the Romania problem the cost of path might be its length in kilometers.
- The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). It is assumed that the step costs are non negative.
- A **solution** to the problem is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.



A simplified Road Map of part of Romania

Advantages:

- They are easy enough because they can be carried out without further search or planning
- The choice of a good abstraction thus involves removing as much details as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

EXAMPLE PROBLEMS

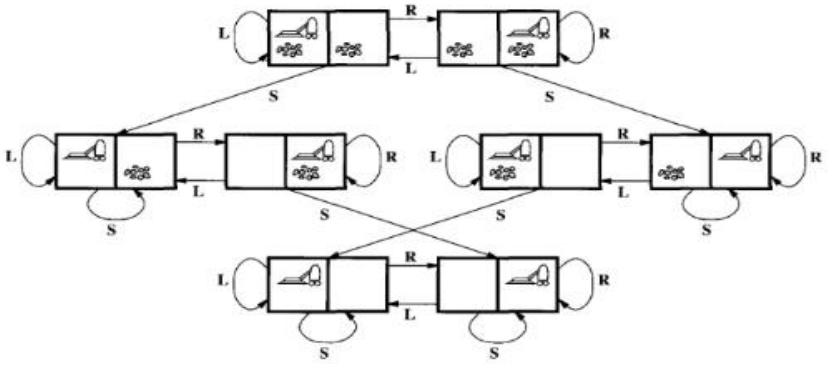
- The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below.
- They are distinguished as toy or real-world problems
  - A T**oy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
  - A R**eal world problem** is one whose solutions people actually care about.

TOY PROBLEMS

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
- **Goal Test** : This tests whether all the squares are clean.
- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

Vacuum World State Space

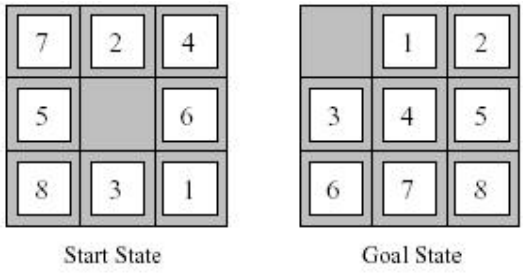


The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

8-puzzle:

- An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space. The object is to reach the specific goal state ,as shown in figure

Example: The 8-puzzle



A typical instance of 8-puzzle.

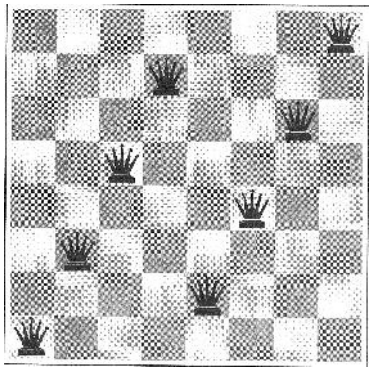
The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions (blank moves Left,Right,Vp or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost**: Each step costs 1,so the path cost is the number of steps in the path.
- **The 8-puzzle** belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI.
- This general class is known as NP-complete.
- The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.
- The **15 puzzle** (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- The **24-puzzle** (on a 5 x 5 board) has around 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-queens problem

- The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal).
- The following figure shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.
- An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and move them around.

In either case the path cost is of no interest because only the final state counts.



8-queens problem

- The first incremental formulation one might try is the following :
  - **States** : Any arrangement of 0 to 8 queens on board is a state.
  - **Initial state** : No queen on the board.
  - **Successor function** : Add a queen to any empty square.
  - **Goal Test** : 8 queens are on the board,none attacked.
- In this formulation, we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.
- A better formulation would prohibit placing a queen in any square that is already attacked. :
  - **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
  - **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queen state space from 3 x 10^{14} to just 2057,and solutions are easy to find.
- For the 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states.
- This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

REAL WORLD PROBLEMS

- A real world problem is one whose solutions people actually care about.
- They tend not to have a single agreed upon description, but attempt is made to give general flavor of their formulation,
- The following are the some real world problems,
  - Route Finding Problem
  - Touring Problems
  - Travelling Salesman Problem
  - Robot Navigation

ROUTE-FINDING PROBLEM

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

- **States :** Each is represented by a location(e.g.,an airport) and the current time.
- **Initial state :** This is specified by the problem.
- **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.
- **Goal Test :** Are we at the destination by some prespecified time?
- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

TOURING PROBLEMS

- **Touring problems** are closely related to route-finding problems,but with an important difference.
- Consider for example, the problem, "Visit every city at least once" as shown in Romania map.
- As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.
  - **Initial state** would be "In Bucharest; visited{Bucharest}".
  - **Intermediate state** would be "In Vaslui; visited {Bucharest,Vrziceni,Vaslui}".
  - **Goal test** would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING SALESPERSON PROBLEM (TSP)

- ✓ TSP is a touring problem in which each city must be visited exactly once.
- ✓ The aim is to find the shortest tour. The problem is known to be **NP-hard**.
- ✓ Enormous efforts have been expended to improve the capabilities of TSP algorithms.
- ✓ These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.
- ✓

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface,the space is essentially two-dimensional.

When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing somework already done.

Another important assembly problem is protein design, in which the goal is to find a sequence of

Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals.

The searching techniques consider internet as a graph of nodes (pages) connected by links.

MEASURING PROBLEM-SOLVING PERFORMANCE

- ✓ The output of problem-solving algorithm is either failure or a solution. (Some algorithms might struck in an infinite loop and never return an output.)
- ✓ The algorithm's performance can be measured in four ways :
  - **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
  - **Optinality** : Does the strategy find the optimal solution
  - **Time complexity:** How long does it take to find a solution?
  - **Space complexity:** How much memory is needed to perform the search?

UNINFORMED SEARCH STRATGES

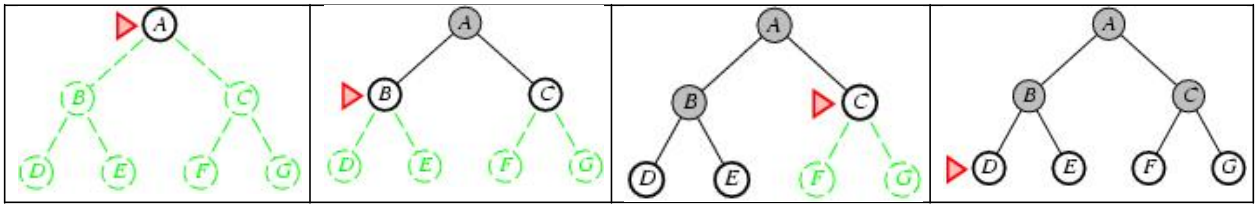
- ✓ **Uninformed Search Strategies** have no additional information about states beyond that provided in the **problem definition**.
- ✓ **Strategies** that know whether one non goal state is "more promising" than another are called I**nformed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Vniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

Breadth-first search

- ✓ Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- ✓ In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- ✓ Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- ✓ In otherwards, calling TREE-SEARCH (problem,FIFO-QVEVE()) results in breadth-first-search.
- ✓ The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.



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

Properties of breadth-first-search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Time and Memory Requirements for BFS – $O(b^{d+1})$

Example:
- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

Time and memory requirements for breadth-first-search.

Time complexity for BFS

- ✓ Assume 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.
- ✓ Each of these generates b more nodes,yielding b^3 nodes at the third level,and so on.
- ✓ Now suppose,that the solution is at depth d.
- ✓ In the worst case,we would expand all but the last node at level d,generating b^{d+1} - b nodes at level d+1.
- ✓ Then the total number of nodes generated is
  $b + b^2 + b^3 + \ldots + b^d + (bd^{+1} + b) = O(b^{d+1)}$.
- ✓ Every node that is generated must remain in memory,because it is either part of the fringe or is an ancestor of a fringe node.
- ✓ The space compleity is,therefore ,the same as the time complexity

UNIFORM-COST SEARCH

- ✓ Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost.
- ✓ uniform-cost search does not care about the number of steps a path has,but only about their total cost.

Properties of Uniform-cost-search:

Expand least-cost unexpanded node

Implementation:
$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

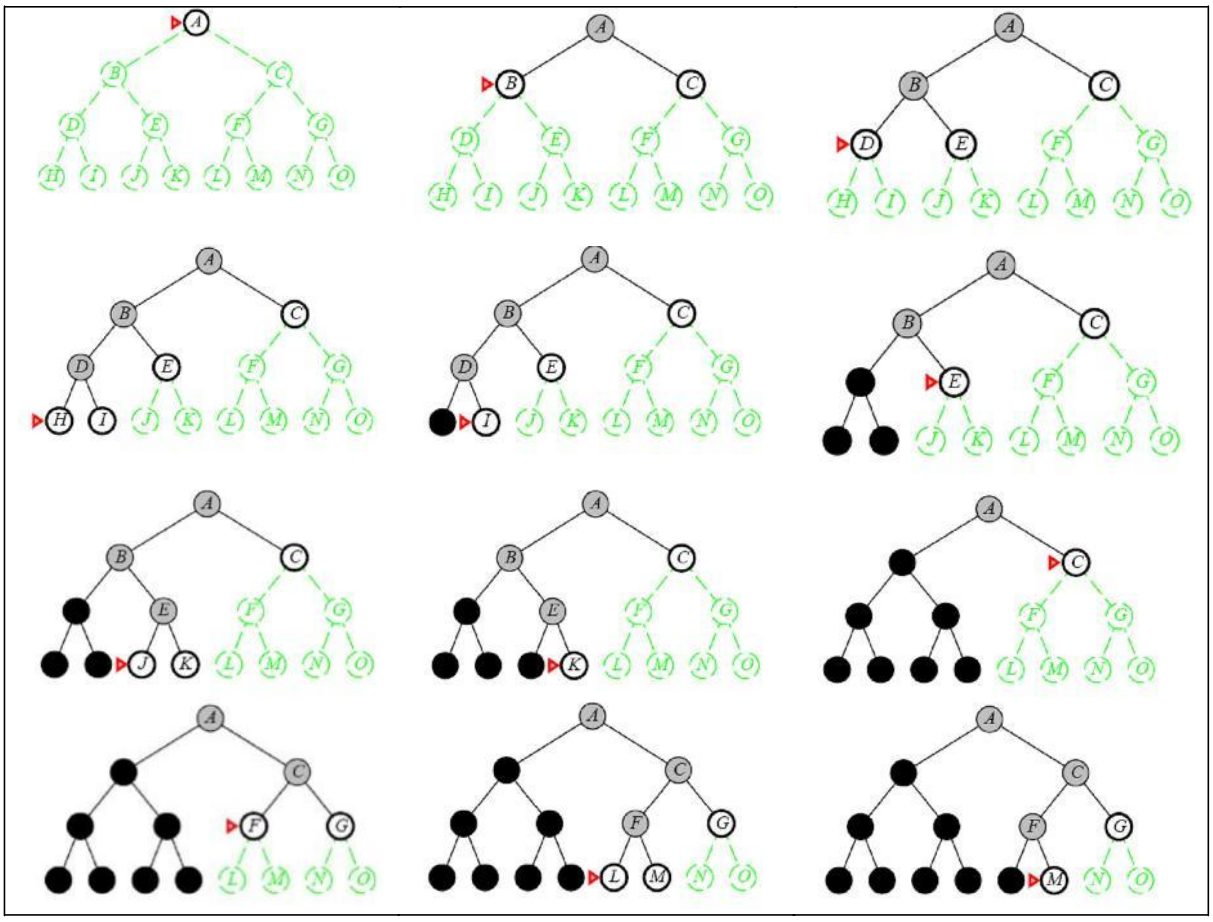
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

DEPTH-FIRST-SEARCH

- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The progress of the search is illustrated in figure.
- The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors.
- As those nodes are expanded,they are dropped from the fringe,
- so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue,also known as a stack.
- Depth-first-search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node,along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded,it can be removed from the memory,as soon as its descendants have been fully explored.
- For a state space with a branching factor b and maximum depth m,depth-first-search requires storage of only bm + 1 nodes.



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

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree.

For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than $O(b^m)$

DEPTH-LIMITED-SEARCH

- The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.
- That is,nodes at depth l are treated as if they have no successors.
- This approach is called **depth-limited-search**.
- The depth limit solves the infinite path problem.
- Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl).
- Depth-first-search can be viewed as a special case of depth-limited search with l = oo
- Sometimes,depth limits can be based on knowledge of the problem.
- For,example,on the map of Romania there are 20 cities.
- Therefore,we know that if there is a solution.,it must be of length 19 at the longest,So l = 10 is a possible choice.
- However,it oocan be shown that any city can be reached from any other city in at most 9 steps.
- This number known as the **diameter** of the state space,gives us a better depth limit.
- Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm.
- The pseudocode for recursive depth-limited-search is shown.
- It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.
- Depth-limited search = depth-first search with depth limit l, returns cut off if any path is cut off by depth limit
- Recursive implementation of Depth-limited-search:

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/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 not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

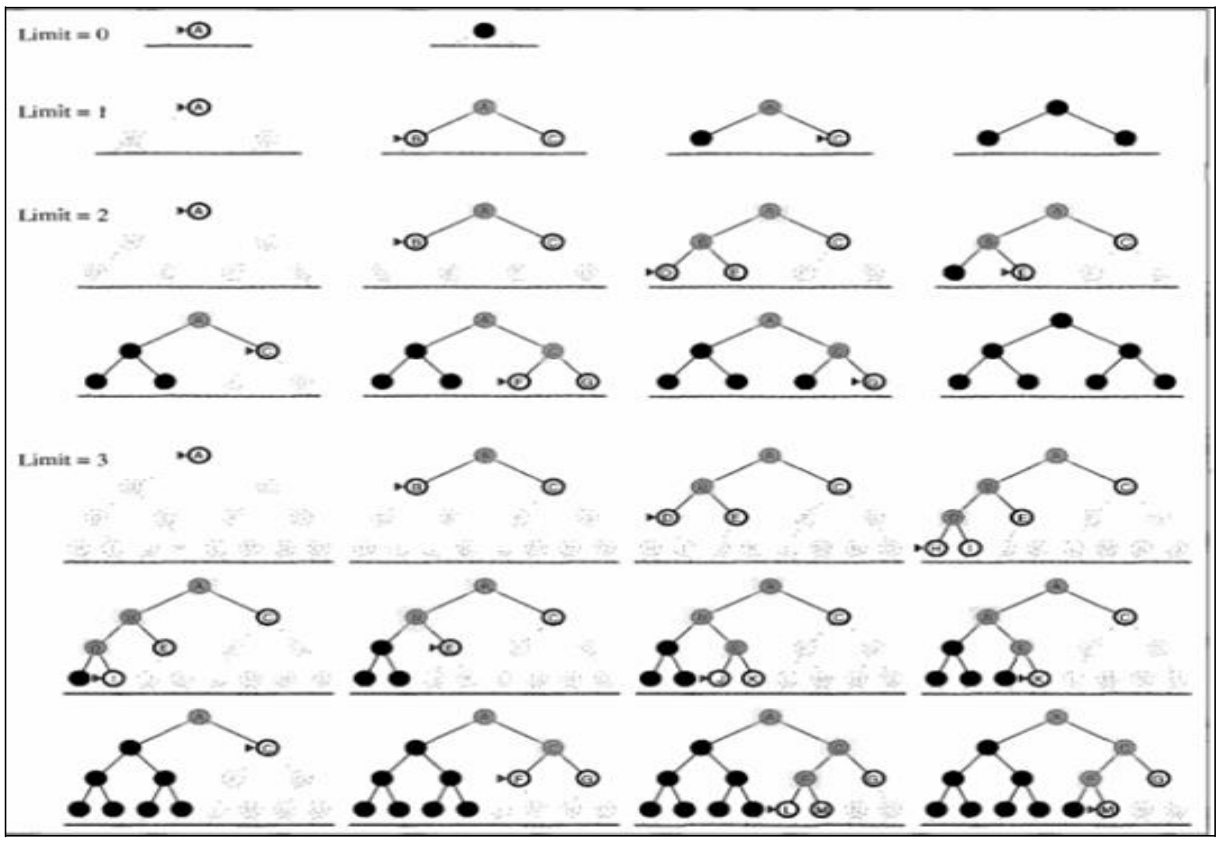
ITERATIVE DEEPENING DEPTH-FIRST SEARCH

- Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search,that finds the better 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.

- Iterative deepening combines the benefits of depth-first and breadth-first-search
- Like depth-first-search,its memory requirements are modest;O(bd) to be precise.
- Like Breadth-first-search,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 the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.

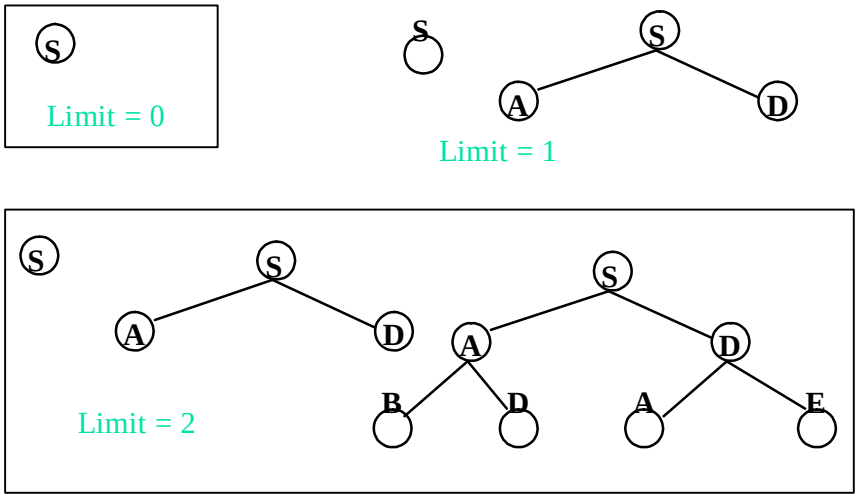
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

The **iterative deepening search algorithm**, which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search returns *failure*, meaning that no solution exists.



Four iterations of iterative deepening search on a binary tree

Iterative deepening search



- Iterative search is not as wasteful as it might seem

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

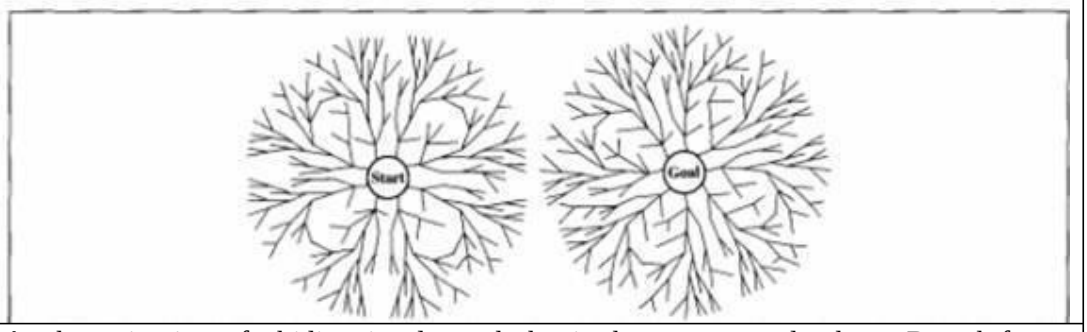
IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

- In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches
  - ✓ one forward from the initial state and
  - ✓ other backward from the goal,
- It stops when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ much less than $b^{d/}$



A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

Comparing Uninformed Search Strategies

The following table compares search strategies in terms of the four evaluation criteria.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Evaluation of search strategies,b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

INFORMED SEARCH AND EXPLORATION

Informed (Heuristic) Search Strategies

- **Informed search strategy** is one that uses problem-specific knowledge beyond the definition of the problem itself.
- It can find solutions more efficiently than uninformed strategy.

Best-first search

- **Best-first search** is an instance of 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 lowest evaluation is selected for expansion,because the evaluation measures the distance to the goal.
- This can be implemented using a priority-queue,a data structure that will maintain the fringe in ascending order of f-values.

Heuristic functions

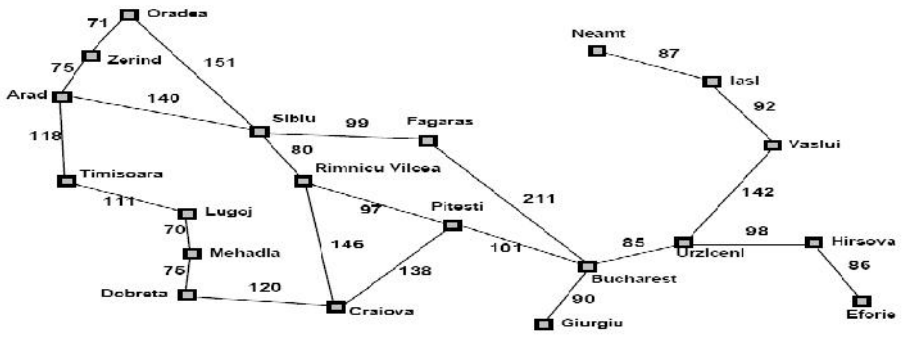
- A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- The key component of Best-first search algorithm is a **heuristic function**,denoted by 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 a **straight-line distance** from Arad to Bucharest
- Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

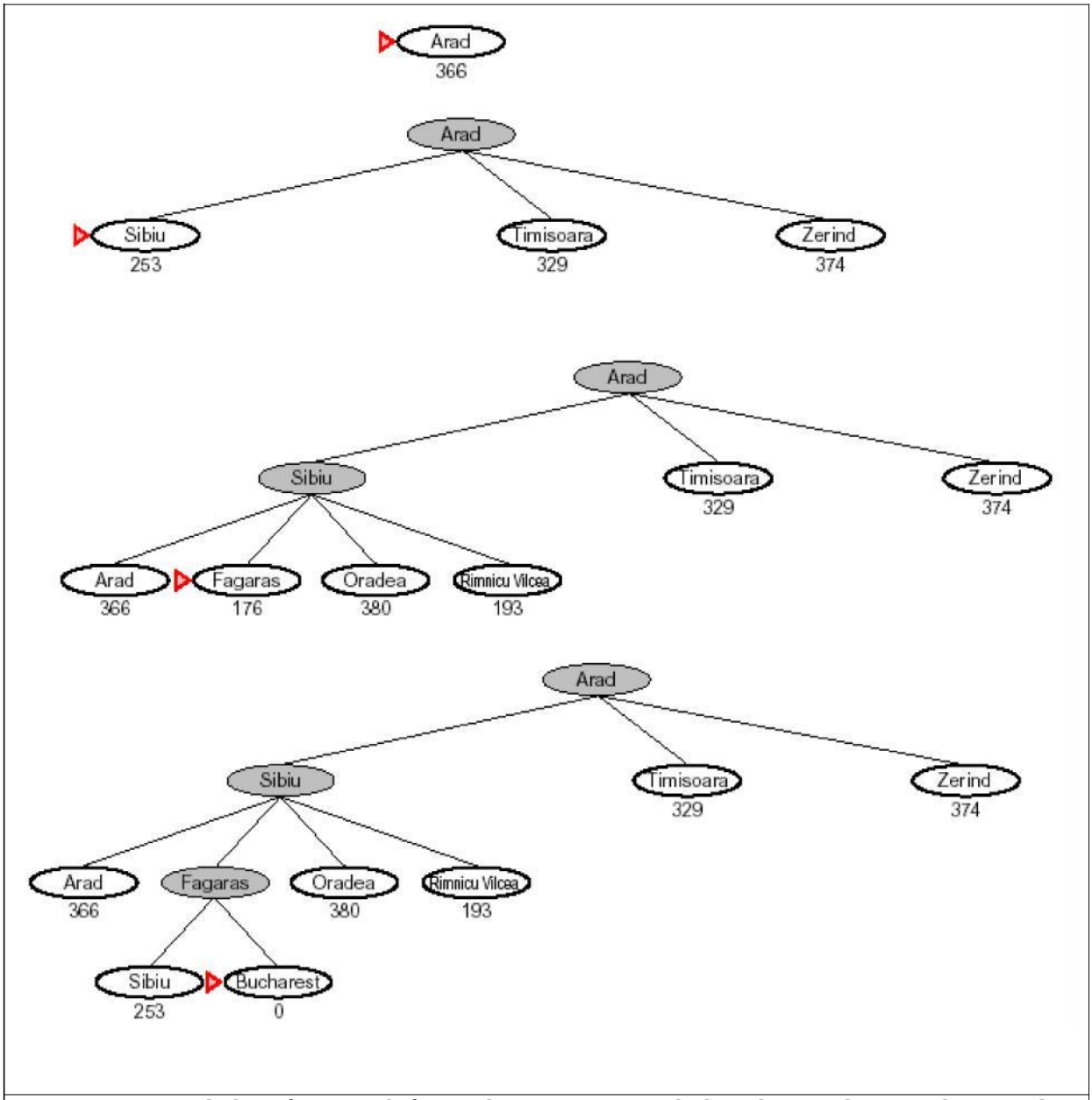
Greedy Best-first search

- **Greedy best-first search** tries to expand the node that is closest to the goal,on the grounds that this is likely to a solution quickly.
- It evaluates the nodes by using the heuristic function f(n) = h(n).
- Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad.
- We need to know the straight-line distances to Bucharest from various cities.
- For example, the initial state is In(Arad) ,and the straight line distance heuristic h_{SLD}(In(Arad)) is found to be 366.
- Vsing the **straight-line distance** heuristic **h_{SLD} ,**the goal state can be reached faster.



Straight line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Siblu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Values of h_{SLD} - straight line distances to Bucharest



Strategies in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- The above figure shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.
- 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.

Properties of greedy search

- **Complete??** No–can get stuck in loops, e.g.,
  Iasi ! Neamt ! Iasi ! Neamt !
  Complete in finite space with repeated-state checking
- **Time??** O(bm), but a good heuristic can give dramatic improvement
- **Space??** O(bm)—keeps all nodes in memory
- **Optimal??** No

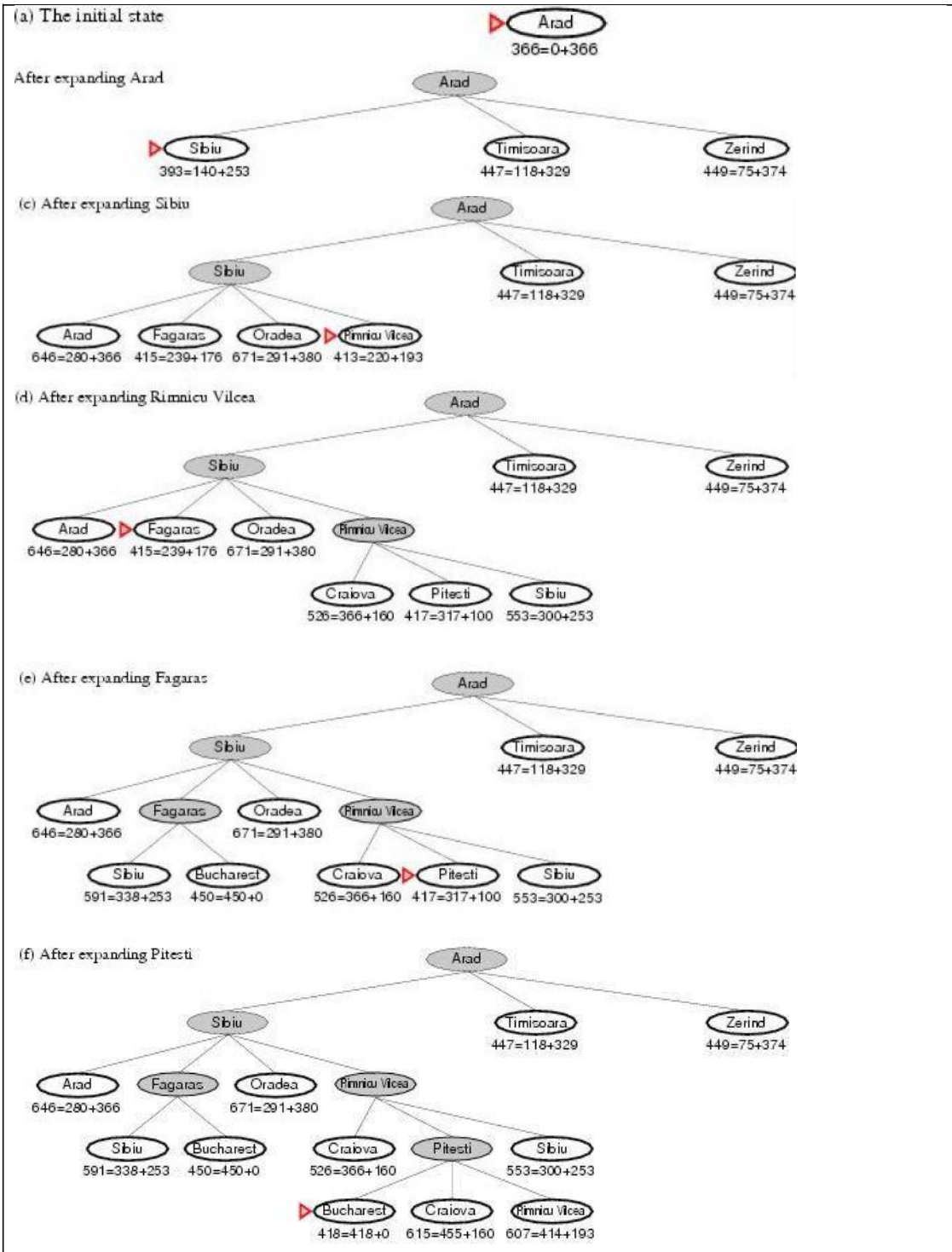
- Greedy best-first search is not optimal,and it is incomplete.
- The worst-case time and space complexity is $O(b^m)$,where m is the maximum depth of the search space.

A* Search

- **A* Search** is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining
  (1) **g(n) =** the cost to reach the node,and
  (2) **h(n) =** the cost to get from the node to the **goal** :
  $$f(n) = g(n) + h(n).$$
- A^* Search is both optimal and complete. A^* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance h_{SLD}.
- It cannot be an overestimate.
- A^* Search is optimal if h(n) is an admissible heuristic – that is,provided that h(n) never overestimates the cost to reach the goal.
- An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest.
- The progress of an A^* tree search for Bucharest is shown in above figure.
- The values of 'g ' are computed from the step costs shown in the Romania,Also the values of h_{SLD} are given in Figure Route Map of Romania.

Recursive Best-first Search (RBFS)

- Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space.
- The algorithm is shown in below figure.
- Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,it keeps track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit,the recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.

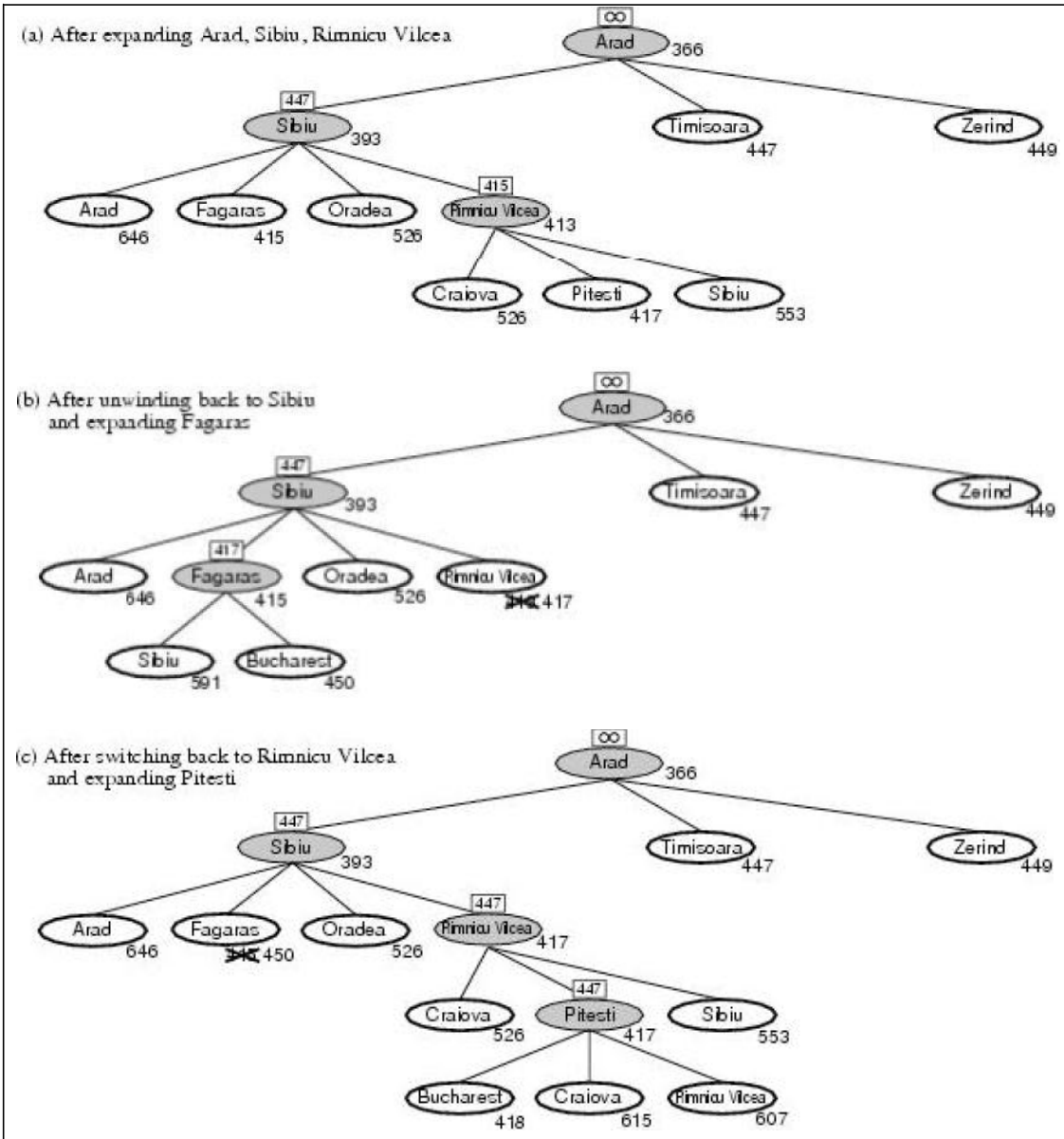


Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure Route map of Romania

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-
    cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f [s] — max(g(s) + h(s), f [node])
    repeat
        best — the lowest f-value node in successors
        if f [best] > f_limit then return failure, f [best]
        alternative — the second lowest f-value among successors
        result, f [best] — RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

The algorithm for recursive best-first search



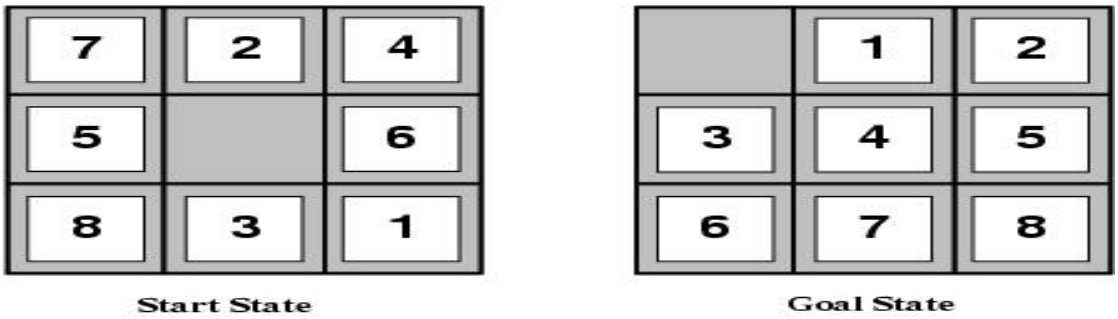
- Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node.
- (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).
- (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.
- (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded.
- This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

RBFS Evaluation:

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, optimal if *h(n)* is admissible
- Space complexity is *O(bd)*.
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
- IDA* en RBFS suffer from ***too little*** memory.

Heuristic Functions

- A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search



A typical instance of the 8-puzzle.

- The solution is 26 steps long.

The 8-puzzle

- The 8-puzzle is an example of Heuristic search problem.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- The average cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three).
- This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1 X 10^{10} states.
- By keeping track of repeated states, we could cut this down by a factor of about 170, 000, because there are only 9!/2 = 181,440 distinct states that are reachable.
- This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}.

- If we want to find the shortest solutions by using A^*,we need a heuristic function that never overestimates the number of steps to the goal.
- The two commonly used heuristic functions for the 15-puzzle are :

  (1) h_1 = the number of misplaced tiles.
- In the above figure all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic.

  (2) h_2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

h_2 is admissible ,because all any move can do is move one tile one step closer to the goal.

- Tiles 1 to 8 in start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

- Neither of these overestimates the true solution cost, which is 26.

The Effective Branching factor

- One way to characterize the **quality of a heuristic** is the **effective branching factor b*.** If the total number of nodes generated by A* for a particular problem is **N**,and the **solution depth** is **d,**then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- For example,if A^* finds a solution at depth 5 using 52 nodes,then effective branching factor is 1.92.
- A well designed heuristic would have a value of b^* close to 1,allowing failru large problems to be solved.
- To test the heuristic functions h_1 and h_2,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with A^* search using both h_1 and h_2.
- The following table gives the average number of nodes expanded by each strategy and the effective branching factor.
- The results suggest that h_2 is better than h_1,and is far better than using iterative deepening search.
- For a solution length of 14,A^* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Comparison of search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* Algorithms with h_1,and h_2. Data are average over 100 instances of the 8-puzzle,for various solution lengths.

Inventing admissible heuristic functions

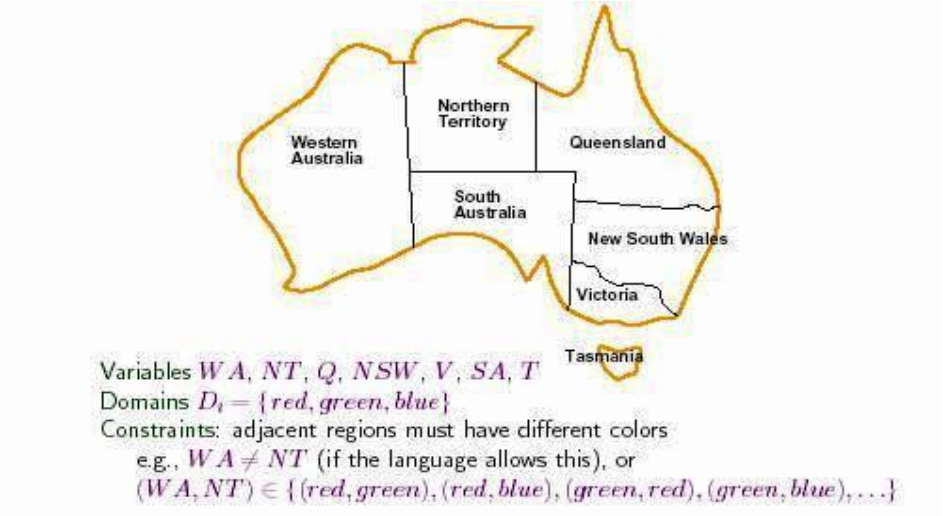
Relaxed problems

- 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 *h2(n)* gives the shortest solution

CONSTRAINT SATISFACTION PROBLEMS (CSP)

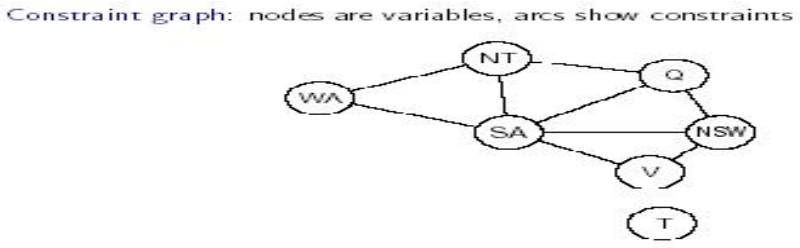
- A **Constraint Satisfaction Problem** (or CSP) is defined by a
  - ✓ set of **variables** X_1, X2….X_n, and a
  - ✓ set of constraints $C_1,C_2,\ldots,C_m$.
  - ✓ Each variable X_i has a nonempty **domain** D,of possible **values**.
  - ✓ Each constraint C_i involves some subset of variables and specifies 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**.
- For Example for Constraint Satisfaction Problem :
- The following 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}.



Principle states and territories of Australia. Coloring this map can be viewed as aconstraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

- It is helpful to visualize a CSP as a constraint graph,as shown in the following figure.
- The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.



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 pairs of variables.

Example: SA # WA

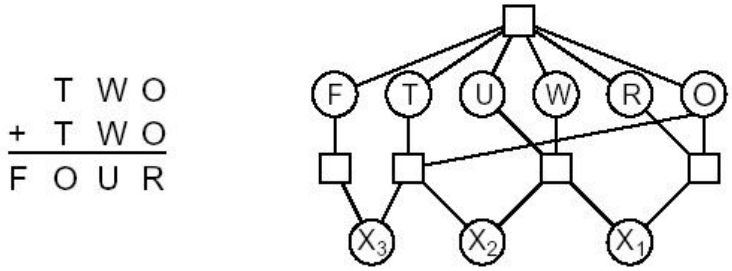
(iii) **Higher order constraints** involve 3 or more variables.

Example: cryptarithmetic puzzles.

(iv) **Absolute constraints** are the constraints, which rules out a potential solution when they are violated

(v) **Preference constraints** are the constraints indicating which solutions are preferred

Example: Vniversity Time Tabling Problem



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
$alldiff(F,T,U,W,R,O)$
$O + O = R + 10 \cdot X_1$, etc.

- Cryptarithmetic problem.
- Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed.
- The constraint hypergraph for the cryptarithmetic problem,showint the *Alldiff* constraint as well as the column addition constraints.
- Each constraint is a square box connected to the variables it contains.

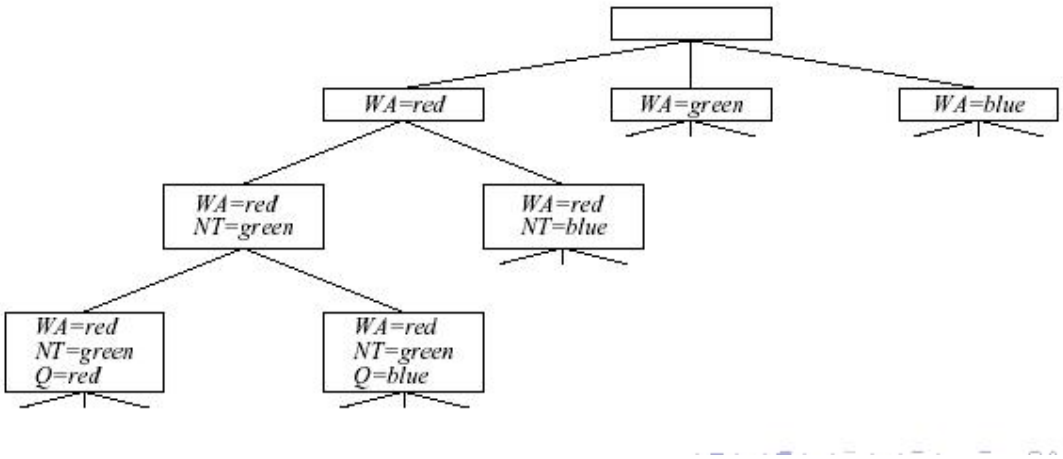
Backtracking Search for CSPs

- 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 following algorithm shows the Backtracking Search for CSP

```
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
```

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



Part of search tree generated by simple backtracking for the map coloring problem.

Propagating information through constraints

- So far our search algorithm considers the constraints on a variable only at the time that the Variable is chosen by SELECT-VNASSIGNED-VARIABLE.
- But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

- One way to make better use of constraints during search is called **forward checking**.
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X.
- The following figure shows the progress of a map-coloring search with forward checking.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

Constraint propagation

- Although forward checking detects many inconsistencies, it does not detect all of them.
- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.

Arc Consistency

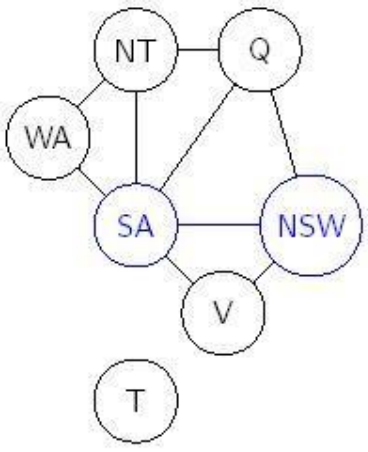


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value x of *SA*
  - There is some value y of *NSW* that is consistent with x
- Examine arcs for consistency in *both* directions

K-Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is ***k*-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the k-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

The Structure of Problems

Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Sub problems

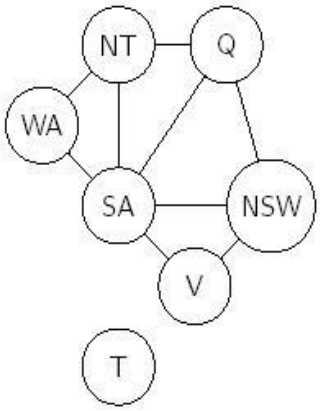


Figure: Australian Territories

- T is not connected
- Coloring T and coloring remaining nodes are **independent subproblems**
- *Any* solution for T combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

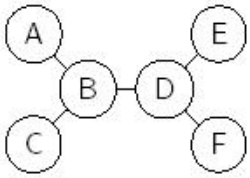


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

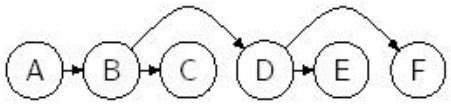


Figure: Linear ordering

****

FIRST UNIT-I PROBLEM SOLVING FINISHED

****

ARTIFICIAL INTELLIGENCE

UNIT-II LOGICAL

REASONING

Logical Agents – propositional logic – inferences – first-order logic – inferences in first-order logic – forward chaining- backward chaining – unification – resolution

2.0 Logic:

- A knowledge representation language in which syntax and semantics are defined correctly is known as logic.
- A formal language to represent the knowledge in which reasoning is carried out to achieve the goal state.
- Logics consists of the following two representations in sequence,
  - A formal system is used to describe the state of the world
    - ✓ Syntax " Which describes how to make sentences"
    - ✓ Semantics " Which describes the meaning of the sentences"
  - The proof theory "a set of rule for deducing the entailments of a set of sentences.
- We will represent the sentences using two different logics, They are,
  - Propositional Logic (or) Boolean logic
  - Predicate logic (or) First order logic

2.1 Logical Agents:

- The Logical agent has to perform the following task using logic representation. The tasks are,
  - ✓ To know the current state of the world
  - ✓ How to infer the unseen properties of the world
  - ✓ New changes in the environment
  - ✓ Goal of the agent

✓ How to perform actions depends on circumstances

2.2 Propositional Logic:

- Each fact is represented by one symbol.
- Proposition symbols can be connected with Boolean connectives, to give more complex meaning. Connectives ,
  - **Λ** Logical Conjunction
  - **V** Logical disjunction
  - ¬ Negation
  - ⇔ Material Equivalence or Biconditional
  - ⇒ Material Implication or conditional
- Simple statements are implemented
- The Symbols of propositional logic are the logical constants, (True and False)
- For Example: - P, Q

Connectives

Λ (and) ----- **Example: - P Λ Q**
V (or) ----- **Example: - P V Q**
⇒ (implies) ----- **Example: - (P Λ Q) ⇒ R**
⇔ (equivalent) ----- **Example: - (P Λ Q) ⇔ (Q Λ P)**
¬ (not) ----- **Example: - ¬ P**

- A **BNF** (Backus-Naur Form) grammer of sentence in propositional logic

Sentence ---------> **Atomic sentence | complex sentence**
Atomic sentence ---------> **True | False | P | Q | R |**
Complex sentence ---------> **(sentence) | sentence connective sentence | ¬ Sentence**
Connective ---------> **Λ | V | ⇒ | ⇔**

- Order of precedence (from highest to lowest) : **¬ , Λ , V , ⇒ and ⇔**
- Example : - ¬P VQ ΛR ⇒ S is equivalent to ((¬P)V(Q Λ R)) ⇒ S
- The following truth table shows the five logical connectives

P	Q	¬ P	P Λ Q	P V Q	P ⇒ Q	P ⇔ Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

- These truth table can be used to define the validity of a sentence.
- If the sentence is true in every row (i.e. for different types of logical constants) then the sentence is a valid sentence.
- For Example: - ((P V H) Λ¬H) ⇒ P Check whether the given sentence is a valid sentence or not.

P	H	P V H	(P V H) Λ¬H	((P V H) Λ¬H) ⇒ P
False	False	False	False	True
False	True	True	False	True
True	False	True	True	True
True	True	True	False	True

- The given sentence is ((P V H) Λ¬H) ⇒ P valid sentence, because the sentence is TRUE in every row for different types of logical statements.

2.3 Inference Rules for Propositional logic:

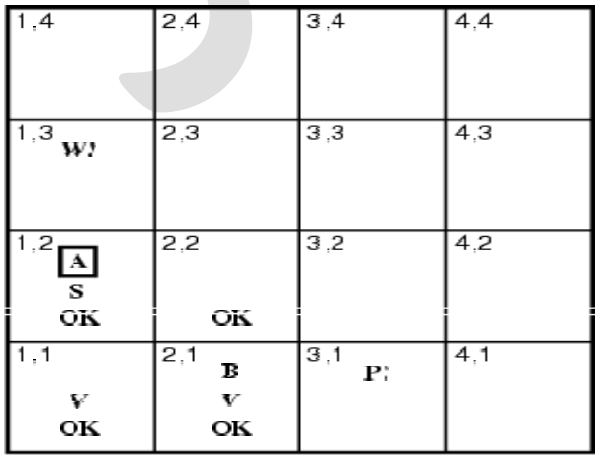
- The propositional logic has seven inference rules.
- Inference means conclusion reached by reasoning from data or premises; speculation.
- A procedure which combines known facts to produce ("infer") new facts.
- **Logical inference** is used to create new sentences that logically follow from a given set of predicate calculus sentences (KB).
- An inference rule is **sound** if every sentence X produced by an inference rule operating on a KB logically follows from the KB. (That is, the inference rule does not create any contradictions)
- An inference rule is **complete** if it is able to produce every expression that logically follows from (is entailed by) the KB. (Note the analogy to complete search algorithms.)
- Here are some examples of sound rules of inference

  A rule is sound if its conclusion is true whenever the premise is true

Rule	Premise	Conclusion
Modus Ponens	A, A → B	B
And Introduction	A, B	A ∧ B
And Elimination	A ∧ B	A ∨ B
Or Introduction	A, B	A
Double Negation	¬¬A	A
Unit Resolution	A ∨ B, ¬B	A
Resolution	A ∨ B, ¬B ∨ C	A ∨ C

2.4 An Agent for the Wumpus world – Propositional logic

- We will discuss the knowledge base representation and a method to find the wumpus using propositional logic representation.
- From the following figure assume that the agent has reached the square (1,2)



- **The Knowledge Base:** The agent percepts are converted into sentences and entered into the knowledge base, with some valid sentences that are entailed by the percept sentences
- From the above figure we can perceive the following percept sentences and it is added to the knowledge base.

$\neg S_{1,1}$	$\neg B_{1,1}$	-----	for the square (1, 1)
$\neg S_{2,1}$	$\neg B_{2,1}$	-----	for the square (2, 1)
$S_{1,2}$	$\neg B_{1,2}$	-----	for the square (1, 2)
$S_{1,2}$		-----	There is a stench in (1, 2)
$\neg B_{1,2}$		-----	There is a breeze in (1, 2)
$\neg S_{2,1}$		-----	There is a stench in (2, 1)
$B_{2,1}$		-----	There is a breeze in (2, 1)
$\neg S_{1,1}$		-----	There is a stench in (1, 1)
$\neg B_{1,1}$		-----	There is a breeze in (1, 1)

- The rule of three squares,
  - R_{1} :- $\neg S_{1,1} \Rightarrow \neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,1}$
  - R_{2}:- $\neg S_{2,1} \Rightarrow \neg W_{1\,1} \wedge \neg W_{2\,2} \wedge \neg W_{2\,1} \wedge \neg W_{3\,1}$
  - R_{3}:- $\neg S_{1,2} \Rightarrow \neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,2} \wedge \neg W_{1\,3}$
  - R_{4}:- $\neg S_{1,2} \Rightarrow W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1}$

- Finding the wumpus as, We can prove that the Wumpus is in (1, 3) using the four rules given.
  - Apply Modus Ponens with $\neg S_{1\,1}$ and R_1:
    - $\neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,1}$
  - Apply And-Elimination to this, yielding three sentences:
    - $\neg W_{1\,1}, \neg W_{1\,2}, \neg W_{2\,1}$
  - Apply Modus Ponens to $\neg$S21 and R2, then apply And-elimination:
    - $\neg W_{2\,2}, \neg W_{2\,1}, \neg W_{3\,1}$
  - Apply Modus Ponens to S12 and R4 to obtain:
    - $W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1}$
  - Apply Unit resolution on $(W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1})$ and $\neg W_{1\,1}$:
    - $W_{1\,3} \vee W_{1\,2} \vee W_{2\,2}$
  - Apply Unit Resolution with $(W_{1\,3} \vee W_{1\,2} \vee W_{2\,2})$ and $\neg W_{2\,2}$:
    - $W_{1\,3} \vee W_{1\,2}$
  - Apply Unit Resolution with $(W_{1\,3} \vee W_{1\,2})$ and $\neg W_{1\,2}$:
    - $W_{1\,3}$

2.5 First-Order Logic:

- **First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our commonsense knowledge.
- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**

2.5.1 Representation Revisited:

- It is necessary to know about the nature of representation languages.
- The following are the some languages,
  - ✓ Programming languages
  - ✓ Propositional logic languages
  - ✓ Natural languages

Programming languages:

- Programming languages like C++ or Java are the largest class of formal languages in common use.
- Programs represent only computational processes.
- Data structures within programs can represent facts.
- For Example, 4 x 4 arrays can be used by a program to represent the contents of the Wumpus world.
- Thus the programming language statement *World [2, 2]* ← *Pit* is a fairly natural way to assert that there is a pit in square [2, 2].

Disadvantages:

- Programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain.
- A second drawback of data structures in programs is the lack of any easy way to say
- For Example, "There is a Pit in [2,2] or [3,1]" or "If the Wumpus is in [I,I] then he is not in [2,2]".
- Programs lack the expressiveness required to handle partial information.

Propositional Logic Languages:

- Propositional logic is a declarative language
- The following are the properties of propositional logic
- Its semantics is based on a truth relation between sentences and possible worlds.
- It also has sufficient expressive power to deal with partial information, using disjunction and negation.

- It also has **compositionality** that is desirable in representation languages namely **compositionality.**
- In a **compositionality** language, the meaning of a sentence is function of the meaning of its parts.
- For Example, "$S_{1,4}$ **Λ** $S_{1,2}$" is related to the meanings of "$S_{1,4}$" and "$S_{1,2}$"
- It would be very strange if "$S_{1,4}$" meant that there is a stench in square [1,4] and "$S_{1,2}$" meant that there is a stench in square [1,2], but "$S_{1,4}$ **Λ** $S_{1,2}$" meant that France and Poland drew 1-1 in last week's ice hockey qualifying match.
- Clearly, non Compositionality makes life much more difficult for the reasoning system.

Advantages:

- Declarative
- Context-Independent
- Unambiguous

Disadvantages:

- It lacks the expressive power to describe an environment with many objects concisely.
- For Example, it is forced to write a separate rule about breezes and pits for each square, such as $B_{1,1} \Leftrightarrow (P_{1,2} \text{ V } P_{2,1})$.
- The procedural approach of programming languages can be contrasted with the declarative nature of propositional logic, in which knowledge and inference are separate and inference is entirely domain-independent.

Natural Languages:

- A moments thought suggests that natural languages like English are very expressive indeed.
- Natural language is essentially a declarative knowledge representation language and attempts to pin down its formal semantics.
- The modern view of natural language is that it serves a somewhat different purpose, namely as a medium for communication rather than pure representation.
- When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops.
- The meaning of the above sentence depends both on the sentence itself and on the context in which the sentence was spoken.

Disadvantages:

- It is difficult to understand how the context can be represented.
- This is because one could not store a sentence such as "Look!" in knowledge base and expect to recover its meaning without also storing a representation of the context.
- They are also non-compositional
- They suffer from ambiguity, which would cause difficulties for thinking.
- For Example, when people think about spring, they are not confused as to whether they are thinking about a season or something that goes *boing-and* if one word can correspond to two thoughts, thoughts can't be words.

2.5.2 First-Order Logic:

- **First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our commonsense knowledge.
- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**
- **FOL** adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks
- The Syntax of natural language contains elements such as,

  Nouns and noun phrases that refer to objects (Squares, pits, rumpuses)

  Verbs and verb phrases that refer to among objects (is breezy, is adjacent to)
- Some of these relations are functions-relations in which there is only one "Value" for a given "input".
- For Example,

  Objects: People, houses, numbers

  Relations: These can be unary relations or properties such as red, round,

  More generally n-ary relations such as brother of, bigger than,

  Functions: father of, best friend,…
- Indeed, almost any assertion can be thought of as referring to objects and properties or relations
- For Example, in the way of Sentence " One plus Two is Three"
- Where,

Objects: One, Two, Three, One plus Two

Relations: equals

Function: plus

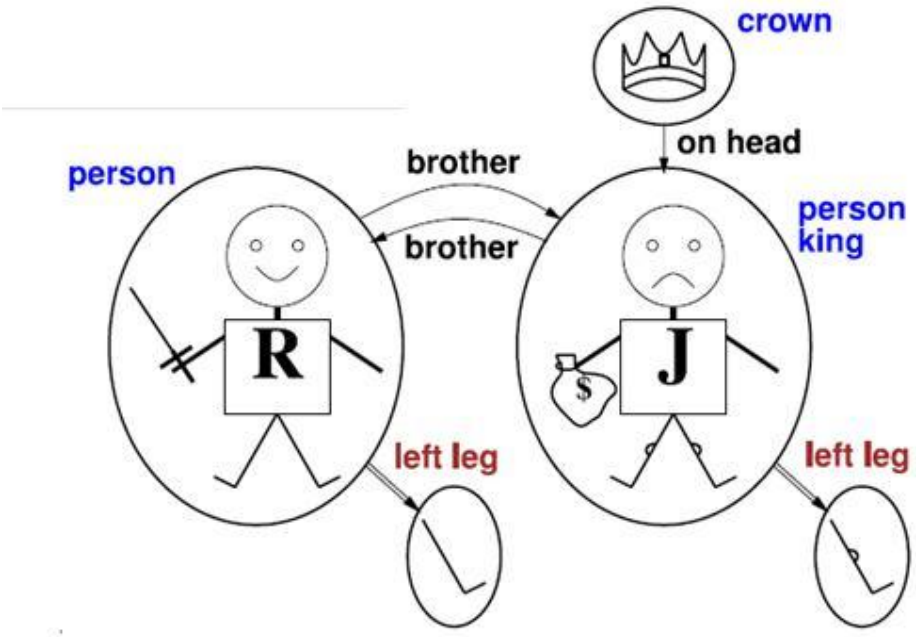
- Ontological commitment of First-Order logic language is "Facts", "Objects", and "Relations".
- Where ontological commitment means "WHAT EXISTS IN THE WORLD".
- Epistemological Commitment of First-Order logic language is "True", "False", and "Unknown".
- Where epistemological commitment means "WHAT AN AGENT BELIEVES ABOUT FACTS".

Advantages:

- It has been so important to mathematics, philosophy, and Artificial Intelligence precisely because those fields can be usefully thought of as dealing with objects and the relations among them.
- It can also express facts about some or all of the objects in the universe.
- It enables one to represent general laws or rules, such as the statement "Squares neighboring the wumpus are smelly".

2.5.3 Syntax and Semantics of First-Order Logic:

- The models of a logical language are the formal structures that constitute the possible worlds under consideration.
- Models for propositional logic are just sets of truth values for the proposition symbols.
- Models for first-order logic are more interesting.
- First they have objects in them.
- The domain of a model is the set of objects it contains; these objects are sometimes called domain elements.
- The following diagram shows a model with five objects



- The five objects are,
  - ✓ Richard the Lionheart
  - ✓ His younger brother
  - ✓ The evil King John
  - ✓ The left legs of Richard and John
  - ✓ A crown
- The objects in the model may be related in various ways, In the figure Richard and John are brothers.
- Formally speaking, a relation is just the set of tuples of objects that are related.
- A tuple is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- Thus, the brotherhood relation in this model is the set

  {(Richard the Lionheart, King John),(King John, Richard the Lionheart)}
- The crown is on King John's head, so the "on head" relation contains just one tuple, (the crown, King John).
- The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

- Certain kinds of relationships are best considered as functions that relates an object to exactly one object.
- For Example:- each person has one left leg, so the model has a unary "left leg" function that includes the following mappings

  (Richard the Lionheart) ----> Richard's left leg

  (King John) ----> John's left leg

➢ **Symbols and Interpretations:**

- The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations** and **functions**

❖ **Kinds of Symbols**

- The symbols come in three kinds namely,
  - ✓ Constant Symbols standing for **Objects** (Ex:- Richard)
  - ✓ Predicate Symbols standing for **Relations** (Ex:- King)
  - ✓ Function Symbols stands for **functions** (Ex:- LeftLeg)
    - o Symbols will begin with uppercase letters
    - o The choice of names is entirely up to the user
    - o Each predicate and function symbol comes with an arity
    - o Arity fixes the number of arguments.
- The semantics must relate sentences to models in order to determine truth.
- To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.
- One possible interpretation called as the intended interpretation- is as follows;
  - ✓ **Richard** refers to **Richard the Lionheart** and **John** refers to the **evil King John.**
  - ✓ **Brother** refers to the brotherhood relation, that is the set of tuples of objects given in equation **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
  - ✓ **OnHead** refers to the "on head" relation that holds between the crown and King John; **Person, King** and **Crown** refer to the set of objects that are persons, kings and crowns.
  - ✓ **Leftleg** refers to the "left leg" function, that is, the mapping given in **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**

- There are many other possible interpretations relating these symbols to this particular model.
- The truth of any sentence is determined by a model and an interpretation for the sentence symbols.
- The syntax of the FOL with equality specified in BNF is as follows

Sentence	⟶	AtomicSentence
	|	(Sentence Connective Sentence)
	|	Quantifier Variable,…Sentence
	|	- Sentence
AtomicSentence	⟶	Predicate (Term…) | Term = Term
Term	⟶	Function (Term,….)
	|	Constatnt
	|	Variable
Connective	⟶	**⇒ | Λ | V | ⇔**
Quantifier	⟶	**∀ | ∃**
Constant	⟶	**A | X_1 | John**
Variable	⟶	**a | x | s | …**
Predicate	⟶	**Before | HasColor | Raining | ….**
Function	⟶	**Mother | LeftLeg | ….**

- Where,

Λ	Logical Conjunction
V	Logical disjunction
∀	Universal Quantification
∃	Existential Quantification
⇔	Material Equivalence
⇒	Material Implication

❖ **Terms:**

- A Term is a logical expression that refers to an object
- Constant symbol are therefore terms, but it is not always convenient to have a distinct symbol to name every object.
- For Example:- in English we might use the expression “King Johns left leg” rather than giving a name to his leg.

- A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.
- It is just like a complicated kind of name. It's not a "subroutine call" that returns a value".
- The formal semantics of terms is straight forward,

  Consider a term $f(t1……t_n)$

  Where

  f- some function in the model (call it F)

  The argument terms – objects in the domain

  The term – object that is the value of the function F applied to the domain

- For Example:- suppose the **LeftLeg** function symbol refers to the function is,

  (Richard the Lionheart) ------ Richards left leg

  (King John) ------ Johns left leg

  John refers to King John, then **LeftLeg** (John) refers to king Johns left leg.

- In this way the Interpretation fixes the referent of every term.

❖ **Atomic Sentences:-**

- An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms:

  Brother (Richard, John)

- This states that Richard the Lionheart is the brother of King John.
- Atomic Sentences can have complex terms as arguments.
- Thus, **Married(Father(Richard), Mother(John))** states that Richard the Lionheart's father is married to King John's mother
- An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

❖ **Complex Sentences:-**

- Logical connectives can be used to construct more complex sentences, just as in propositional calculus.
- The semantics of sentences formed with logical connectives is identical to that in the propositional case.

¬ Brother (LeftLeg (Richard), John)

Brother (Richard, John) Λ Brother (John, Richard)

King (Richard) V King (John)

¬ King (Richard) ⇒ King (John)

¬ **it refers " Logical Negation"**

❖ **Quantifiers:-**

- Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found.
- It has two type,
- The following are the types of standard quantifiers,

  ✓ Universal
  ✓ Existential

❖ **Universal Quantification (∀):-**

- Universal Quantification make statement about every object.
- "All Kings are persons", is written in first-order logic as

  $\forall_x$ king (x) ⇒ Person (x)

- ∀ is usually pronounced "For all….", Thus the sentences says , "For all x, if x is a king, then x is a person".
- The symbol x is called a variable.
- A variable is a term all by itself, and as such can also serve as the argument of a function-for example, **LeftLeg(x)**.
- A term with no variables is called a **ground term.**
- Based on our model, we can extend the interpretation in five ways,

  x --------- Richard the Lionheart

  x --------- King John

  x --------- Richard's Left leg

  x --------- John's Left leg

  x --------- the crown

- The universally quantified sentence is equivalent to asserting the following five sentences

  Richard the Lionheart ------ Richard the Lionheart is a person

King John is a King ------ King John is a Person

Richard's left leg is King -------- Richard's left leg is a person

John's left leg is a King -------- John's left leg is a person

The crown is a King -------- The crown is a Person

❖ **Existential Quantification (∃):-**

- An existential quantifier is used make a statement about some object in the universe without naming it.
- To say, for example :- that King John has a crown on his head, write ∃x crown (x) Λ OnHead (x, John).
- ∃x is pronounced " There exists an x such that.," or "For some x.."
- Consider the following sentence,

$$\exists_x \textbf{ crown (x) } \Rightarrow \textbf{ OnHead (x,John)}$$

- Applying the semantics says that at least one of the following assertion is true,

Richard the Lionheart is a crown	Λ Richard the Lionheart is on John's head
King John is Crown	Λ King John is on John's head
Richard's left leg is a crown	Λ Richard's left leg is on John's head
John's left leg is a crown	Λ John's left leg is on John's head
The crown is a crown	Λ The crown is on John's head

- Now an implication is true if both premise and conclusion are true, or if its premise is false.

❖ **Nested Quantifiers:-**

- More complex sentences are expressed using multiple quantifiers.
- The following are the some cases of multiple quantifiers,
- The simplest case where the quantifiers are of the same type.
- For Example:- "Brothers are Siblings" can be written as

$$\forall_x \forall_y \textbf{ Brother (x,y) } \Rightarrow \textbf{ sibling (x,y)}$$

- Consecutive quantifiers of the same type can be written as one quantifier with several variables.
- For Example:- to say that siblinghood is a symmetric relationship as

$$\forall_{x,y} \textbf{ sibling (x,y) } \Leftrightarrow \textbf{ sibling (y,z)}$$

- In some cases it is possible to have mixture of quantifiers.
- For Example:- "Everybody loves somebody" means that for every person, there is someone that person loves:

$\forall_x \exists_y$ Loves (x, y).

- On the other hand , to say "There is someone who is loved by everyone", we can write as

$\exists_y \forall_x$ Loves (x, y).

❖ **Connections between ∀ and ∃**

- The two quantifiers are actually intimately connected with each other , through negation,
- Declaring that everyone dislikes parsnips is the same as declaring there does not exist someone who likes them, and vice versa:

$\forall_x \neg$ Likes (x, Parsnips) is equivalent to $\neg \exists_x$ Likes (x, Parsnips)

- Going one step further: "Everyone likes ice cream" means that there os no one who does not like ice cream:

$\forall_x \neg$ Likes (x, IceCream) is equivalent to $\neg \exists_x$ Likes (x, IceCream)

- Because ∀ is really a conjunction over the universe of objects ∃ is a disjunction, it should not be surprising that they obey de Morgan's rules.
- The de Morgan's rules for quantified and un-quantified sentences are as follows:
- **≡ it refers definition**

$\forall_x \neg P \equiv \neg \exists_x P$	$\neg P \wedge \neg Q \equiv \neg (P \vee Q)$
$\neg \forall_x P \equiv \exists_x \neg P$	$\neg (P \wedge Q) \equiv \neg P \vee \neg Q$
$\forall_x P \equiv \neg \exists_x \neg P$	$P \wedge Q \equiv \neg (\neg P \vee \neg Q)$
$\exists_x P \equiv \neg \forall_x \neg P$	$P \vee Q \equiv \neg (\neg P \wedge \neg Q)$

❖ **Equality:-**

- First – order logic includes one more way of using equality symbol to make atomic sentences.
- Use of equality symbol
  - ✓ The equality symbol can be used to make statements to the effect that two terms refer to the same object.
  - ✓ For Example: - Father (John) = Henry says that the object referred to by Father (John) and the object referred to by Henry are the same.
  - ✓ Determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.
  - ✓ The equality symbol can be used to state facts about a given function

- ✓ It can also be used with negation to insist that two terms are not the same object.
- ✓ For Example:- To say that Richard has at least two brothers, write as

$\exists_{x,y}$ Brother(x, Richard) Λ Brother(y,Richard) $\Lambda \neg$ (x = y)

2.5.4 Using First – Order Logic

The best way to learn about FOL is through examples. In Knowledge representation, a domain is just some part of the world about which some knowledge is to be expressed.

❖ **Assertions:-**

- Sentences that are added to knowledge base using **TELL**, exactly as in propositional logic are called assertion (Declaration/Statement).
- For Example:- It can be declared that " John is a King and that Kings are persons"

TELL (KB, King(John))
TELL (KB, $\forall_x$ King(x) $\Rightarrow$ Person(x))

❖ **Queries:-**

- Questions of the knowledge base can be asked using **ASK.**
- For Example:- **ASK(KB, King(John))** returns **true**.
- Questions asked using **ASK** are called queries or goals.
- Generally speaking, any query that is logically needed by the knowledge base should be answered positively.
- For Example:- Given the two assertions in the preceding line, the query

ASK(KB, Person(John) should also return **true**

❖ **Substitution/Binding List:-**

- Substitution or Binding list is a set of variable/term pairs.
- It is a standard form for an answer of a query with existential variables.
- For Example:- "Is there an x such that…" is solved by providing such an x.
- Given Just the two assertions, the answer would be {x/John}
- If there is more than one possible answer, a list of substitutions can be returned.

❖ **The Kinship Domain:-**

- Kinship domain is the domain of family relationships, or Kinship.
- This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent".
- The objects in this domain are people.
- There will be two unary predicates as **"Male"** and **"Female"**
- Kinship relations will be represented by binary predicates.

- For Example:- parenthood, brotherhood, marriage and so on are represented by Parent, sibling, Brother, Sister, Child, Daughter,Son,Spouse,Wife, Husband, Grandparents, Grandchild, Cousin, Aunt, and Uncle.
- For Example:-
- **One's mother is One's female parent:**

$$\forall m, c \ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c)$$

❖ **Axioms:-**

- Axioms are commonly associated with purely mathematical domains.
- The axioms define, the Mother function and the Husband, Male, Parent and Sibling predicates in terms of other predicates.
- They provide the basic factual information from which useful conclusions can be derived.
- Kinship axioms are also definitions : they have the form $\forall_{x,y}$ **p(x,y)** $\Leftrightarrow$ ...

❖ **Theorems:-**

- Not all logical sentences about a domain are axioms.
- Some are Theorems-that is, they are caused by the axioms.
- For Example:-

$$\forall_{x,y} \ Sibling(x,y) \Leftrightarrow ... Sibling(y,x)$$

- The above declaration that siblinghood is symmetric
- It's a theorem that follows logically from the axiom that defines siblinghood.
- If ASK Questions the knowledge base this sentence, it should return true
- From logical point of view, a knowledge base need contain only axioms and no theorems
- From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences.

❖ **Numbers:-**

- Numbers are perhaps the most brilliant example of how a large theory can be built up from a tiny heart of axioms.
- Requirements
  - ✓ A predicate ***NatNum*** is needed that will be true of natural numbers
  - ✓ One constant symbol, 0
  - ✓ One function symbol, S (Successor)

- **Peano Axioms:-**
  - ✓ The peano axioms define natural numbers and addition.
  - ✓ Natural numbers are defined recursively:

NatNum (0)

$\forall_n$ NatNum (n) ⇒ NatNum (S(n))

- ✓ That is, 0 is a natural number, and for every object n, if n is a natural number then S(n) is a natural number,
- ✓ So the natural numbers are 0, S(0), S(S(0)), and so on..

❖ **Sets:-**

- The domain of sets is also fundamental to mathematics as well as to commonsense reasoning
- The empty set is a constant written as { }.
- There is one unary predicate, Set, which is true of sets.
- The binary predicates are x € s (**x** is a member of set **s**) and s1 ⊆ s2 (set s1 is a subset, not necessarily proper, of set s2)
- The binary functions are s1 ∩ s2 (the intersection of two sets), s1 ∪ s2 (the union of two sets), and {x/s} (the set resulting from adjoining element x to set **s**)
- One possible set of axioms is as follows,

✓ The only sets are the empty set and those made by adjoining something to a set

$$\forall_s \text{Set}(s) \Leftrightarrow (s = \{\}) \vee (\exists_{x,s2} \text{Set}(s2) \wedge s = \{x/s2\})$$

✓ There is no way to decompose EmptySet into a smaller set and an element:

$$\neg \exists_{x,s} \{x/s\} = \{\}$$

✓ Adjoining an element already in the set has no effect:

$$\forall_{x,s} \quad x \in (\text{set membership}) \; s \Leftrightarrow s = \{x/s\}$$

✓ The only members of a set are the elements that were connected into it. This can be expressed recursively, saying that **x** is a member of **s** if and only if **s** is equal to some set S2 connected with some element **y**, where either **y** is the same as **x** or **x** is a member of S2

$$\forall_{x,s} \quad x \in s \Leftrightarrow [\exists_{y,s2} (s=\{y/s2\} \wedge (x=y \vee x \in s2))]$$

✓ A set is subset of another set if and only if all of the first sets members are members of the second set

$$\forall_{s1,s2} \quad s1 \subseteq s2 \Leftrightarrow (\forall_x x \in s1 \Rightarrow x \in s2)$$

✓ Two sets are equal if and only if each is a subset of the other

$$\forall_{s1,s2} \quad (s1 = s2) \Leftrightarrow (s1 \subseteq s2 \wedge s2 \subseteq s1)$$

- ✓ An object is in the Intersection of two sets if and only if it is a member of both sets

$$\forall_{x, s1, s2} \quad x \in (s1 \cap s2) \quad (x \in s1 \ \Lambda \ x \in s2)$$

- ✓ An object is in the union of two sets if and only if it is a member of either set

$$\forall_{x, s1, s2} \quad x \in (s1 \cup s2) \quad (x \in s1 \ V \ x \in s2)$$

❖ **Lists:-**

- Lists are similar to sets.
- The differences are that lists are ordered and the same element can appear more than once in a list.
- **Nil** is the constant list with no elements
- **Cons, Append, First,** and **Rest** are functions.
- **Find** is the predicate that does for lists what ***Member*** does for sets.
- **List?** is a predicate that is true only of lists.
- The empty list is **f1.**
- The term **Cons (x, y),** where y is a nonempty list, is written **[x/y]**.
- The term **Cons (x, Nil)**, (i.e. The list containing the element **x**), is written as **x1.**
- A list of several elements, such as [A,B,C] corresponds to the nested term **Cons(A, Cons(B, Cons(C, Nil)))**

❖ **The Wumpus World:-**

- The first order axioms of wumpus world are more concise, capturing in a natural way what exactly we want to represent the concept.
- Here the more interesting question is **"how an agent should organize what it knows in order to take the right actions".**
- For this purpose we will consider three agent architectures:

  - ✓ Reflex agents - classify their percept and act accordingly
  - ✓ Model based agents - construct an internal representation of the World and use it to act
  - ✓ Goal based agents - form goals and try to achieve them

- The first step of wumpus world agent construction is to define the interface between the environment and the agent.
- The percept sentence must include both the percept and the time at which it occurred, to differentiate between the consequent percepts.
- For Example:-

Percept ([Stench, Breeze, Glitter, None, None], 3)

- In this sentence
  Percept - predicate

Stench, Breeze, Glitter - Constants
3 - Integer to represent to time.

- The agents action are,
  Turn (Right)
  Turn (Left)
  Forward
  Shoot
  Grab
  Release
  Climb
- To determine which is best, the agent program constructs a query such as
  $\exists_a$ **BestActions(a,5)**
- **ASK** solves this query and returns a binding list such as {a/Grab}.
- The agent program then calls **TELL** to record the action which was taken to update the Knowledge base **KB.**

❖ **Types of Sentences:-**

- The percept sentences are classified in to two as,
  - ✓ **Synchronic (Same time)**
  - ✓ **Diachronic (across time)**
- **Synchronic: -** The sentences dealing with time is synchronic if they relate properties of a world state to other properties of the same world state.
- **Diachronic: -** The sentences describing the way in which the world changes (or does not change) are diachronic sentences

❖ **Kinds of Synchronic Rules:-**

- There are two kinds of synchronic rules that could allow to capture the necessary information for deductions are,
  - ✓ Diagnostic rules
  - ✓ Casual rules

  - o **Diagnostic Rules: -** Infer the presence of hidden properties directly from the percept – derived (observed) information.
  - o For Example: - For finding pits, if a square is breezy, some adjacent square must contain a pit.

    $\forall_s$ **Breezy(s)** $\Rightarrow \exists_r$ **Adjacent (r,s)** Λ **Pit(r)**

  - o If a square is not breezy, no adjacent square contains a pit.

    $\forall_s \neg$**Breezy(s)** $\Rightarrow \neg\exists_r$ **Adjacent (r,s)** Λ **Pit(r)**

  - o Combining these two, the derived biconditional sentence is :

    $\forall_s$ **Breezy(s)** $\Leftrightarrow \exists_r$ **Adjacent (r,s)** Λ **Pit(r)**

- **Casual Rules: -** Reflect the assumed direction of casuality in the world. Some hidden property of the world causes certain percepts to be generated.
- For Example:- A Pit causes all adjacent squares to be breezy.

$$\forall_r \text{Pit}(r) \Rightarrow [\forall_s \text{Adjacent}(r, s) \Rightarrow \text{Breezy}(s)]$$

- If all squares adjacent to a given square are pitless, the square will not be breezy

$$\forall_s [\forall_r \text{Adjacent}(r,s) \Rightarrow \neg \text{Pit}(r)] \Rightarrow \neg \text{Breezy}(s)$$

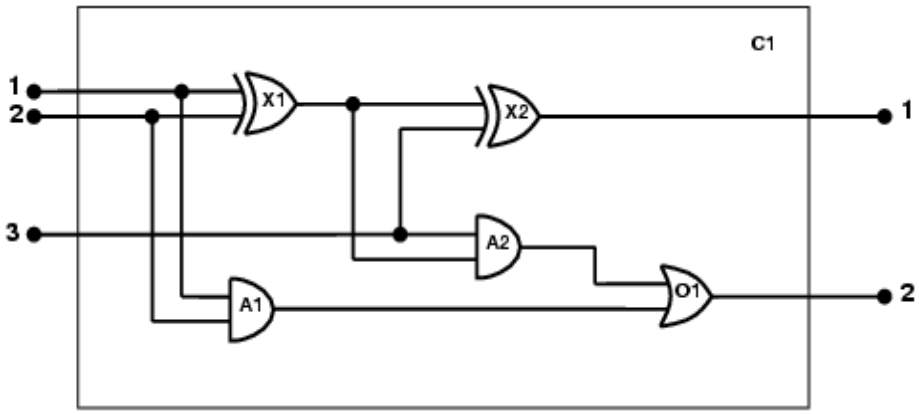
- System that reason with casual rules are called model-based reasoning systems, because the casual rules from a model of how the environment operates.

2.5.5 KNOWLEDGE ENGINEERING IN FIRST ORDER LOGIC:-

- **Knowledge Engineering: -** The general process of Knowledge Base **KB** Construction.
- **Knowledge Engineer:** - Who investigates a particular domain, learns what concepts are important in that domain and creates a formal representation of the objects and relations in the domain.
- The knowledge engineering projects vary widely in content, scope and difficulty, but all projects include the following steps,

  - ✓ **Identify the Task: -** The Knowledge engineer should identify the **PEAS** description of the domain.
  - ✓ **Assemble the relevant knowledge: -** The idea of combining expert's knowledge of that domain (i.e.) a process called **knowledge acquisition.**
  - ✓ **Decide on a vocabulary of predicates, functions and constants: -** Translate the important domain-level concepts into logical level name. The resulting vocabulary is known as **ontology** of the domain, which determines what kinds of things exist, but does not determine their specific properties and interrelationships.
  - ✓ **Encode general knowledge about the domain: -** The knowledge engineer writes the axioms (rules) for all the vocabulary terms. The misconceptions are clarified from step 3 and the process is iterated.
  - ✓ **Encode a description of the specific problem instance: -** To write simple atomic sentences about instances of concepts that are already part of the ontology.
  - ✓ **Pose queries to the inference procedure and get answers: -** For the given query the inference procedure operate on the axioms and problem specific facts to derive the answers.
  - ✓ **Debug the knowledge base: -** For the given query , if the result is not a user expected on then **KB is** updated with relevant or missing axioms.

- The seven step process is explained with the domain of **ELECTRONIC CIRCUITS DOMAIN.**



- ✓ **Identify the task: -**
  - Analyse the circuit functionality, does the circuit actually add properly? (Circuit Verification).
- ✓ **Assemble the relevant knowledge: -**
  - The circuit is composed of wire and gates.
  - The four types of gates (AND, OR, NOT, XOR) with two input terminals and one output terminal knowledge is collected.
- ✓ **Decide on a vocabulary: -**
  - The functions, predicates and constants of the domain are identified.
  - Functions are used to refer the type of gate.
    Type (x1) = XOR,
    where x1 ---- Name of the gate and Type ---- function
  - The same can be represented by either binary predicate (or) individual type predicate.
    Type (x1, XOR) – binary predicate
    XOR (x1) – Individual type
  - A gate or circuit can have one or more terminals. For x1, the terminals are x1In1, x1In2 and x1 out1
    Where x1 In1 ----- 1st input of gate x1
    x1 In2 ----- 2nd input of gate x1
    x1 out1 ---- output of gate x1
  - Then the connectivity between the gates represented by the predicate connected. (i.e.) connected (out(1, x1), In(1,x2)).
  - Finally the possible values of the output terminal C1, as true or false, represented as a signal with 1 or 0.
- ✓ **Encode general knowledge of the domain:-**
  - This example needs only seven simple rules to describe everything need to know about circuits
  - If two terminals are connected, then they have the same signal:
    - $\forall t_1,t_2$ Connected(t_1, t_2) $\Rightarrow$ Signal(t_1) = Signal(t_2)
  - The signal at every terminal is either 1 or 0 (but not both):
    - $\forall t$ Signal(t) = 1 $\vee$ Signal(t) = 0
    - $1 \neq 0$

- Connected is a commutative predicate
  - $\forall t_1,t_2$ Connected(t_1, t_2) $\Rightarrow$ Connected(t_2, t_1)
- An OR gate's output is 1 if and only if any of its input is 1:
  - $\forall g$ Type(g) = OR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 1
- An AND gate's output is 0 if and only if any of its inputs is 0
  - $\forall g$ Type(g) = AND $\Rightarrow$ Signal(Out(1,g)) = 0 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 0
- An XOR gate's output is 1 if and only if inputs are different:
  - $\forall g$ Type(g) = XOR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow$ Signal(In(1,g)) $\neq$ Signal(In(2,g))
- A NOT gate's output is different from its input:
  - $\forall g$ Type(g) = NOT $\Rightarrow$ Signal(Out(1,g)) $\neq$ Signal(In(1,g))

✓ **Encode the specific problem instance:**

- First, we categorize the gates:

  Type(X_1) = XOR Type(X_2) = XOR

  Type(A_1) = AND Type(A_2) = AND

  Type(O_1) = OR

- Then, we show the connections between them

Connected(Out(1,X_1),In(1,X_2)) Connected(In(1,C_1),In(1,X_1))

Connected(Out(1,X_1),In(2,A_2)) Connected(In(1,C_1),In(1,A_1))

Connected(Out(1,A_2),In(1,O_1)) Connected(In(2,C_1),In(2,X_1))

Connected(Out(1,A_1),In(2,O_1)) Connected(In(2,C_1),In(2,A_1))

Connected(Out(1,X_2),Out(1,C_1)) Connected(In(3,C_1),In(2,X_2))

Connected(Out(1,O_1),Out(2,C_1)) Connected(In(3,C_1),In(1,A_2))

✓ **Pose Queries to the inference procedure:**

- What combinations of inputs would cause the first output of C1(the sum bit) to be 0 and the second output of C1 (the carry bit) to be 1?

  $\exists i_1,i_2,i_3$ Signal(In(1,C1)) = $i_1 \wedge$ Signal(In(2,C_1)) = $i_2 \wedge$ Signal(In(3,C_1)) = $i_3 \wedge$ Signal(Out(1,C_1)) = o $\wedge$ Signal(Out(2,C_1)) = 1

- The answers are substitutions for the variables i_1,i_2 and i_3 Such that the resulting sentence is entailed by the knowledge base.

- There are three such substitutions as:

  { i_1/1,i_2/1 , i_3/0} { i_1/1,i_2/0 , i_3/1} { i_1/0,i_2/1 , i_3/1}

- What are the possible sets of values of all the terminals for the adder circuit?

  $\exists i_1,i_2,i_3,o_1,o_2$ Signal(In(1,C1)) = $i_1 \wedge$ Signal(In(2,C_1)) = $i_2 \wedge$ Signal(In(3,C_1)) = $i_3 \wedge$ Signal(Out(1,C_1)) = $o_1 \wedge$ Signal(Out(2,C_1)) = o_2

✓ **Debug the knowledge base:**

- The knowledge base is checked with different constraints.
- For Example:- if the assertion $1 \neq 0$ is not included in the knowledge base then it is variable to prove any output for the circuit, except for the input cases 000 and 110.

2.6 Inference in First-order Logic:-

- We have learned seven inference rules of propositional logic.
- These rules are applicable for First-order logic also
- With those rules First-order logic holds some additional rules "with quantifiers" in which substituting particular individual for the variable is done (i.e.) SUBST(Θ, α) to denote the result of applying the substitution (or) binding list Θ to the sentence α.
- For Example:-
  SUBST ({x/Ram, y/John} Likes(x, y)) = Likes(Ram, John)
- The following are the new three inference rules for First-order Logic.
  - ✓ Universal Elimination
  - ✓ Existential Elimination
  - ✓ Existential Introduction
- **Universal Elimination:-** For any sentence α, variable v and ground term g;

  ∀v α / SUBST({v/g}, α)
  Example:-
  ∀x likes (x, Icecream) is a sentence **α**.

SUBST ({x/John}, α) is a substitution Θ = John
Likes (John, Icecream) – Inferred sentence

- **Existential Elimination :-** For any sentence α variable v, and constant symbol k that does not appear elsewhere in the Knowledge base:

∃v α / SUBST({v/k}, α)
Example:-
∃x Kill (x, Victim) – **α**
SUBST({x/Murderer, Victim}, **α**) where **Θ** = Murderer
Kill (Murderer, Victim) – Inferred Sentence

- **Existential Introduction :-** For any sentence α, variable v that does not occur in **α**, and ground term **g** that does occur in **α**

α / ∃v SUBST({g/v}, α)
Example:-
Likes (John, Icecream) – **α**
∃x Likes (x, Icecream) – Inferred Sentence

2.6.1 AN EXAMPLE PROOF USING FIRST- ORDER LOGIC:-

- An application of inference rule is matching their premise patterns to the sentences in the KB and then adding their conclusion patterns to the KB.
- **Task: -** For the given situation described in English, Convert it into its equivalent FOL representation and prove that "West is a Criminal".
- **Situation: -** The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.
- **Solution: -** The given description is splitted into sentences and is converted into its corresponding FOL representation.

  - ✓ It is a crime for an American to sell weapons to hostile nations:
    ∀x y z American(x) ∧ Weapon(y) ∧ Nation(z) ∧ Hostile(z) ∧ Sells(x, y, z ⇒ Criminal(x)
  - ✓ Nono … has some missiles,
    ∃x Owns(Nono,x) ∧ Missile(x)
  - ✓ all of its missiles were sold to it by Colonel West
    ∀x Missiles(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
  - ✓ We will also need to know that missiles are weapons
    ∀x Missile(x) ⇒ Weapon(x)
  - ✓ An enemy of America counts as "hostile"
    ∀x Enemy(x,America) ⇒ Hostile(x)
  - ✓ West, who is American …
    American(West)

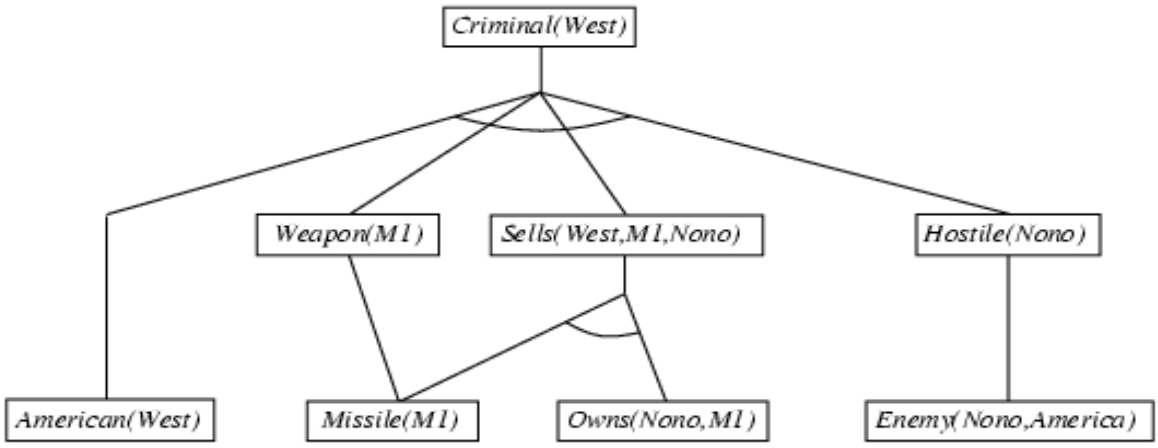
- ✓ Nono, is a nation
  Nation (Nono)
- ✓ Nono, an enemy of America
  Enemy (Nono, America)
- ✓ America is nation
  Nation (America)

2.7 Forward Chaining:-

- The Generalized Modus Ponens rule can be used by Forward Chaining.
- From the sentences in the **KB** which in turn derive new conclusions.
- Forward chaining is preferred when new fact is added to the database and we want to generate its consequences.
- **Forward Chaining Algorithm:-**
  - ✓ Forward chaining is triggered by the addition of new fact "p" into the knowledge base (i.e.) the action **TELL** is performed.
  - ✓ If the new fact is a rename of any other existing sentence in the **KB** then it is not included in **KB.**
  - ✓ With the new fact "p" find all premises that had "p" as premise and if any other premise is already known to hold then its consequence is included in **KB.**
  - ✓ The important operation of forward chaining is renaming : One sentence is a renaming of another if, they are identical except for the names of the variables.
  - ✓ **For Examples:-**
    - o Likes(x, Icecream) and Likes(y, Icecream) are renaming of each other.
    - o Likes(x,x) and Likes(x,y) are not renaming of each other (i.e.) its variable differs, the meaning is logically different.
  - ✓ Consider the **KB** of crime problem represented in Horn form to explain the concept of forward chaining.
  - ✓ The implication sentences are (i), (iv), (v), (vi)
  - ✓ Two iterations are required:
  - ✓ On the first iteration,
    - o Step (i) has unsatisfied premises
    - o Step (iv) is satisfied with {x/M1} and sells (west, M1,Nono) is added
    - o Step (v) is satisfied with {x/m1} and weapon (M1) is added
    - o Step (vi) is satisfied with {x/Nono}, and Hostile (Nono) is added
  - ✓ On the second iteration,
    - o Step (i) is satisfied with {x/West, y/M1, z/Nono} and Criminal(west) is added.
  - ✓ The following table shows the forward chaining algorithm,
  - ✓ **Inputs:- KB**, the Knowledge base, a set of first-order definite clauses ***α,*** the query, an atomic sentence
  - ✓ **Local variable:- new,** the new sentences inferred on each iteration

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

- ✓ The following figure shows the proof tree generated by forward chaining algorithm on the Crime Example,



- ✓ The above discussed inference processes are not directed towards solving any particular problem: for this reason it is called a **data-driven or data-directed procedure.**
- ✓ In this problem, no new inferences are possible because every sentence concluded by forward chaining is already exist in the **KB.** Such a **KB** is called a **fixed point** of the inference process.
- ✓ FOL-FC-ASK function is sound and complete.
- ✓ Every inference is an application of generalized modus ponens, which is sound.
- ✓ Then it is complete for definite clauses **KB** (i.e.) it answers every query whose answers are entailed by any **KB** of definite clauses.

2.7.1 Efficient Forward chaining:-

- The above discussed FC Algorithm has three possible types of complexity.

- ✓ **Pattern Matching: -** "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the **KB.**
- ✓ **Matching rules against known facts:-** The algorithm re-checks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the **KB** on each iteration
- ✓ Irrelevant facts to the goal are generated

- In forward chaining approach, inference rules are applied to the knowledge base, yielding new assertions.
- This process repeats forever or until some stopping criterian is met.
- This method is appropriate for the design of an agent, that is on each cycle, we add the percepts to the knowledge base and run the forward chainer, which chooses an action to perform according to a set of condition action rules.
- Theoretically a **production system** can be implemented with a theorem prover, using resolution to do forward chaining over a full first order knowledge base.
- An efficient language can be used to perform this task, because it reduces the branching factor.
- The typical production system has three features:
  - ✓ The system maintained a **KB** called the working memory which has a set of positive literals with no variable
  - ✓ The system maintains a **rule memory**. This contains a set of inference rules $P_1 \wedge P_2 \Rightarrow act_1 \wedge act_2 \ldots$. That act_i is executed when p_i is satisfied, which performs adding or deleting an element from the **working memory – match phase.**
  - ✓ In each cycle, the system computes the subset of rules whose left-hand side is satisfied by the current contents of the **working memory - match phase.**

2.8 Backward Chaining:-

- Backward chaining is designed to find all answers to a question asked to the knowledge base. Therefore it requires a **ASK** procedure to derive the answer.
- The procedure **BACK WARD-CHAIN** will check two constraints.
  - ✓ If the given question can derive a answer directly from the sentences of the knowledge base then it returns with answers.
  - ✓ If the first constraint is not satisfied then it finds all implications whose conclusion unifies with the query and tries to establish the premises of those implications. If the premise is a conjunction then BACK-CHAIN processes the conjunction conjunct by conjunct, building up the unifiers for the whole premises as it goes.
- **Composition of Substitutions:-** COMPOSE(Θ_1, Θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn (i.e.),

  SUBST (COMPOSE (Θ_1, Θ_2), p) = SUBST (Θ_2, SUBST (Θ_1, p))
- For Example:-

  P – Sells (x,M1, y)

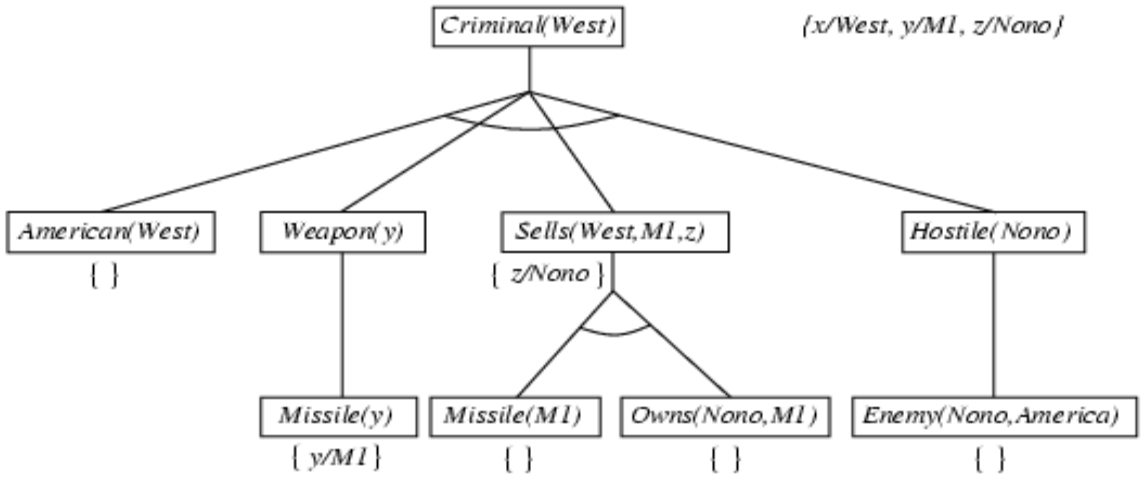
SUBST (Θ_2, SUBST (Θ_1, p))
SUBST (Θ_2, SUBST (x/West, p)) i.e. (Θ_1 = x/west)
SUBST ((y/Nono), (x/West, p)) i.e. (Θ_2 = y/Nono)
Therefore p – Sells (West, M1, Nono)

- The following table shows the backward chaining algorithm,

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
inputs: KB, a knowledge base
$goals$, a list of conjuncts forming a query
θ, the current substitution, initially the empty substitution { }
local variables: ans, a set of substitutions, initially empty

if $goals$ is empty **then return** $\{\theta\}$
$q' \leftarrow$ SUBST(θ, FIRST($goals$))
for each r **in** KB where STANDARDIZE-APART(r) = ($p_1 \wedge \ldots \wedge p_n \Rightarrow q$)
and $\theta' \leftarrow$ UNIFY(q, q') succeeds
$ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta, \theta'$)) $\cup$ ans
return ans

- The following graph shows the proof tree to infer that west is a criminal,



- To prove Criminal(x), we have to prove the five conjuncts below it
- Some of which are directly exist in the knowledge base, others require one more iteration of backward chaining.
- In the search process the substitution of values for the variables has to be done in a correct way, otherwise it may lead to failure solution.
- The following are the some properties of Backward Chaining,
  - Depth-first recursive proof search: space is linear in size of proof
  - Incomplete due to infinite loops
    - $\Rightarrow$ fix by checking current goal against every goal on stack

- Inefficient due to repeated subgoals (both success and failure)
  - ⇒ fix using caching of previous results (extra space)
- Widely used for logic programming

2.8.1 Logic Programming:-

- A system in which KB can be constructed and used.
- A relation between logic and algorithm is summed up in Robert Kowalsh equation

  Algorithm = Logic + Control
- Logic programming languages, usually use backward chaining and input/output of programming languages.
- A logic programming language makes it possible to write algorithms by augmenting logic sentences with information to control the inference process.
- For Example:- PROLOG
  - ✓ A prolog program is described as a series of logical assertions each of which is a Horn Clause.
  - ✓ A Horn Clause is a Clause that has atmost one positive literal,

    Example: - P, ¬P∧Q
  - ✓ Implementation: - All inferences are done by backward chaining, with depth first search. The order of search through the conjuncts of an antecedent is left to right and the clauses in the KB are applied first-to-last order.
- **Example for FOL to PROLOG conversion:-**
  - **FOL**
    - ✓ **∀x Pet(x) ∧ Small(x) ⇒ Apartment(x)**
    - ✓ **∀x Cat(x) v Dog(x) ⇒ Pet(x)**
    - ✓ **∀x Product(x) ⇒ Dog(x) ∧ Small(x)**
    - ✓ **Poodle(fluffy)**
  - **Equivalent PROLOG representation**
    - ✓ **Apartment(x) :- Pet(x), Small(x)**
    - ✓ **Pet(x) :- Cat(x)**
      Pet(x) :- Dog(x)
    - ✓ **Dog(x) :- Poodle(x)**
      Small(x) :- Poodle(x)
    - ✓ **Poodle(fluffy)**
  - In the PROLOG representation the consequent or the left hand side is called as head and the antecedent or the right hand side is called as **body.**
- **Execution of a PROLOG program:-**
  - The execution of a prolog program can happen in two modes,
    1. Interpreters
    2. Compilers
  - **Interpretation:**
    - ✓ A method which uses **BACK-CHAIN** algorithm with the program as the **KB.**

- ✓ To maximize the speed of execution, we will consider two different types of constraints executed in sequence, They are
  1. **Choice Point: -** Generating sub goals one by one to perform interpretation.
  2. **Trail: -** Keeping track of all the variables that have been bound in a stack is called as trail.

- **Compilation:-**
  - ✓ Procedure implementation is done to run the program (i.e.) calling the inference rules whenever it is required for execution.

2.9 Unification:-

- The process of finding all legal substitutions that make logical expressions look identical and
- Unification is a "pattern matching" procedure that takes two atomic sentences, called literals, as input, and returns "failure" if they do not match and a substitution list, Theta, if they do match.
- Theta is called the most general unifier (mgu)
- All variables in the given two literals are implicitly universally quantified
- To make literals match, replace (universally-quantified) variables by terms
- The unification routine, UNIFY is to take two atomic sentences p and q and returns α substitution that would make p and q look the same

  UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q)

  θ = Unifier of two sentences
- For example:-

  p – S1(x, x) q

  – S1(y, z)

  Assume θ = y

  p – S1(y, y) – x/y (Substituting y for x)

  q – S1(y, y) – z/y (Substituting y for z)
- In the above two sentences (p, q), the unifier of two sentences (i.e.) θ = y is substituted in both the sentences, which derives a same predicate name, same number of arguments and same argument names.
- Therefore the given two sentences are unified sentences.
- The function UNIFY will return its result as fail, for two different types of criteria's as follows,
  - ✓ If the given two atomic sentences (p, q) are differs in its predicate name then the UNIFY will return failure as a result

    For Example: - p – hate (M, C), q – try (M, C)
  - ✓ If the given two atomic sentences (p, q) are differs in its number of arguments then the UNIFY will return failure as a result

    For Example: - p – try (M, C), q – try (M, C, R)
- For Example: - The Query is **Knows (John, x) whom does John Know?**

- Some answers to the above query can be found by finding all sentences in the KB that unify with knows (John, x)
- Assume the KB has as follows,
  Knows (John, John)
  Knows (y, Leo)
  Knows (y, Mother(y))
  Knows (x, Elizabeth)
- The results of unification are as follows,
  UNIFY (knows (john, x), knows (John, Jane)) = {x/Jane}
  UNIFY (knows (john, x), knows (y, Leo)) = {x/Leo, y/John}
  UNIFY (knows (john, x), knows (y, Mother(y)) = {y/John, x/Mother (John)}
  UNIFY (knows (john, x), knows (x, Elizabeth)) = fail
- x cannot take both the values John and Elizabeth at the same time.
- Knows (x, Elizabeth) means "Everyone knows Elizabeth" from this we able to infer that John knows Elizabeth.
- This can be avoided by using standardizing apart one of the two sentences being unified (i.e.) renaming is done to avoid name clashes.
- For Example:-
  UNIFY (Knows (john, x), knows (x_1, Elizabeth)) = {x/Elizabeth, x_1/John}

2.9.1 Most general Unifier (MGU):-

- UNIFY should return a substitution that makes the two arguments look the same, but there may be a chance of more than one unifier.
- For Example:-
  UNIFY (knows (john, x), knows (y, z)) = {y/John, x/z} or {y/John, x/John, z/John}
- The result of applying 1st unifier is knows (John, z) and the 2nd unifier is knows (John, John).
- Here the first unifier result is more general than the second one, because it places less restriction on the values of the variables.
- This indicates that every unifiable pair of expressions, a single MGU is exist until renaming of variables.
- The following table shows the unification algorithm,
- The following are the steps to be done for unification of two sentences p and q is given below,
  - ✓ Recursively explore the two expressions simultaneously along with unifier returns failure if two corresponding points in the structure do not match. Therefore the time complexit y is $O(n^2)$ in the size of the expressions being unified.
  - ✓ When the variable is matched against a complex term, one must check, whether the variable itself occurs, if it is true then returns failure (consistent unifier is not allowed) – occur check.

```
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
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

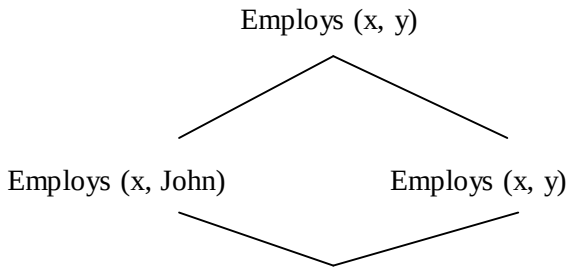
- The above table shows the unification algorithm

2.9.2 Storage and retrieval:-

- Once the data type for sentences and terms are defined, we need to maintain asset of sentences in a KB (i.e.) to store and fetch a sentence or term,
  - ✓ Store (s) – stores a sentence s.
  - ✓ Fetch (q) – returns all unifiers
- Such that the query q unifies with some sentences in the KB.
- For Example: - The unification for Knows (John, x) is an Instance of fetching.
- The simplest way to store and fetch is maintain a long list in sequential order.
- For a Query q, call UNIFY (q, s) for every sentences s in the list, requires O(n) time on an n-element KB.
- To make the fetch more efficient, indexing the facts in KB is done.
- The different types of indexing are as follows,
  - ✓ Table based Indexing
  - ✓ Tree based Indexing

- ✓ Predicate based Indexing

- A simple form of indexing is predicate indexing puts all the knows facts in one bucket and all the brother facts in another.
- The buckets are stored in hash table for efficient access,
- Where hash table means "a data structure for storing and retrieving information indexed by fixed keys"
- Given sentence to be stored, it is possible to construct indices for all possible queries that unify with it,
- For Example:- For the fact Employs (SSN, John) the queries are as follows,
  - ✓ Employs (SSN, John) Does SSN employ John?
  - ✓ Employs (x, John) who employs John?
  - ✓ Employs (SSN, y) whom does SSN employ?
  - ✓ Employs (x, y) who employs whom?
- These queries form a substitution lattice (i.e.) Properties of Lattice:- child of any node in the lattice is obtained from its parent by a single substitution and the highest common descendent of any two nodes is the result of applying their most general unifier.
- For a predicate with n arguments, the lattice contains $O(2^n)$ nodes
- The following diagram shows the subsumption lattice,



2.9.3 Advantages and Disadvantages:-

Advantages:-

- The scheme works very well whenever the lattice contains a small number of nodes.
- For a predicate with n arguments, the lattice contains O(2n) nodes.

Disadvantages:-

- If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices.
- At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices.

2.10 Resolution:-

- Resolution is a complete inference procedure for first order logic
- Any sentence a entailed by KB can be derived with resolution
- Catch: proof procedure can run for an unspecified amount of time

- At any given moment, if proof is not done, don't know if infinitely looping or about to give an answer
- Cannot always prove that a sentence a is *not* entailed by KB
- First-order logic is semidecidable

- Rules used in the resolution procedure are :
  - Simple resolution inference rule – premises have exactly two disjuncts

$$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$$

  - Resolution inference rule – two disjunctions of any length, if one of disjunct in the class (p_j) unifies with the negation of the disjunct in the other class, then infer the disjunction of all the disjuncts except for those two,
  - **Using disjunctions:-** For literals p_i and q_i, where UNIFY (p_j, $\neg q_k$) = θ

$$\frac{p_1 \vee \ldots\ldots\ldots p_j \ldots\ldots\ldots \vee\ p_m,\ q_1 \vee \ldots\ldots\ldots q_k \ldots\ldots\ldots \vee q_n}{SUBST(\theta, (p_1 \vee \ldots\ldots\ldots p_{j-1} \vee\ p_{j+1} \ldots\ldots\ldots p_m \vee q_1 \ldots\ldots\ldots q_{k-1} \vee q_{k+1} \ldots\ldots\ldots \vee q_n}$$

  - **Using implications:-** For atoms p_i, q_i, r_i, s_i where UNIFY (p_j, q_k) = θ

$$p_1 \wedge \ldots\ldots\ldots p_j \wedge \ldots\ldots\ldots p_{n1} \Rightarrow r_1 \vee \ldots\ldots r_{n2}$$

$$\frac{s_1 \wedge \ldots\ldots\ldots s_{n3} \Rightarrow,\ q_1 \vee \ldots\ldots\ldots q_k \ldots\ldots\ldots q_{n4}}{SUBST(\theta, (p_1 \wedge \ldots p_{j-1} \wedge\ p_{j+1} \ldots\ldots p_{n1} \wedge s_1 \wedge \ldots\ldots s_{n3} \Rightarrow r_1 \vee \ldots\ldots r_{n2} \vee q_1 \vee \ldots\ldots\ldots\ldots q_{k-1} \vee q_{k+1} \ldots\ldots\ldots \vee q_{n4}}$$

2.10.1 Canonical Form (or) Normal form for Resolution:-

- The canonical form representation of sentences for resolution procedure (to derive pa proof) is done in two ways, they are as follows,
  - **Conjunctive Normal form (CNF):-** All the disjunctions are joined aqs one big sentences.
  - **Implicative Normal Form (INF):-** All the conjunctions of atoms on the left and a disjunction of atoms on the right.
- The following table shows the Knowledge base for CNF and INF,

Conjuctive Normal Form	Implicative Normal Form
$\neg$ P(w) $\vee$ Q(w)	P(w) $\Rightarrow$ Q(w)
P(x) $\vee$ R(x)	True $\Rightarrow$ P(x) $\vee$ R(x)
$\neg$ Q(y) $\vee$ S(y)	Q(y) $\Rightarrow$ S(y)
$\neg$ R(z) $\vee$ S(z)	R(z) $\Rightarrow$ S(z)

- Resolution is a generalization of Modus Ponen.
- The following is the representation of Modus Ponen rule in resolution as a special case (i.e.),

$\frac{\alpha, \alpha \Rightarrow \beta}{\beta}$ is equivalent to $\frac{True \Rightarrow \alpha, \alpha \Rightarrow \beta}{True \Rightarrow \beta}$

2.10.2 Methods used for resolution technique

- **Resolution Proof:-** A set of inference rules and resolution rules can be used to derive a conclusion from the KB.
- **Resolution with refutation (or) proof by contradiction (or) reduction and absurdum:-** One complete inference procedure using resolution is refutation. The idea is to prove P, we assume P is false (i.e. add $\neg P$ to KB) and prove a contradiction, that is the KB implies P.

$$\mathbf{(KB \wedge \neg P \Rightarrow False) \Leftrightarrow (KB \Rightarrow P)}$$

- **Example:**

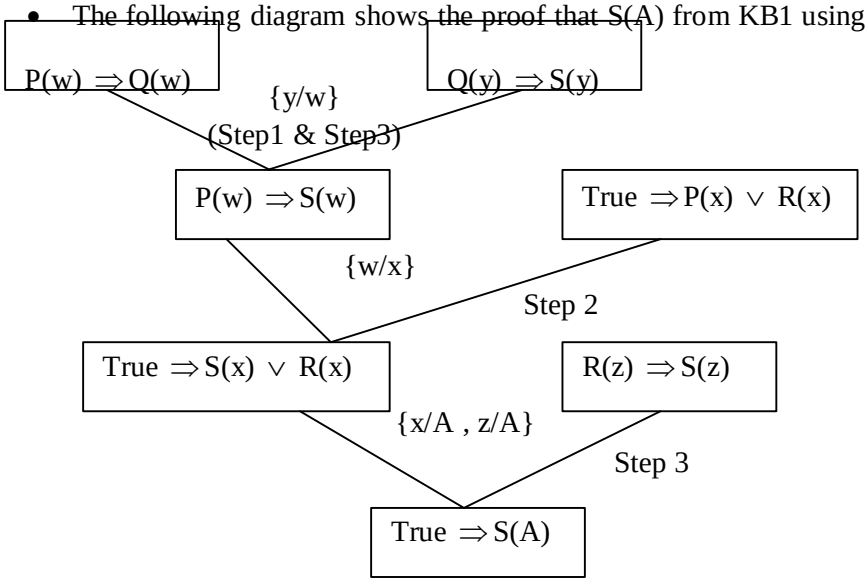
1. Prove that S(A) ts true from KB1 of CNF and INF representation using
   - ✓ Resolution Proof
   - ✓ Resolution with refutation

(a) Resolution Using INF representation:-

Given (KB1):-

1. $P(w) \Rightarrow Q(w)$
2. $True \Rightarrow P(x) \vee R(x)$
3. $Q(y) \Rightarrow S(y)$
4. $R(z) \Rightarrow S(z)$

- The following diagram shows the proof that S(A) from KB1 using resolution



- Resolution rule:- In the first step transitive implication rule is used
- Substitution of one predicate in the other. (i.e.) $P(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow P(x) \vee S(x)$ that is instead of P(x), S(x) is substituted.

- Substitution of one predicate in the other. (i.e.) R(A) $\Rightarrow$ S(A) is substituted in True $\Rightarrow$ S(A) $\vee$ R(A) that is instead of R(A), S(A) is substituted, which derives True $\Rightarrow$ S(A) $\vee$ S(A) is equivalent to True $\Rightarrow$ S(A)
- Therefore S(A) is true is proved using resolution technique for INF representation.
- Where each "Vee" Shape in the proof tree represents a resolution step,
- The two sentences at the top are the premises from the KB, and the one at the bottom is the conclusion (or) resolvent.

b. Resolution Using CNF representation:-
Given (KB1):-

1. ~P(w) $\vee$ Q(w)
2. P(x) $\vee$ R(x)
3. ~Q(y) $\vee$ S(y)
4. ~R(z) $\vee$ S(z)

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{\sim P(w) \vee Q(w), \sim Q(w) \vee S(w)}{\sim P(w) \vee S(w)}$$

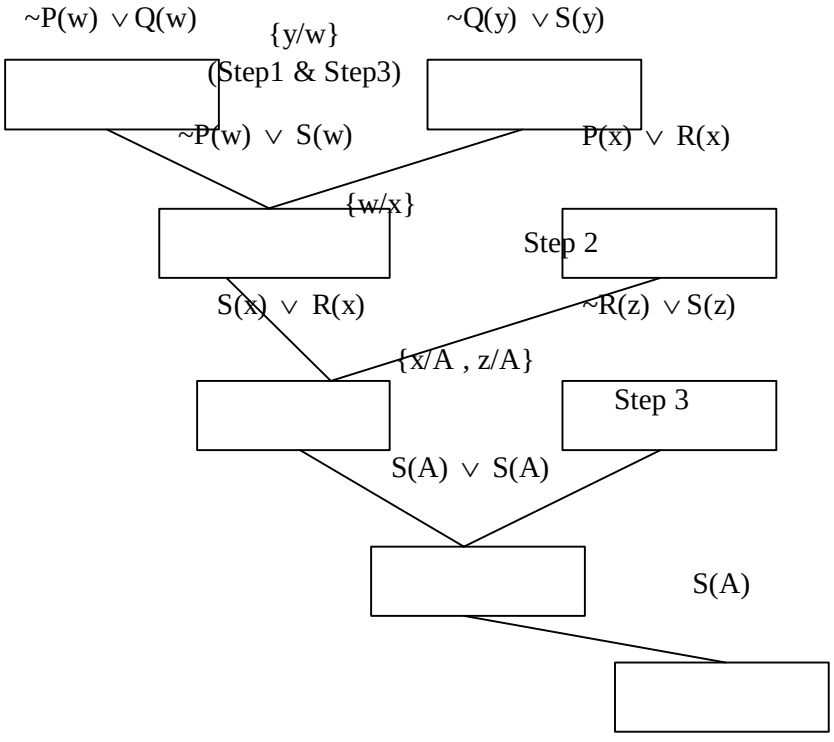
- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{R(x) \vee P(x), \sim P(x) \vee S(x)}{R(x) \vee S(x)}$$

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{S(A) \vee R(A), \sim R(A) \vee S(A)}{S(A) \vee S(A)}$$

- Therefore S(A) is true is proved using resolution technique for CNF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution.



C. Resolution with refutation Proof using INF representation:-

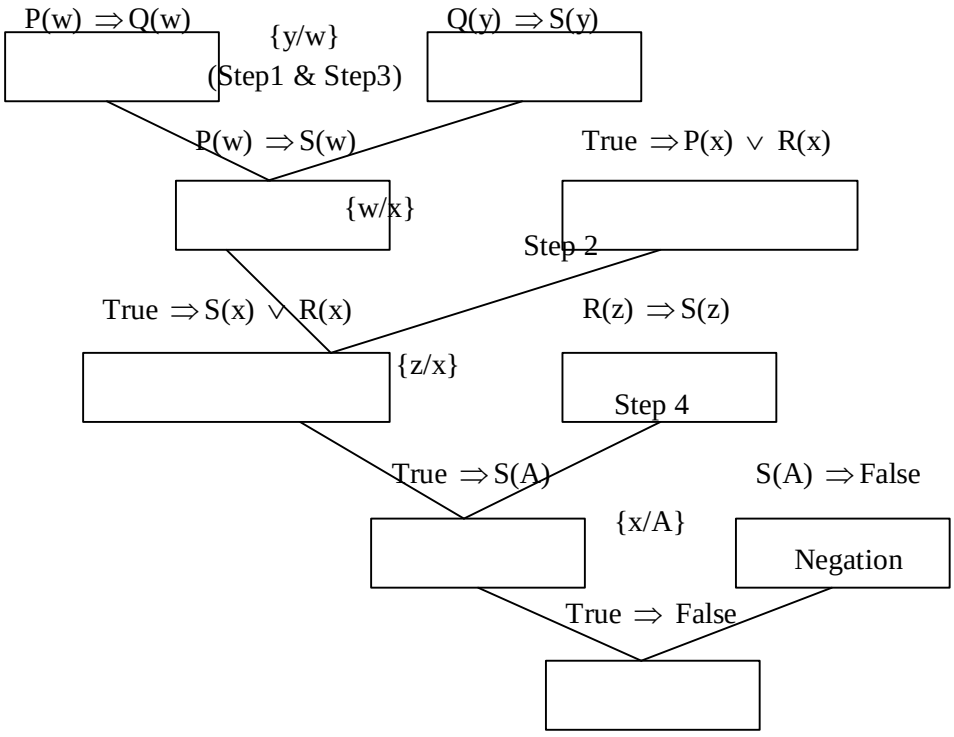
Given (KB1):-

1. $P(w) \Rightarrow Q(w)$
2. $True \Rightarrow P(x) \vee R(x)$
3. $Q(y) \Rightarrow S(y)$
4. $R(z) \Rightarrow S(z)$

- **Resolution rule:-** In this step transitive implication is applied,

$$\frac{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\alpha \Rightarrow \gamma}$$

- Substitution of one predicate in the other. (i.e.) $P(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow P(x) \vee S(x)$ that is instead of P(x), S(x) is substituted.
- Substitution of one predicate in the other. (i.e.) $R(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow S(x) \vee R(x)$ that is instead of R(x), S(x) is substituted, which derives $True \Rightarrow S(x) \vee S(x)$ is equivalent to $True \Rightarrow S(x)$
- To prove using refutation, negation of given proof is added to the KB and derives a contradiction, then the given statement is true otherwise it is false.
- Therefore in step 4, we assume $S(A) \Rightarrow False$, derives a contradiction $True \Rightarrow False$.
- Therefore S(A) is true is proved using refutation technique in INF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution with refutation in INF representation.



D. Resolution with refutation using CNF representation:-

Given KB1:-

1. ~P(w) ∨ Q(w)
2. P(x) ∨ R(x)
3. ~Q(y) ∨ S(y)
4. ~R(z) ∨ S(z)

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{\sim P(w) \vee Q(w), \sim Q(w) \vee S(w)}{\sim P(w) \vee S(w)}$$

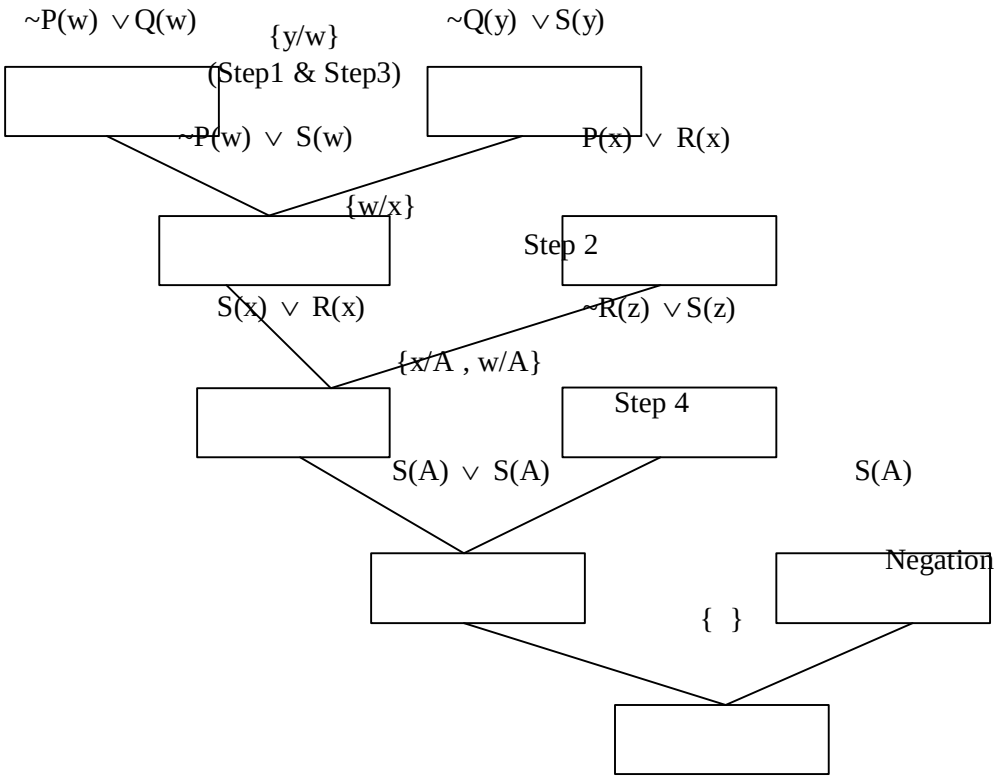
- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{R(x) \vee P(x), \sim P(x) \vee S(x)}{R(x) \vee S(x)}$$

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \text{ (i.e.) } \frac{S(A) \vee R(A), \sim R(A) \vee S(A)}{S(A) \vee S(A)}$$

- To prove using refutation, the negation of the given statement to be proved is added to the KB and it derives a empty set, represents that the statement is True in the KB.
- Therefore S(A) is True is proved using resolution with refutation in CNF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution with refutation in CNF representation.



2.10. 3 Resolution technique (From FOL Representation):-

- In the previous section we learned how to prove the given fact using two different types of resolution techniques (Resolution proof, Resolution with refutation proof) from the given KB representation (CNF, INF).
- Suppose if the KB is given as a English description, to prove a fact , then how to derive the conclusion? What are the sequence of steps to be done? Are discussed one by one,
  - The given KB description is converted into FOL Sentences
  - FOL sentences are converted to INF (or) CNF representation without changing the meaning of it (using conversion to normal form procedure)
  - Apply one of the resolution technique (Resolution proof (or) Resolution with refutation proof), to resolve the conflict.
  - Derive the fact to be proved or declare it as a incomplete solution.

2.10.3.1 Conversion to Normal Form or Clause Form (Procedure) :-

1. **Eliminate Implications:-**
   Eliminate Implication by the corresponding disjunctions,
   (i.e.) $p \Rightarrow q$ is the same as $\neg p \vee q$
2. **Move $\neg$ inwards :-**
   Negations are allowed only on atoms in CNF and not at all in INF. Therefore eliminate negations with, Demorgan's laws, the quantifier equivalences and double negation.
   Demorgans Law
   $\neg(p \vee q)$ becomes $\neg p \wedge \neg q$
   $\neg(p \wedge q)$ becomes $\neg p \vee \neg q$
   Quantifier equivalences
   $\neg \forall x p$ becomes $\exists x \neg p$
   $\neg \exists x p$ becomes $\forall x \neg p$
   Double Negation
   $\neg\neg p$ becomes p Double negation
3. **Standardize variables:-**
   If the sentence consists of same variable name twice, change the name of one of the variable. This avoids confusion later when we drop the quantifiers,
   (i.e.) $(\forall x p(x)) \vee (\exists x Q(x))$ is changed into $(\forall x p(x)) \vee (\exists y Q(y))$
4. **Move quantifiers left:-**
   The quantifiers in the middle of the sentence to be moved to the left, without changing the meaning of the sentence
   (i.e.) $p \vee \forall$ xq becomes $\forall x \;\; p \vee q$, which is true because p here is guaranteed not to contain x.
5. **Skolimize:-**
   Skolimization is the process of removing existential quantifiers by elimination, it is very similar to the existential elimination rule.
   (i.e.) $\exists x p(x)$ into p(A), where A is a constant that does not appear elsewhere in the KB.

For Example:- "Everyone has a heart"

- FOL : $\forall x$ person(x) $\Rightarrow$ $\exists y$ Heart (y) $\wedge$ Has (x, y)
- Replace $\exists y.....y$ with a constant H

$\forall x$ person (x) $\Rightarrow$ Heart (H) $\wedge$ Has (x, y)

- The above representation says that everyone has the same heart H, not necessarily shared between each other. It can be rewritten by applying a function to each person that maps from person to heart.

  $\forall x$ person (x) $\Rightarrow$ Heart (F(x) $\wedge$ Has (x, F(x)) where F is a function that does not appear elsewhere in the KB and it is called as Skolem function.

6. **Distribute $\wedge$ over $\vee$:-**

   $(a \wedge b) \vee c$ becomes $(a \vee c) \wedge (b \vee c)$

7. **Flatten nested conjunctions and disjunctions :-**

   $(a \vee b) \vee c$ becomes $(a \vee b \vee c)$

   $(a \wedge b) \wedge c$ becomes $(a \wedge b \wedge c)$

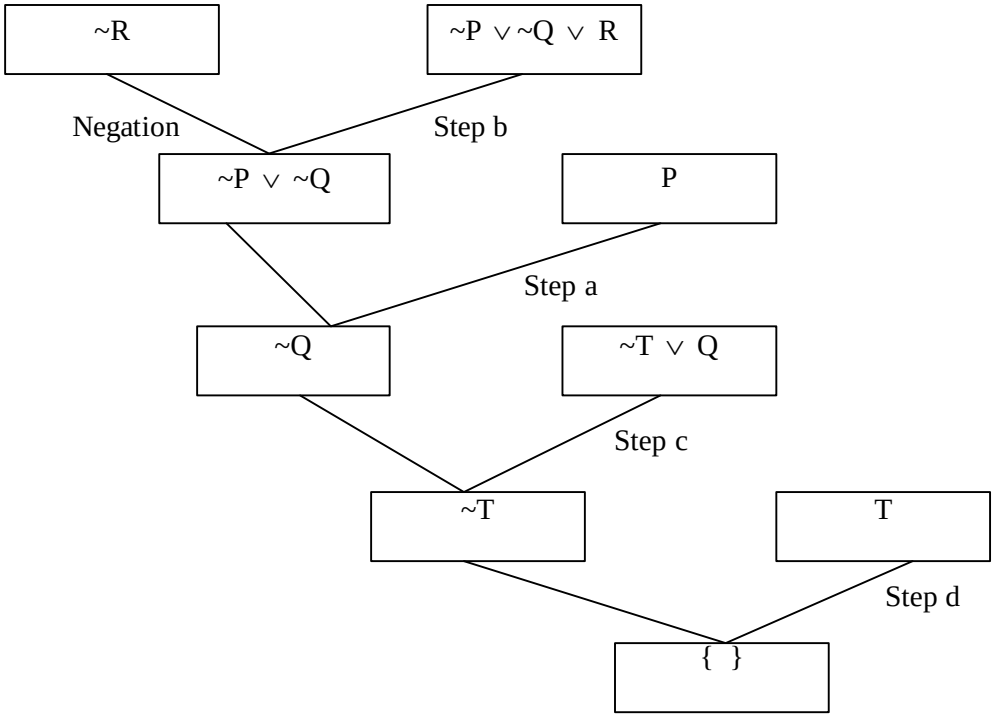
   It is a conjunction where every conjunct is a disjunction of literals.

8. **Convert disjunctions to implications :-**

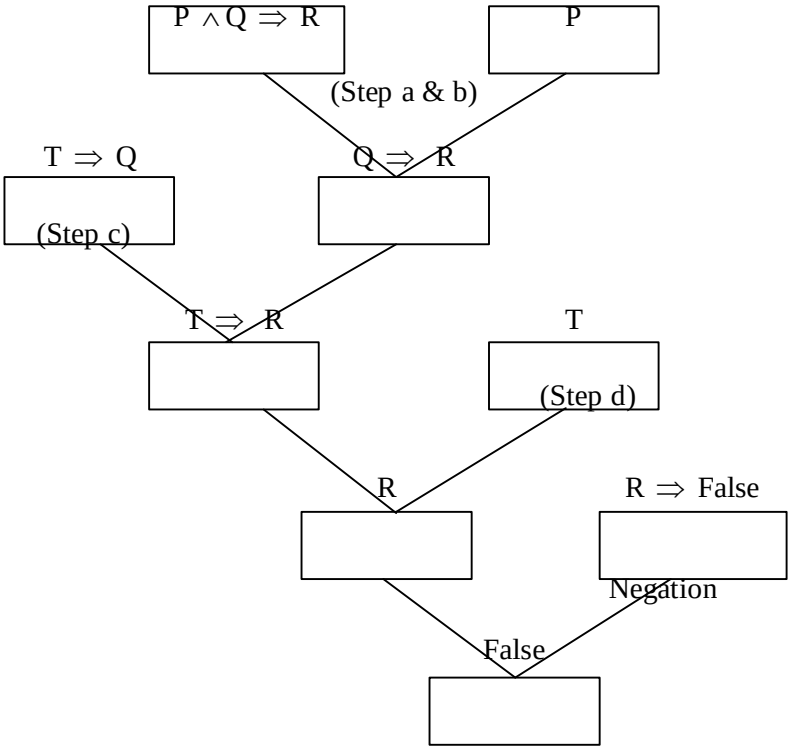
   From the CNF it is possible to derive the INF, (i.e.) combine the negative literals into one list, the positive literals into another and build an implication from them,

   (i.e.) $(\neg a \vee \neg b \vee c \vee d)$ becomes $(a \wedge b \Rightarrow c \vee d)$

- 1**. For Example: -** Covert the given axioms into equivalent clasuses form and prove that R is true using CNF and INF representation.
  - Axioms:
    - P
    - P $\wedge$ Q $\Rightarrow$ R
    - S $\vee$ T $\Rightarrow$ Q
    - T
  - Proof:
    - ✓ Equivalent Conjunctive Clause Form Representation:
      - P
      - $\neg$ P $\vee$ $\neg$ Q $\vee$ R
      - $\neg$ S $\vee$ Q, $\neg$ T $\vee$ Q
      - T
    - ✓ Equivalent INF Representation:
      - P
      - P $\wedge$ Q $\Rightarrow$ R
      - S $\Rightarrow$ Q, T $\Rightarrow$ Q
      - T
- The negation of the given statement (~R) derives a contradiction ({ }).
- Therefore it is proved that R is true.
- The following diagram shows the proof of resolution with refutation using CNF.



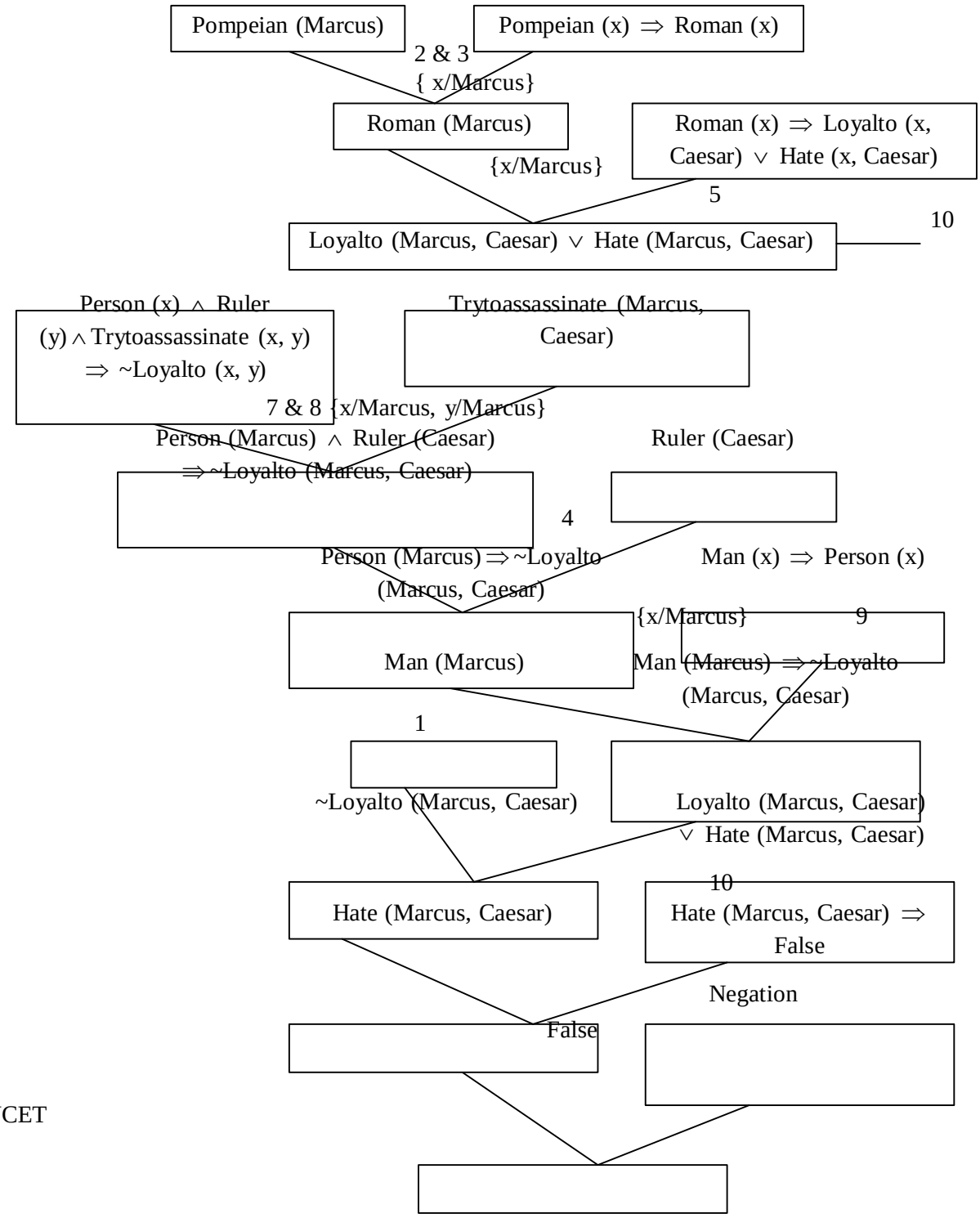
- The negation of the given statements (R ⇒ False) derives a contradiction (True ⇒ False).
- Therefore it is proved that R is True (i.e. Taken assumption is wrong)
- The following diagram shows the proof of Resolution with refutation using INF.



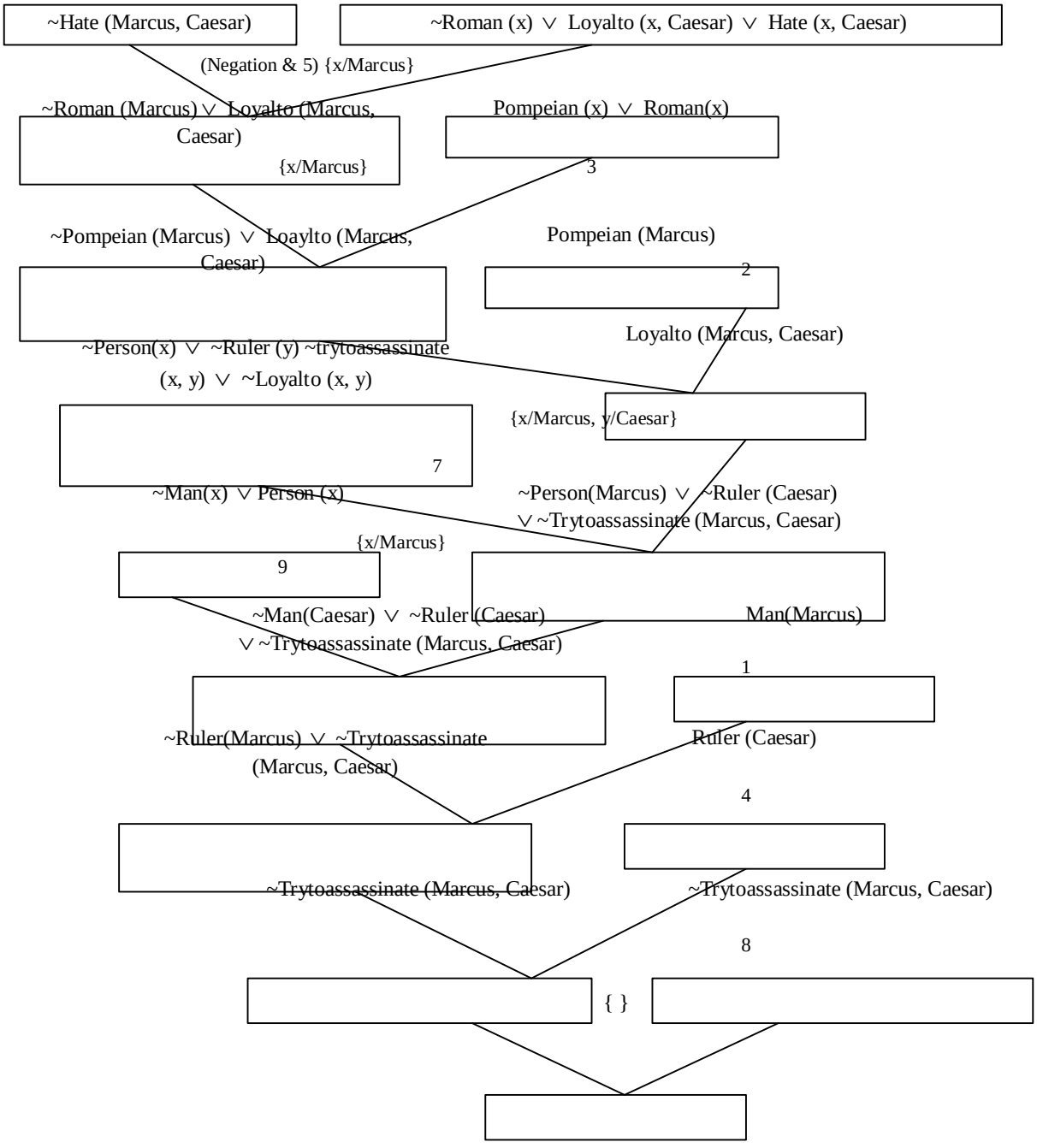
- **2. For Example: -** Convert the given sentences (KB2) into its equivalent FOL representation and then convert it into its CNF and INF representation and solve the task using resolution with refutation proof.

  - Given KB2:-
    - Marcus was a man
    - Marcus was a Pompeian
    - All pompeians were romans
    - Caesar was a ruler
    - All romans were either loyalto Caesar or hated him
    - Everyone is loyal to someone
    - People only try to assassinate rulers they are not loyal to.
    - Marcus tried to assassinate Caesar
    - All man are people
  - **Task: -** Did Marcus hate Caesar?
  - **Solution:-**
    - ✓ **FOL Representation:**
      1. Man (Marcus)
      2. Pompeian (Marcus)
      3. $\forall x$ Pompeian (x) $\Rightarrow$ Roman (x)
      4. Ruler (Caesar)
      5. $\forall x$ Roman (x) $\Rightarrow$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
      6. $\forall x \exists y$ Loyalto (x, y)
      7. $\forall x \ \forall \mathbf{y}$ Person(x) $\wedge$ Ruler(y) $\wedge$ Trytoassassinate(x, y) $\Rightarrow \neg$ Loyalto(x, y)
      8. Trytoassassinate (Marcus, Caesar)
      9. $\forall x$ Man (x) $\Rightarrow$ Person (x)
    - ✓ **INF Representation:**
      1. Man (Marcus)
      2. Pompeian (Marcus)
      3. Pompeian (x) $\Rightarrow$ Roman (x)
      4. Ruler (Caesar)
      5. Roman (x) $\Rightarrow$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
      6. Loyalto (x, y)
      7. Person(x) $\wedge$ Ruler(y) $\wedge$ Trytoassassinate(x, y) $\Rightarrow \neg$ Loyalto(x, y)
      8. Trytoassassinate (Marcus, Caesar)
      9. Man (x) $\Rightarrow$ Person (x)
    - ✓ **CNF Representation :**
      1. Man (Marcus)
      2. Pompeian (Marcus)
      3. ~Pompeian (x) $\vee$ Roman (x)
      4. Ruler (Caesar)
      5. ~Roman (x) $\vee$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
      6. Loyalto (x, y)
      7. ~Person(x) $\vee$~Ruler(y) $\vee$~Trytoassassinate(x, y) $\vee$~Loyalto(x, y)
      8. Trytoassassinate (Marcus, Caesar)
      9. ~Man (x) $\vee$ Person (x)

- To prove the given fact, we assumed the negation of it (i.e.) Hate(Marcus, Caesar) ⇒ False and using inference rules we derive a contradiction False, which means that the assumption must be false, and Hate(Marcus, Caesar) is True.
- Therefore it is proved that Hate (Marcus, Caesar) is True using INF Representation.
- The following diagram shows the proof of Resolution with refutation using INF representation.



- To prove the given fact, we assumed that the negation of it (i.e.) ~ Hate (Marcus, Caesar) and using inference rules, we derive a contradiction as empty set, which means that the assumption must be false, and Hate (Marcus, Caesar) is true in the given KB.
- Therefore it is proved that Hate (Marcus, Caesar) is true using CNF representation.
- The following diagram shows the proof of Resolution with refutation using CNF representation.



.10.4 Dealing with equality:-

- The unification algorithm is used to find a equality between variables with other terms (i.e.) p(x) unifies with p(A), but p(A) and p(B) fails to unify, even if the sentence A=B in the KB.
- The unification does only a syntactic test based on the appearance of the argument terms, not a true semantic test (meaning) based on the objects they represent.
- To solve this problem, one of the way is, the properties can be followed as,

1. $\forall x \quad x = x$ - Reflexive
2. $\forall x, y \quad x = y \Rightarrow y = x$ - Symmetric
3. $\forall x, y, z \quad x = y \wedge y = z \Rightarrow x = z$ - Transitive
4. $\forall x, y \quad x = y \Rightarrow (p_1(x) \Leftrightarrow p_1(y))$ $x = y$, if the predicate name
   $\forall x, y \quad x = y \Rightarrow (p_2(x) \Leftrightarrow p_2(y))$ and arguments are same.
5. $\forall w, x, y, z \quad w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z))$
   $w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z))$

- The other way to deal with equality is **demodulation rule**. For any terms x, y and z where UNIFY (x, z) = θ, defined as :

$$\frac{x = y, (.....z.....)}{(.....SUBST(\theta, y).....)}$$

- A more powerful rule named paramodulation deals with the case where we do not know $x = y$, but we do know $x = y \vee p(x)$.

2.10.5 Resolution strategies:-

- A strategy which is used to guide the search towards a proof of the resolution inference rule. Different types of strategies are as follows,

✓ **Unit Preferences:-**

This strategy prefers a sentence with single literal, to produce a very short sentence

For Example:- P and $P \wedge Q \Rightarrow R$ derives the sentence $Q \Rightarrow R$.

✓ **Set of support:-**

A subset of the sentence are identified (set of support) and resolution combines a set of support with another sentence and adds the resolvent into the set off support, which reduces the size of a sentence in the KB.

Disadvantage: - Bad choice for the set of support will make the algorithm incomplete.

Advantage: - Goal directed proof trees are generated

✓ **Input resolution:-**

This strategy combines one of the input sentences (from the KB or the query) with some other sentence.

For Example:- Resolution proof, Resolution with refutation proof.

Input resolution is complete for KB that are in Horn form, but incomplete in the general case.

✓ **Linear resolution:-**

Two predicates are resolved (P and Q), if either P is in the original KB or if P is an ancestor of Q in the proof tree.

✓ **Subsumption:-**

A strategy which eliminate all sentences that are more specific than the existing sentence.

For example:- if P(x) is in the KB, then there is no sense in adding P(A) to KB.

2.10.6 Theorem Provers (or) Automated reasoners

- Theorem provers use resolution or some other inference procedure to prove sentences in full first order logic, often used for mathematical and scientific reasoning tasks.
- For Example:- MRS, LIFE.

Logic Programming language (PROLOG)	**Theorem Provers**
Handles only the horn clauses	Handles full FOL representation
Choice of writing sentences in different form with same meaning affects the execution order	Does not affects the execution order derives the same conclusion
Example:- writing $A \leftarrow B \wedge C$ instead of $A \leftarrow C \wedge B$ affects the execution of the program	Example:- User can write either $A \leftarrow B \wedge C$ or $A \leftarrow C \wedge B$ or another form $\neg B \leftarrow C \wedge \neg A$ and the results will be exactly same.

Design of a Theorem Prover:-

- An example for theorem prover is: OTTER=> Organized Techniques for Theorem proving and Effective Research, with particular attention to its control strategy.
- To prepare a problem for OTTER, the user must divide the knowledge into four parts;
  1. **SetOfSupport (SOS):** A set of clauses, which defines the important facts about the problem. Resolution step resolves the member of SOS with another axiom.

2. **Usable axioms: -** A set of axioms that are outside the set of support, provides background knowledge about the problem area. The boundary between SOS and the axiom is up to the user's judgement.
3. **Rewrites (or) Demodulators :-** A set of equations evaluated or reduced from left-to-right order (i.e.) x + 0 = x in which x + 0 should be replaced by the term x.
4. A set of parameters and clauses that defines the control strategy that is,
    a. Heuristic function :- to control the search
    b. Filtering function :- eliminates some subgoals which are uninteresting

2.10.7 Execution:-

- Resolution takes place by combining the element of SOS with useful axiom, generally prefers unit strategy.
- The process continuous until a reputation is found or there are no more clauses in the SOS.
- The following shows the algorithm of execution,

Algorithm:-

```
Procedure OTTER(sos, usable)

    Inputs: sos, a set of support-clauses defining the problem (global variable)

            Usable background knowledge potentially relevant to the problem
    Repeat:
            clause          the lightest member of sos
                    ←move clause from sos to usable
                        PROCESS (INFER(clause, usable),sos)
    Until sos = [] or a refutation has been found
function INFER(clause, usable) returns clauses
```

```
            resolve clause with each member of usable
            return the resulting clauses after applying FILTER

Procedure PROCESS (clauses, sos)
```

```
    for each clause in clauses do
        clause          SIMPLIFY (clause)
        merge identical literals
        discard clause if it is a tautology
        sos             [clause | sos]
        if clause has no literals then a refutation has been found
        if clause has one literal then look for unit refutation
    end
```

2.10.8 Extending Prolog:-

- Theorem prover can be build using prolog compiler as a base and a sound complete reasoned of full first order logic is done using PTTP.
- Where PTTP is Prolog Technology Theorem Prover.

- PTTP includes five significant changes to prolog to restore the completeness and expensiveness.

Prolog	PTTP
Depth First Search	Iterative deepening search
Not possible like a PTTP implementation	Negated literals are allowed (i.e.) in the implementation P, ¬ P can be derived using two separate routines
Locking is not allowed	Locking is allowed (i.e.) A clause with n atoms is stored as n different rules. Example:- $A \Leftarrow B \wedge X$ is equivalent to $\neg B \Leftarrow X \wedge \neg A, \neg X \Leftarrow B \wedge \neg A$
Inference is not complete	Inference is complete since refutation is allowed
Unification is less efficient	Unification is more eficient

Drawback of PTTP:-

- Each inference rule is used by the system both in its original form and contrapositive form
  Example:- $(f(x, y) = f(a, b)) \Leftarrow (x = a) \wedge (y = b)$
- Prolog proves that two f terms are equal, But PTTP would also generate the contrapositive
  $(x \neq a) \Leftarrow (f(x, y) \neq f(a, b)) \wedge (y = b)$
- This is a wasteful way to prove that any two terms x and a are different.

****

SECOND UNIT-II LOGICAL REASONING FINISHED

****

ARTIFICIAL INTELLIGENCE

UNIT-III

KNOWLEDGE INFERENCE

Knowledge Representation - Production based System, Frame based System. Inference - Backward Chaining, Forward Chaining, Rule value approach, Fuzzy Reasoning - Certainity factors, Bayesian Theory - Bayesian Network - Dempster Shafer Theory

3.0 Knowledge representation: -

- The task of coming up with a sequence of actions that will achieve a goal is called Planning.
- "*Deciding in ADVANCE what is to be done*"
- A problem solving methodology
- Generating a set of action that are likely to lead to achieving a goal
- Deciding on a course of actions before acting
- **Representation for states and Goals:-**
  - In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.
  - For example,
    At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^….
  - Goals are also described by conjunctions of literals.
  - For example,
    At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)
  - Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as
- **Representation for actions:-**
  - Our STRIPS operators consist of three components:
  - the ***action description*** is what an agent actually returns to the environment in order to do something.
  - the ***precondition*** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
  - the ***effect*** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.
  - Here's an example for the operator for going from one place to another:
    - **Op(Action:Go(there),**
    - **Precond:At(here)^Path(here, there),**
    - **Effect:At(there)^ ¬At(here))**
- **Representation of Plans:-**
  - Consider a simple problem:
  - Putting on a pair of shoes
  - Goal →RightShoeOn ^ LeftShoeOn
  - Four 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)

Given:-

- A description of an initial state
- A set of actions
- A (partial) description of a goal state

Problem:-

- Find a sequence of actions (plan) which transforms the initial state into the goal state.

Application areas:-

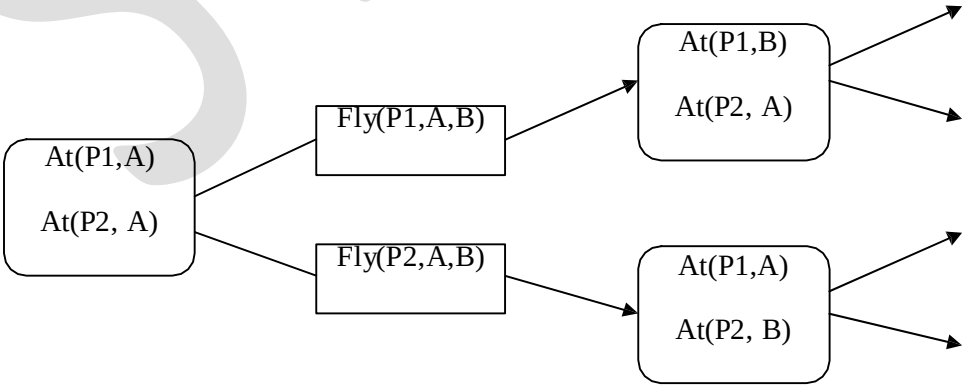
- Systems design
- Budgeting
- Manufacturing product
- Robot programming and control
- Military activities

Benefits of Planning:-

- Reducing search
- Resolving goal conflicts
- Providing basis for error recovery

3.1 Planning with State Space Search:

- Planning with state space search approach is used to construct a planning algorithm.
- This is most straightforward approach.
- The description of actions in a planning problem specifies both preconditions and effects.
- It is possible to search in either direction.
- Either from forward from the initial state or backward from the goal
- The following are the two types of state space search ,
  - Forward state-space search
  - Backward state-space search
- The following diagram shows the Forward state-space search



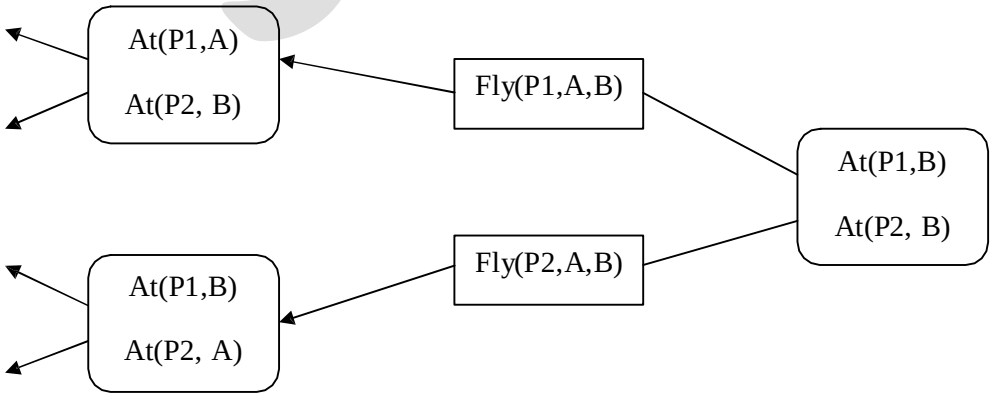
3.1.1 Forward state-space search:-

- Planning with forward state-space search is similar to the problem solving using Searching.

- It is sometimes called as progression Planning.
- It moves in the forward direction.
- we start in the problems initial state, considering sequence of actions until we find a sequence that reaches a goal state.
- The formulation of planning problems as state-space search problems is as follows,
  - The **Initial state** of the search is the initial state from the planning problem.
  - In general, each state will be a set of positive ground literals; literals not appearing are false.
  - The **actions** that are applicable to a state are all those whose preconditions are satisfied.
  - The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals.
  - The goal test checks whether the state satisfies the goal of the planning problem.
  - The step cost of each action is typically 1.
- This method was too inefficient.
- It does not address the irrelevant action problem, (i.e.) all applicable actions are considered from each state.
- This approach quickly bogs down without a good heuristics.
- For Example:-
  - Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo.
  - The Goal is to move the entire cargo form airport A to airport B.
  - There is a simple solution to the Problem,
  - Load the 20 pieces of cargo into one of the planes at A, then fly the plane to B, and unload the cargo.
  - But finding the solution can be difficult because the average branching factor is huge.

3.1.2 Backward state- space search:-

- Backward search is similar to bidirectional search.
- It can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly.
- It is not always obvious how to generate a description of the possible predecessors of the set of goal states.
- The main advantage of this search is that it allows us to consider only relevant actions.
- An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal.
- The following diagram shows the Backward state-space search



- For example:-
  - The goal in our 10-airport cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

    $$At(C1,B) \wedge At(C2,B) \wedge \ldots\ldots \wedge At(C20,B)$$
  - Now consider the conjunct At(C1,B). working backwards, we can seek actions that have this as an effect. There is only one unload(C1,p,B), where plane p is unspecified.
  - In this search restriction to relevant actions means that backward search often has a much lower branching factor than forward search.
- Searching backwards is sometimes called regression planning.
- The principal question is:- what are the states from which applying a given action leads to the goal?
- Computing the description of these states is called regressing the goal through the action.
- consider the air cargo example;- we have the goal as,

  $$At(C1,B) \wedge At(C2,B) \wedge \ldots\ldots \wedge At(C20,B)$$

  and the relevant action Unload(C1,p,B), which achieves the first conjunct.
- The action will work only if its preconditions are satisfied.
- Therefore , any predecessor state must include these preconditions : $In(C1,p) \wedge At(p,B)$, Moreover the subgoal At(C1,B) should not be true in the predecessor state.
- The predecessor description is

  $$In(C1,p) \wedge At(p,B) \wedge At(C2,B) \wedge \ldots\ldots \wedge At(C20,B)$$
- In addition to insisting that actions achieve some desired literal, we must insist that the actions not undo any desired literals.
- An action that satisfies this restriction is called consistent.
- From definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search.
- Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows
  - any positive effects of A that appear in G are deleted
  - Each precondition literal of A is added, unless it already appears
- Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem.

3.1.3 Heuristics for State-space search:-

Heuristic Estimate:-

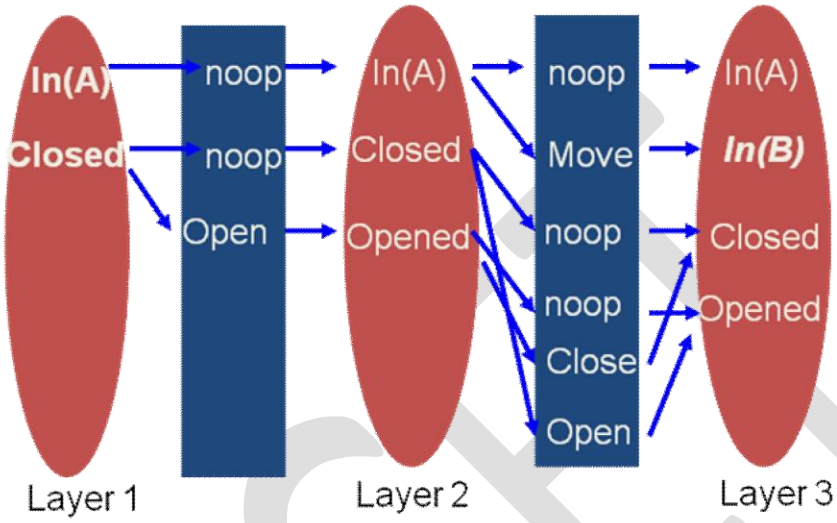
- The value of a state is a measure of how close it is to a goal state.
- This cannot be determined exactly (too hard), but can be approximated.
- One way of approximating is to use the relaxed problem.
  - Relaxation is achieved by ignoring the negative effects of the actions.
  - The relaxed action set, A', is defined by:

    A' = {<pre(a),add(a),0> | a in A}

SVCET

Relaxed Distance Estimate

- Current: In(A), Closed Goal: In(B)



- Layers correspond to successive time points,
- # layers indicate minimum time to achieve goals.

Building the relaxed plan graph:-

- **Start** at the initial state
- **Repeatedly** apply all relaxed actions whose preconditions are satisfied.
  - Their (positive) effects are asserted at the next layer.
- **If** all actions applied and the goals are not all present in the final graph layer
  Then the problem is unsolvable.

Extracting Relaxed solution

- When a layer containing all of the goals is reached ,FF searches *backwards* for a plan.
- The earliest possible achiever is always used for any goal.
  - This maximizes the possibility for exploiting actions in the relaxed plan.
- The relaxed plan might contain many actions happening concurrently at a layer.
- The number of actions in the relaxed plan is an estimate of the true cost of achieving the goals.

How FF uses the Heuristics:-

- FF uses the heuristic to estimate how close each state is to a goal state
  - any state satisfying the goal propositions.

SVCET

- The actions in the relaxed plan are used as a guide to which actions to explore when extending the plan.
- All actions in the relaxed plan at layer i that achieve at least one of the goals required at layer i+1 are considered helpful.
- FF restricts attention to the helpful actions when searching forward from a state.

Properties of the Heuristics:-

- The relaxed plan that is extracted is not guaranteed to be the optimal relaxed plan.

➔ the heuristic is not admissible.

- FF can produce non-optimal solutions.
- Focusing only on helpful actions is not completeness preserving.

➔ Enforced hill-climbing is not completeness preserving.

3.2 Partial Order Planning:-

- Formally a planning algorithm has three inputs:
  - A description of the world in some formal language,
  - A description of the agent's goal in some formal language, and
  - A description of the possible actions that can be performed.
- The planner's o/p is a sequence of actions which when executed in any world satisfying the initial state description will achieve the goal.
- **Representation for states and Goals:-**
  - In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.
  - For example,
    At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^….
  - Goals are also described by conjunctions of literals.
  - For example,
    At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)
  - Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as
- **Representation for actions:-**
  - Our STRIPS operators consist of three components:
  - the ***action description*** is what an agent actually returns to the environment in order to do something.
  - the ***precondition*** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
  - the ***effect*** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.
  - Here's an example for the operator for going from one place to another:
    - **Op(Action:Go(there),**
    - **Precond:At(here)^Path(here, there),**
    - **Effect:At(there)^ ¬At(here))**
- **Representation of Plans:-**
  - Consider a simple problem:
    - Putting on a pair of shoes
    - Goal →RightShoeOn ^ LeftShoeOn
    - Four 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)

- **Least Commitment:-** The general strategy of delaying a choice during search is called Least commitment.
- **Partial-order Planner:-** Any planning algorithm that can place two actions into a plan without specifying which come first is called a partial order planner.
- **Linearization:-** The partial-order solution corresponds to six possible total order plans ; each of these is called a linearization of the partial order plan.
- **Total order planner:-** Planner in which plans consist of a simple lists of steps.
- A plan is defined as a data structure
  - A set of plan steps
  - A set of step ordering
  - A set of variable binding constraints
  - A set of causal links : $s_i \xrightarrow{c} s_j$
    "s_i achieves c for s_j"
- Initial plan before any refinements
  Start < Finish
  Refine and manipulate until a plan that is a solution

$$
\begin{aligned}
&Plan(\text{STEPS}:\{\ S_1 : Op(\text{ACTION}:Start),\\
&\qquad S_2 : Op(\text{ACTION}:Finish,\\
&\qquad\qquad \text{PRECOND}:RightShoeOn \wedge LeftShoeOn)\},\\
&\text{ORDERINGS}: \{S_1 \prec S_2\},\\
&\text{BINDINGS}: \{\},\\
&\text{LINKS}: \{\})
\end{aligned}
$$

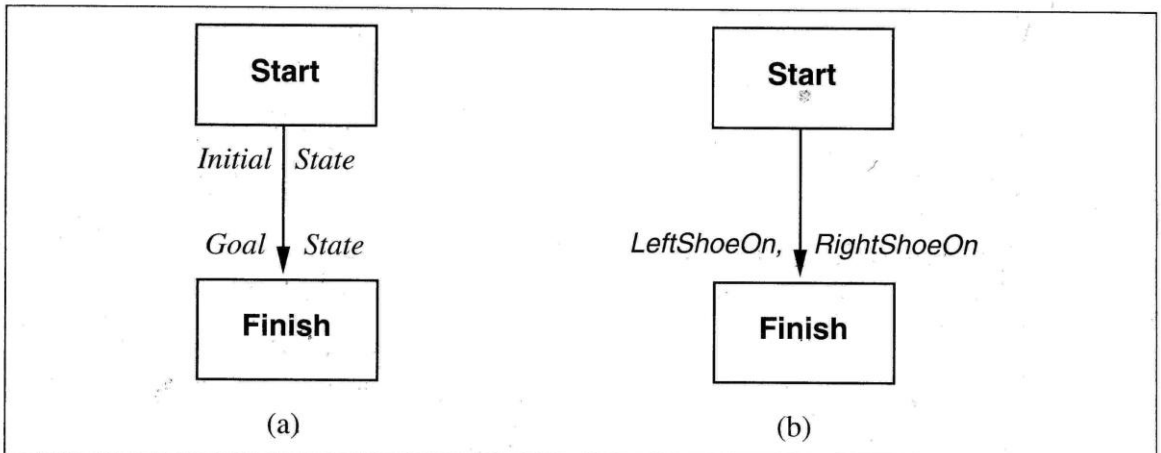
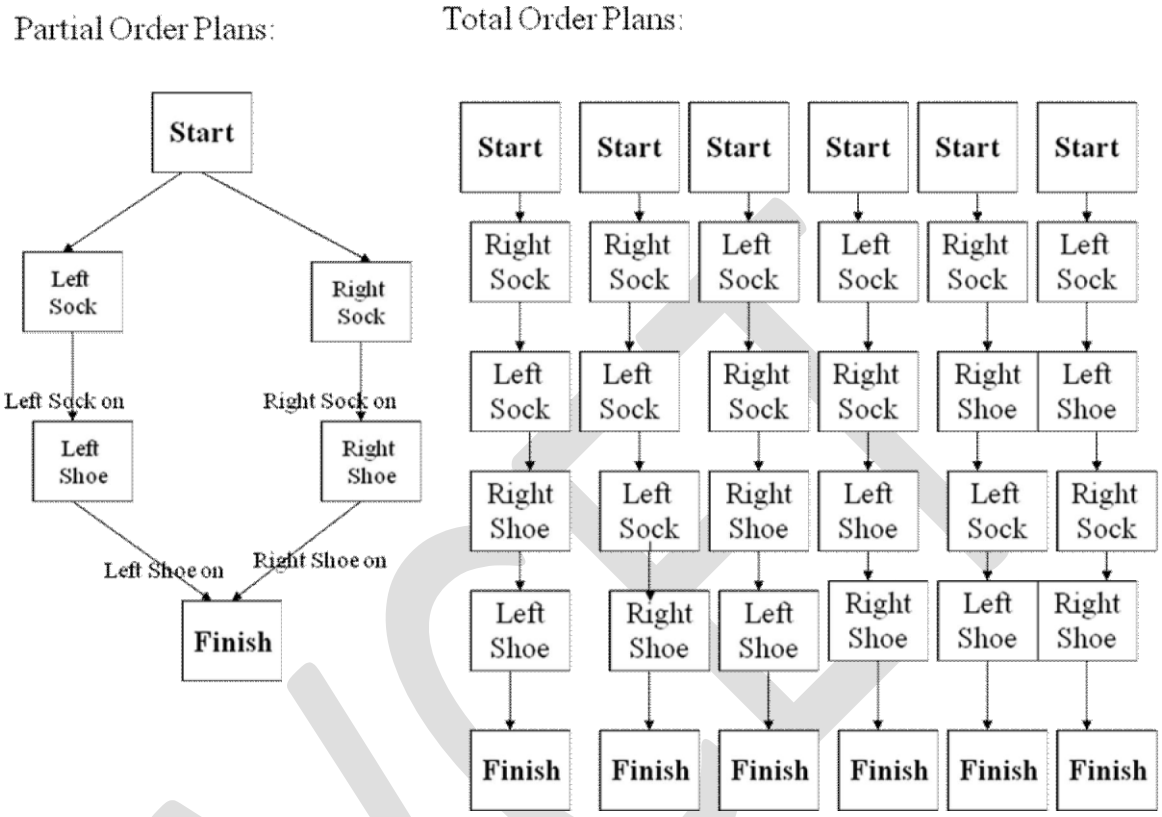


Figure 11.4 (a) Problems are defined by partial plans containing only *Start* and *Finish* steps. The initial state is entered as the effects of the *Start* step, and the goal state is the precondition of the *Finish* step. Ordering constraints are shown as arrows between boxes. (b) The initial plan for the shoes-and-socks problem.

- The following diagram shows the partial order plan for putting on shoes and socks, and the six corresponding linearization into total order plans.



- Solutions
  - solution : a plan that an agent guarantees achievement of the goal
  - a solution is a complete and consistent plan
  - a complete plan : every precondition of every step is achieved by some other step
  - a consistent plan : no contradictions in the ordering or binding constraints. When we meet a inconsistent plan we backtrack and try another branch

3.2.1 Partial order planning Algorithm:-

The following is the Partial order planning algorithm,

```
function pop(initial-state, conjunctive-goal, operators)
 // non-deterministic algorithm
 plan = make-initial-plan(initial-state, conjunctive-goal);
 loop:
  begin
   if solution?(plan) then return plan;
   (S-need, c) = select-subgoal(plan) ; // choose an unsolved goal
   choose-operator(plan, operators, S-need, c);
      // select an operator to solve that goal and revise plan
   resolve-threats(plan);  // fix any threats created
  end
```

```
end

function solution?(plan)
  if causal-links-establishing-all-preconditions-of-all-steps(plan)
     and all-threats-resolved(plan)
     and all-temporal-ordering-constraints-consistent(plan)
     and all-variable-bindings-consistent(plan)
  then return true;
else return false;
end

function select-subgoal(plan)
  pick a plan step S-need from steps(plan) with a precondition c
     that has not been achieved;
  return (S-need, c);
end

procedure choose-operator(plan, operators, S-need, c)
  // solve "open precondition" of some step
  choose a step S-add by either
    Step Addition: adding a new step from operators that
       has c in its Add-list
    or Simple Establishment: picking an existing step in Steps(plan)
       that has c in its Add-list;
  if no such step then return fail;
  add causal link "S-add --->c S-need" to Links(plan);
  add temporal ordering constraint "S-add < S-need" to Orderings(plan);
  if S-add is a newly added step then
    begin
    add S-add to Steps(plan);
    add "Start < S-add" and "S-add < Finish" to Orderings(plan);
    end
end

procedure resolve-threats(plan)
  foreach S-threat that threatens link "Si --->c Sj" in Links(plan)
   begin     // "declobber" threat
     choose either
       Demotion: add "S-threat < Si" to Orderings(plan)
       or Promotion: add "Sj < S-threat" to Orderings(plan);
     if not(consistent(plan)) then return fail;
   end
end
```

- **Partial Order Planning Example:-**
  - Shopping problem: “get milk, banana, drill and bring them back home”
  - assumption
    1)Go action “can travel the two locations”
    2)no need money

- initial state : operator start
  Op(ACTION:Start,EFFECT:At(Home) ∧ Sells(HWS,Drill) ∧ Sells(SM,Milk), Sells(SM,Banana))
- goal state : Finish
  Op(ACTION:Finish, PRECOND:Have(Drill) ∧ Have(Milk) ∧ Have(Banana) ∧ At(Home))
- actions:
  Op(ACTION:Go(there),PRECOND:At(here),EFFECT:At(there) ∧ ¬At(here))
  Op(ACTION:Buy(x),PRECOND:At(store) ∧ Sells(store,x),EFFECT:Have(x))

- There are many possible ways in which the initial plan elaborated
  - one choice : three Buy actions for three preconditions of Finish action
  - second choice:sells precondition of Buy
    - Bold arrows:causal links, protection of precondition
    - Light arrows:ordering constraints

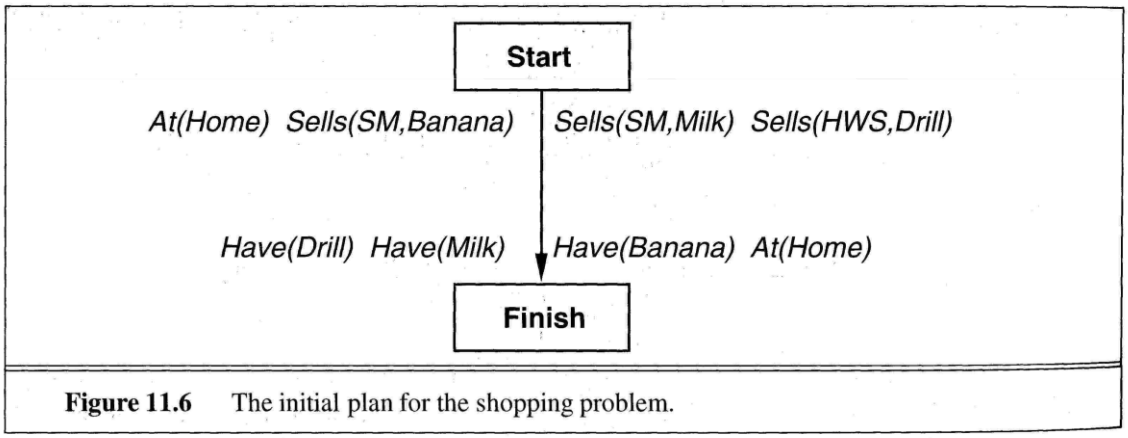
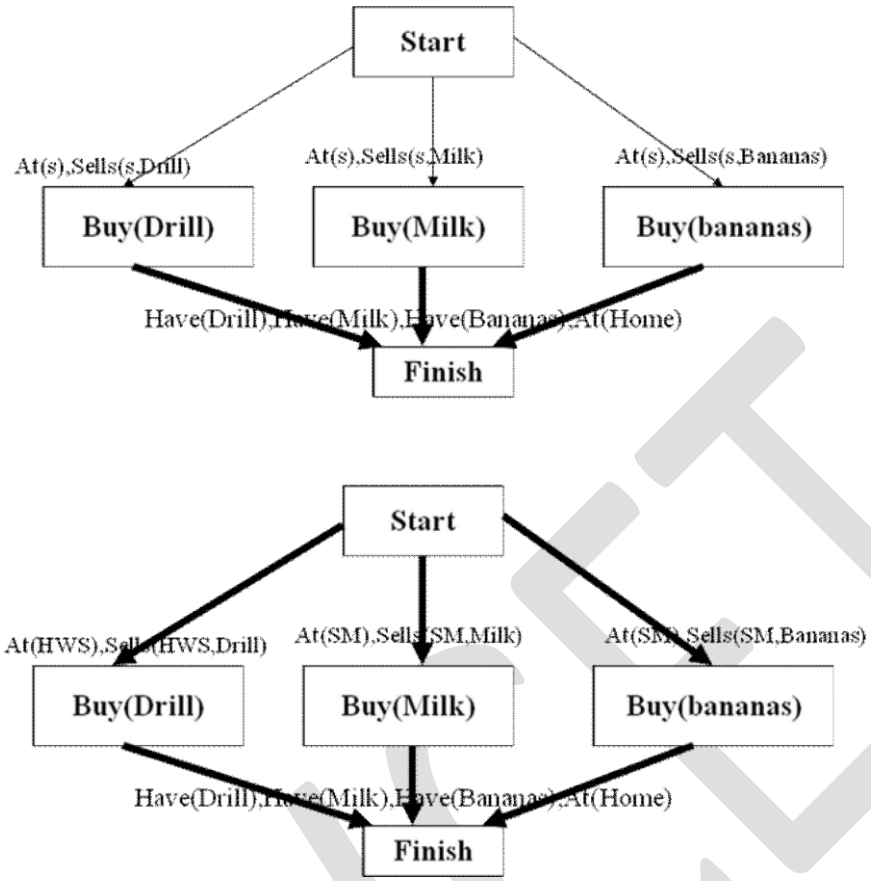


Figure 11.6 The initial plan for the shopping problem.

- The following diagram shows the,
  - partial plan that achieves three of four preconditions of finish
  - Refining the partial plan by adding casual links to achieve the sells preconditions of the buy steps

SVCET



- causal links : protected links
  a causal link is protected by ensuring that threats are ordered to come before or after the protected link
- demotion : placed before
  promotion : placed after

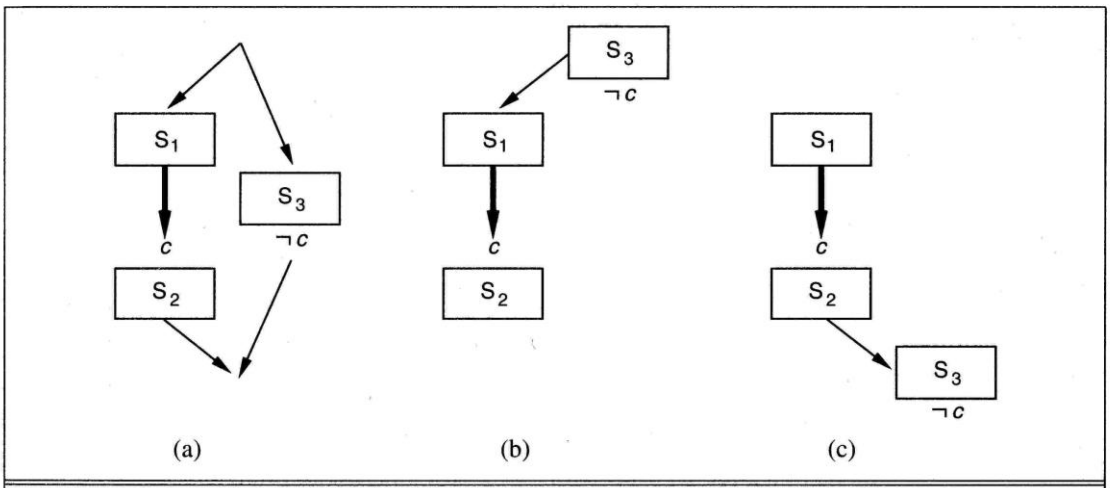
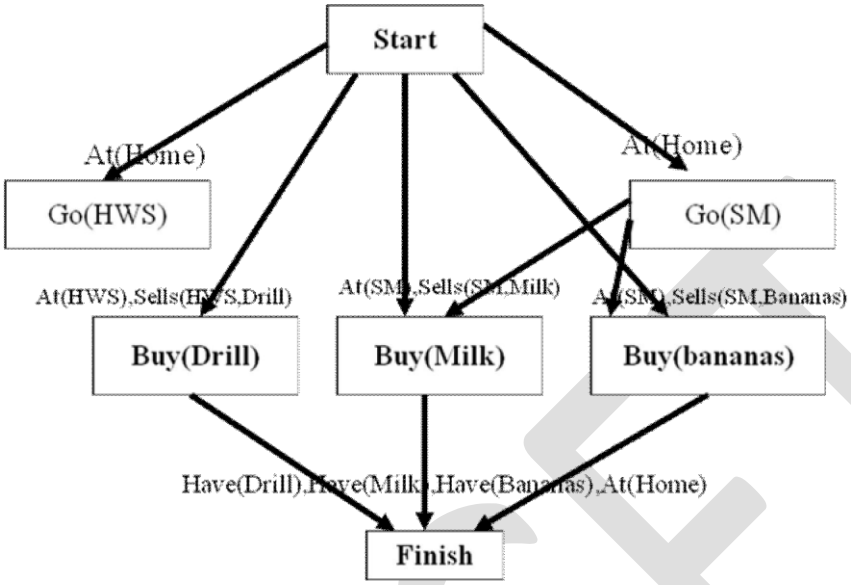


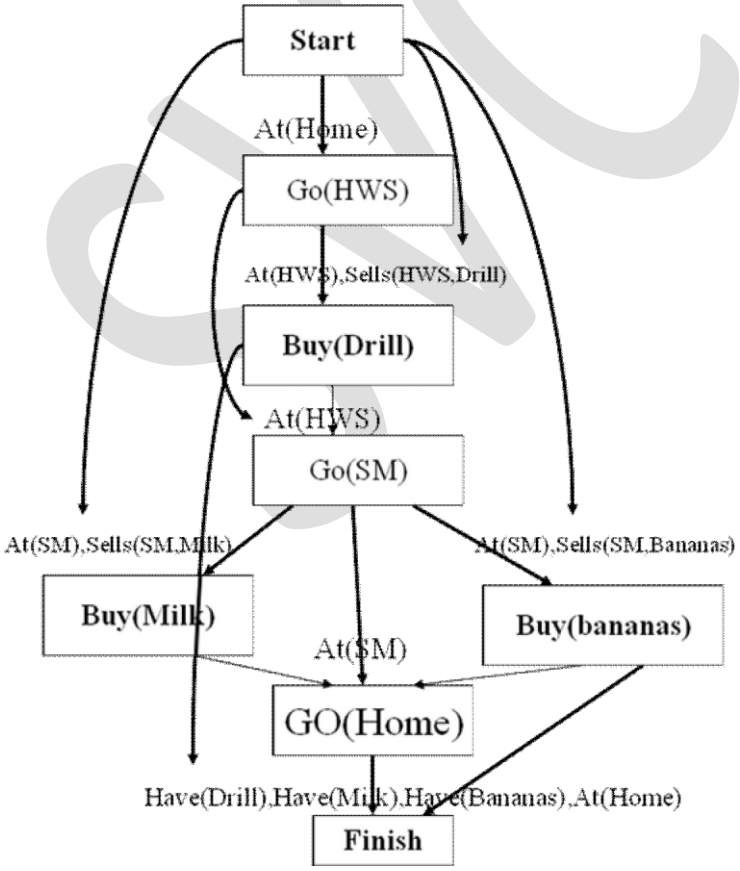
Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

SVCET

- The following diagram shows the partial plan that achieves At Precondition of the three buy conditions



- The following diagram shows the solution of this problem,



- The following are the Knowledge engineering for plan,
- Methodology for solving problems with the planning approach
  (1) Decide what to talk about
  (2) Decide on a vocabulary of conditions, operators, and objects
  (3) Encode operators for the domain
  (4) Encode a description of the specific problem instance
  (5) pose problems to the planner and get back plans
- (ex) The blocks world
  - (1) what to talk about
    - cubic blocks sitting on a table
    - one block on top of another
    - A robot arm pick up a block and moves it to another position
  - (2) Vocabulary
    - objects:blocks and table
    - On(b,x) : b is on x
    - Move(b,x,y) : move b form x to y
    - ¬exist x On(x,b) or ∀x ¬On(x,b) : precondition
    - clear(x)
  - (3)Operators
    Op(ACTION:Move(b,x,y),
    PRECOND:On(b,x) ∧ Clear(b) ∧ Clear(y),
    EFFECT:On(b,y) ∧ Clear(x) ∧ ¬On(b,x) ∧ ¬Clear(y))
    Op(ACTION:MoveToTable(b,x),
    PRECOND:On(b,x) ∧ Clear(b),
    EFFECT:On(b,Table) ∧ Clear(x) ∧ ¬On(b,x))

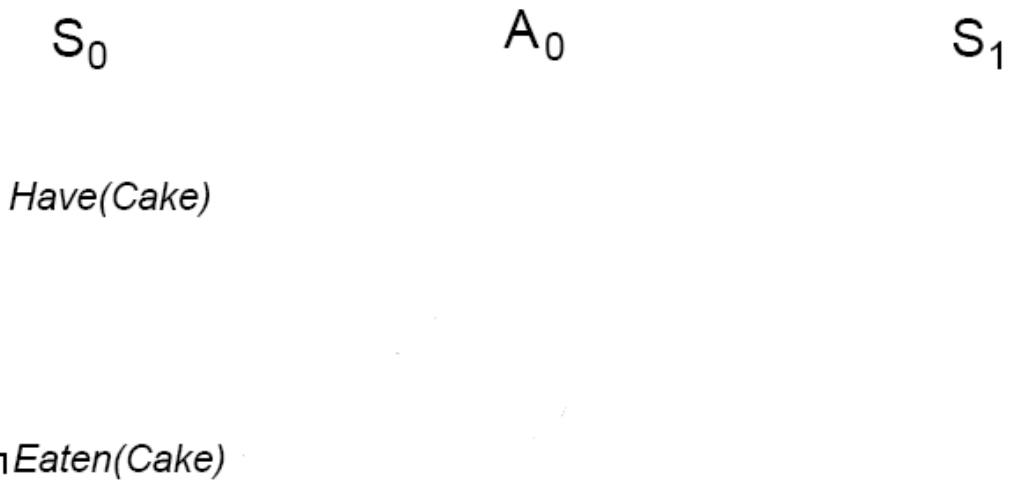
3.3 Planning Graph:-

- Planning graphs are an efficient way to create a representation of a planning problem that can be used to
  - Achieve better heuristic estimates
  - Directly construct plans
- Planning graphs only work for propositional problems.
- Planning graphs consists of a seq of levels that correspond to time steps in the plan.
  - Level 0 is the initial state.
  - Each level consists of a set of literals and a set of actions that represent what *might be* possible at that step in the plan
  - *Might be* is the key to efficiency
  - Records only a restricted subset of possible negative interactions among actions.
- Each level consists of
  - *Literals* = all those that *could* be true at that time step, depending upon the actions executed at preceding time steps.
  - *Actions* = all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.
- For Example:-

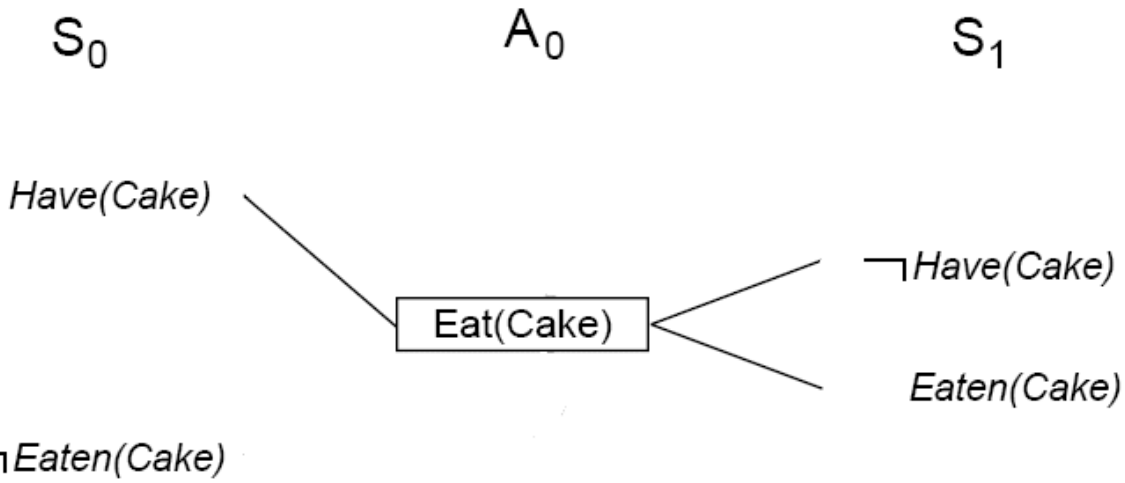
  Init(Have(Cake))
  Goal(Have(Cake) ∧ Eaten(Cake))

Action(Eat(Cake),
PRECOND: Have(Cake)
EFFECT: ¬Have(Cake) ∧ Eaten(Cake))
Action(Bake(Cake),
PRECOND: ¬ Have(Cake)
EFFECT: Have(Cake))

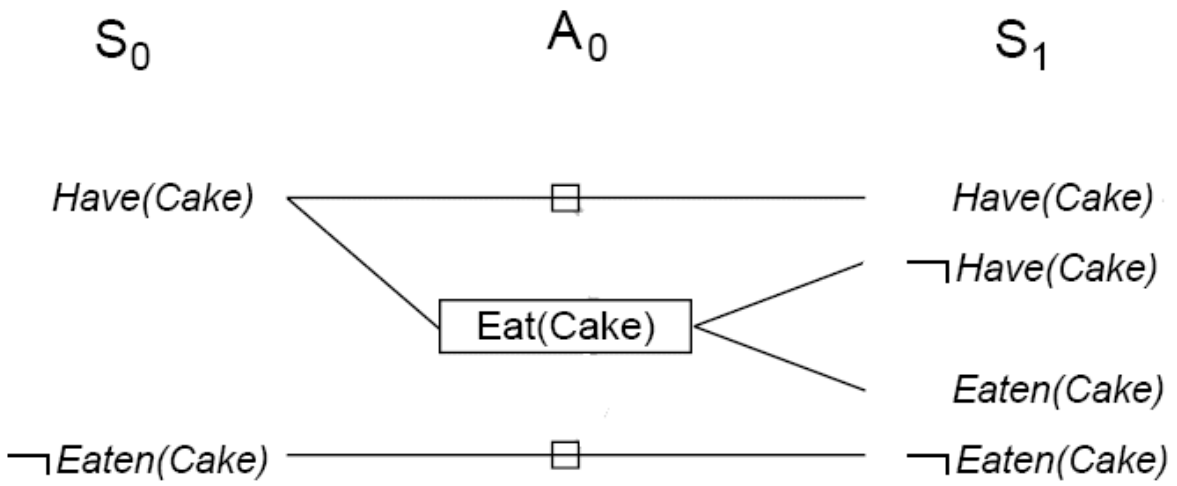
- Steps to create planning graph for the example,
  - Create level 0 from initial problem state.



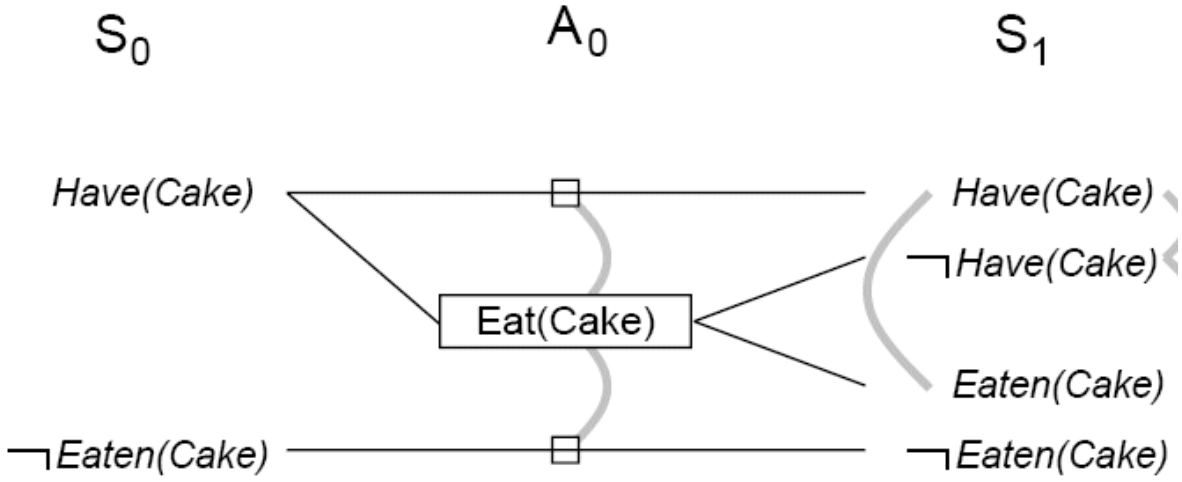
  - Add all applicable actions.
  - Add all effects to the next state.



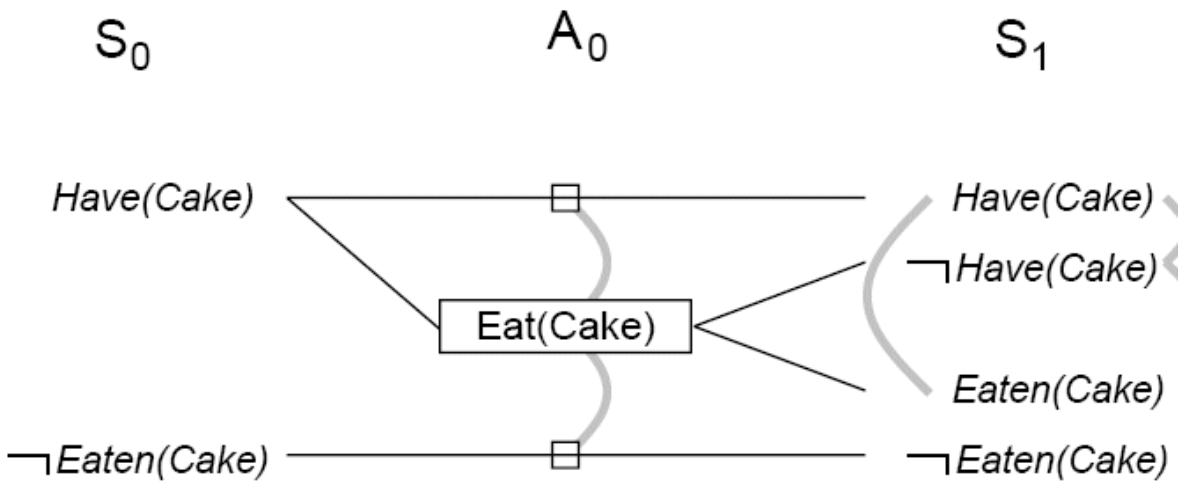
  - Add *persistence actions* (inaction = no-ops) to map all literals in state S_i to state S_{i+1}.



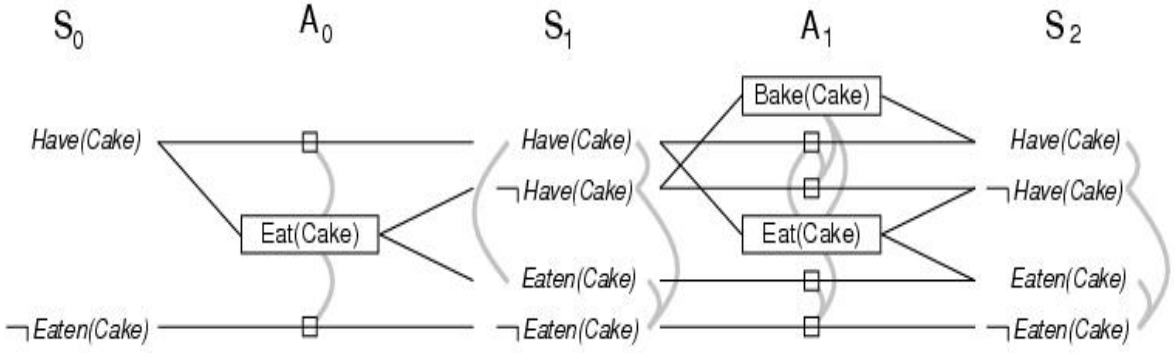
- Identify *mutual exclusions* between actions and literals based on potential conflicts.



- Mutual Exclusion:-
  - A mutex relation holds between **two actions** when:
    - *Inconsistent effects*: one action negates the effect of another.
    - *Interference*: one of the effects of one action is the negation of a precondition of the other.
    - *Competing needs*: one of the preconditions of one action is mutually exclusive with the precondition of the other.
  - A mutex relation holds between **two literals** when:
    - one is the negation of the other OR
    - each possible action pair that could achieve the literals is mutex (inconsistent support).
- Level S_1 contains all literals that could result from picking any subset of actions in A_0
  - Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  - S1 defines multiple states and the mutex links are the constraints that define this set of states.



- Repeat process until graph levels off:
  - two consecutive levels are identical, or
  - contain the same amount of literals (explanation follows later)



- In figure
  - rectangle denotes actions
  - small square denotes persistence actions
  - straight lines denotes preconditions and effects
  - curved lines denotes mutex links

3.3.1 Planning Graphs for Heuristic Estimation:-

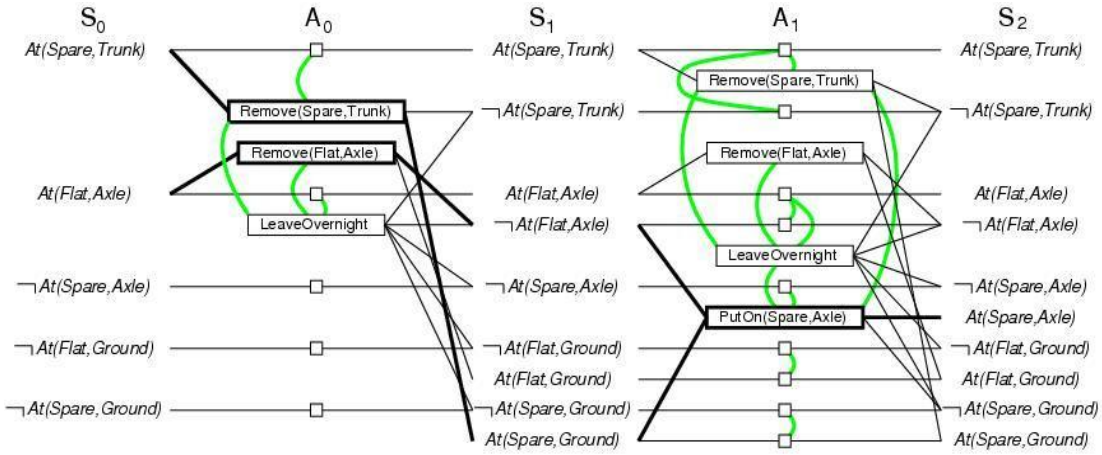
- PG's provide information about the problem
  - PG is a relaxed problem.
  - A literal that does not appear in the final level of the graph cannot be achieved by any plan.
    - $H(n) = \infty$
  - Level Cost: First level in which a goal appears
    - Very low estimate, since several actions can occur

  - Improvement: restrict to one action per level using *serial PG* (add mutex links between *every* pair of actions, except persistence actions).
- Cost of a conjunction of goals
  - Max-level: maximum first level of any of the goals
  - Sum-level: sum of first levels of all the goals
  - Set-level: First level in which all goals appear without being mutex
- The following is the GraphPlan Algorithm,
- Extract a solution directly from the PG

function GRAPHPLAN(*problem*) **return** *solution* or failure
graph ← INITIAL-PLANNING-GRAPH(*problem*)
goals ← GOALS[*problem*]
loop do
if *goals* all non-mutex in last level of graph **then do**
solution ← EXTRACT-SOLUTION(*graph, goals,* LENGTH*(graph)*)
if *solution* ≠ failure **then return** *solution*
else if NO-SOLUTION-POSSIBLE(*graph*) **then return** failure
graph ← EXPAND-GRAPH(*graph, problem*)

- Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si
- EXPAND-GRAPH also looks for mutex relations
  - Inconsistent effects
    - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and **not** At(Spare, Ground)
  - Interference
    - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and **not** At(Flat,Axle) as EFFECT
  - Competing needs
    - E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and **not** At(Flat, Axle)
  - Inconsistent support
    - E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

- In S2, the goal literals exist and are not mutex with any other
  - Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
  - Initial state = last level of PG and goal goals of planning problem
  - Actions = select any set of non-conflicting actions that cover the goals in the state
  - Goal = reach level S0 such that all goals are satisfied
  - Cost = 1 for each action.



3.3.2 Termination of GraphPlan:-

- Termination? YES
- PG are monotonically increasing or decreasing:
  - Literals increase monotonically: - Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; Once a literal shows up, persistence actions cause it to say forever.
  - Actions increase monotonically:- Once a literal appears at a given level, it will appear at all subsequent levels. This is a consequence of literals increasing; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus will the action
  - Mutexes decrease monotonically:- If two actions are mutex at a given level A_i, then they will also be mutex for all previous levels at which they both appear.
- Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off

3.4 Planning and Acting in the Real World:

- In which we see how more expressive representation and more interactive agent architectures lead to planners that are useful in the real world.
- Planners that are used in the real world for tasks such as scheduling,
  - Hubble Space Telescope Observations
  - Operating factories
  - handling the logistics for military campaigns

3.4.1 Time, Schedules and Resources:

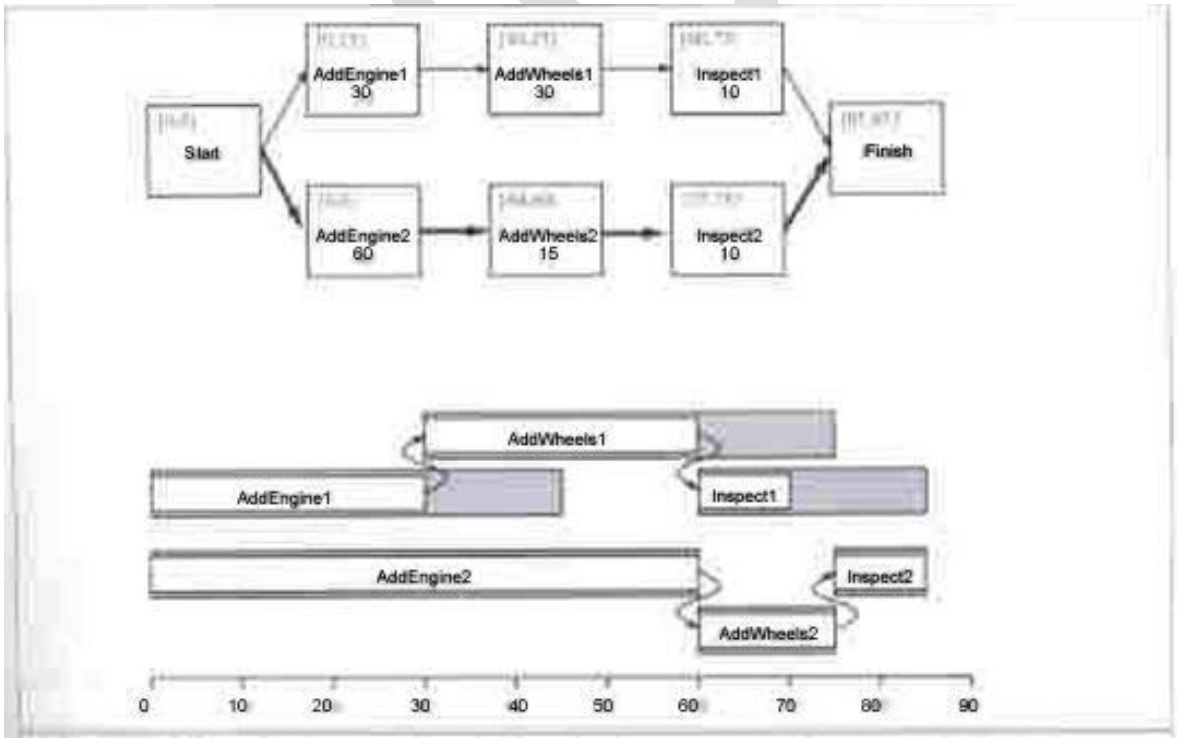
- Time is the essence in the general family of applications called **Job Shop Scheduling.**
- Such a tasks require completing a set of jobs, each of which consists of a sequence of actions, where each action has a given duration and might require some resources.
- The problem is to determine a schedule that minimizes the total time required to complete all the jobs, while respecting the resource constraints.
- For Example:- The following problem is a job shop scheduling.

Init (chassis(C1) ∧ chassis(C2)

∧ Engine (E1,C1,30) ∧ Engine (E2,C2,60)
∧ Wheels (W1,C1,30) ∧ Wheels (W2,C2,15))
Goal (Done(C1) ∧ Done(C2))

Action (AddEngine(e,c,m),
PRECOND: Engine(e,c,d) ∧ chassis(c) ∧ ¬EngineIn(c),
EFFECT: EngineIn(c) ∧ Duration (d))
Action (AddWheels(w,c),
PRECOND: Wheels(w,c,d) ∧ chassis(c),
EFFECT: WheelsOn(c) ∧ Duration (d))
Action (Inspect(c),
PRECOND: EngineIn(c) ∧ WheelsOn (c) ∧ chassis (c),
EFFECT: Done (c) ∧ Duration(10))

- The above table shows the Job Shop scheduling problem for assembling two cars.
- The notation Duration (d) means that an action takes d minutes to execute.
- Engine(E1,C1,30) means that E1 is an Engine that fits into chassis C1 and takes 30 minutes to Install
- The problem can be solved by POP (Partial order planning).
- We must now determine when each action should begin and end.
- The following diagram shows the solution for the above problem
- To find the start and end times of each action apply the Critical Path Method CPM.
- The critical path is the one that is the longest and upon which the other parts of the process cannot be shorter than.



- At the top, the solution is given as a partial order plan.

- The duration of each action is given at the bottom of each rectangle, with the earliest and latest start time listed as [ES, LS] in the upper left.
- The difference between these two numbers is the slack of an action
- Action with zero slack are on the critical path, shown with bold arrows.
- At the bottom of the figure the same solution is shown as timeline.
- Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected.
- The unoccupied portion of a grey rectangle indicates the slack.
- The following formula serve as a definition for ES and LS and also as the outline of a dynamic programming algorithm to compute them:

$$ES(Start) = 0$$
$$ES(B) = \max_{A \prec B} ES(A) + Duration(A)$$
$$LS(Finish) = ES(Finish)$$
$$LS(A) = \min_{A \prec B} LS(B) - Duration(A)$$

- The complexity of the critical path algorithm is just O(Nb).
- where N is the number of actions and b is the branching factor.

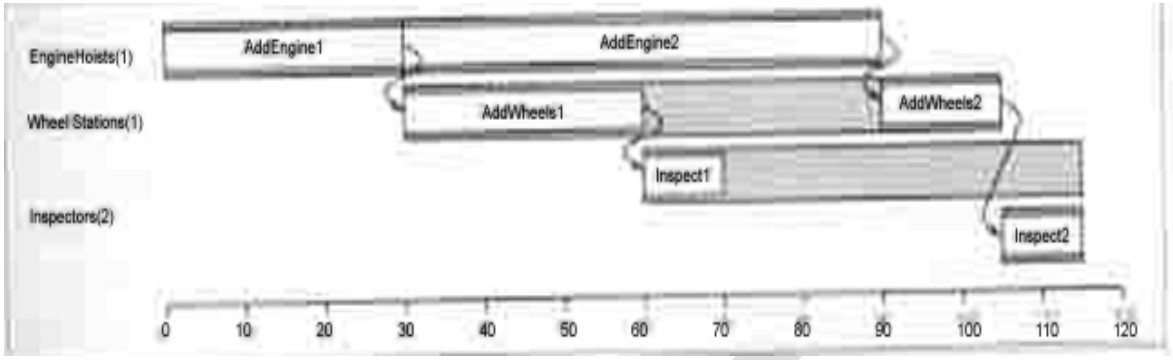
Scheduling with resource constraints:

- Real scheduling problems are complicated by the presence of constraints on resources.
- Consider the above example with some resources.
- The following table shows the job shop scheduling problem for assembling two cars, with resources.

```
Init (chassis(C1) ∧ chassis(C2)
     ∧ Engine (E1,C1,30) ∧ Engine (E2,C2,60)
     ∧ Wheels (W1,C1,30) ∧ Wheels (W2,C2,15)
     ∧ EngineHoists (1) ∧ WheelStations (1) ∧ Inspectors (2))
Goal (Done(C1) ∧ Done(C2))

Action (AddEngine(e,c,m),
     PRECOND: Engine(e,c,d) ∧ chassis(c) ∧ ¬ EngineIn(c),
     EFFECT: EngineIn(c) ∧ Duration (d)
     RESOURCE: EngineHoists (1))
Action (AddWheels(w,c),
     PRECOND: Wheels(w,c,d) ∧ chassis(c),
     EFFECT: WheelsOn(c) ∧ Duration (d),
     RESOURCE: WheelStations (1))
Action (Inspect(c),
     PRECOND: EngineIn(c) ∧ WheelsOn (c) ∧ chassis (c),
     EFFECT: Done (c) ∧ Duration(10),
     RESOURCE: Inspectors (1))
```

- The available resources are on engine assembly station, one wheel assembly station, and two inspectors.
- The notation RESOURCE: means that the resource r is used during execution of an action, but becomes free again when the action is complete.
- The following diagram shows the solution to the job shop scheduling with resources.



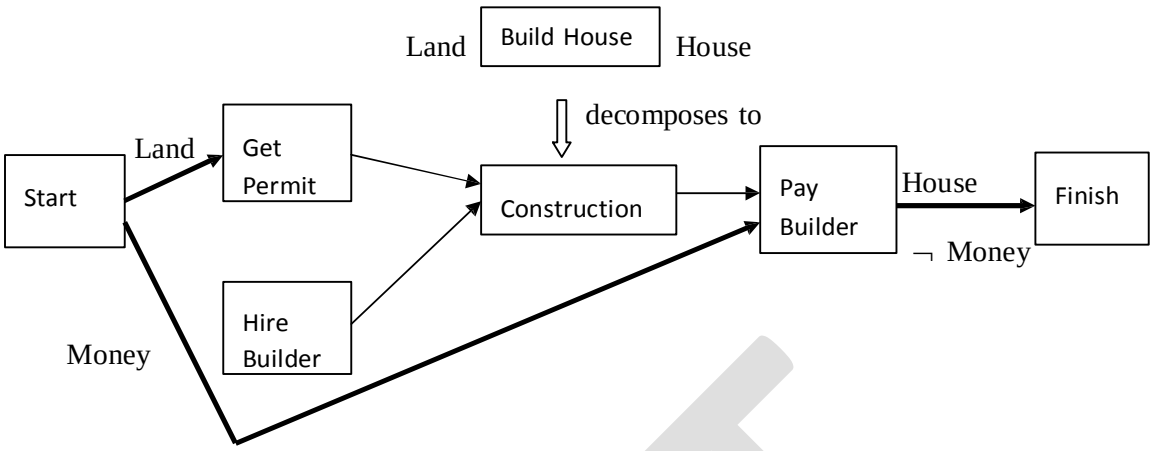
- The left hand margin lists the three resources
- Actions are shown aligned horizontally with the resources they consume.
- There are two possible schedules, depending on which assembly uses the engine station first.
- One simple but popular heuristic is the minimum slack algorithm.
- it schedules actions in a greedy fashion.
- On each iteration, it considers the unscheduled actions that have had all their predecessors scheduled and schedules the one with the least slack for the earliest possible start.
- It then updates the ES and LS times for each affected action and repeats.
- The heuristics is based on the same principle as the most-constrained variable heuristic in constraint satisfaction.

3.4.2 Hierarchical Task Network Planning:

- One of the most pervasive ideas for dealing with complexit y is Hierarchical Decomposition.
- The key benefit of hierarchical structure structure is that, at each level of the hierarchy is reduced to a small number of activities at the next lower level
- So that the computational cost of finding the correct way to arrange those activities for the current problem is small.
- A planning method based on Hierarchical Task Networks or HTNs.
- This approach we take combines ideas from both partial-order planning and the area known as "HTN planning".
- In HTN planning, the initial plan, which describes the problem, is viewed as very high-level description of what is to be done. **For Example: -** Building a House.
- Plans are refined by applying a action decompositions.
- Each action decompositions reduces a high-level action to a partially ordered set of lower-level actions

3.4.2.1 Representing action decompositions:

- The following diagram shows the decomposition of a Building a house action.



- In pure HTN planning, plans are generated only by successive action decompositions.
- Therefore the HTN views planning as a process of making an activity description more concrete, rather than a process of constructing an activity description, starting from the empty activity.
- The action decompositions are represented as, action decompositions methods are stored in a plan library
- From which they are extracted and instantiated to fit the needs of the plan being constructed.
- Each method is an expression of the form Decompose (a, d).
- It means that an action a can be decomposed into the plan d, which is represented as a partial ordered plan.
- The following table shows the action descriptions for the house-building problem and a detailed decomposition for the BuildHouse action.
- The start action of the decomposition supplies all those preconditions of actions in the plan that are not supplied by other actions, such a things called external preconditions.
- In our example external preconditions are land and money.
- Similarly, the external effects, which are the preconditions of Finish, are all those effects of actions in the plan that are not negated by other actions.

Action (BuyLand, PRECOND: Money, EFFECT: Land $\wedge \neg$ Money)
Action (GetLoan, PRECOND: GoodCredit, EFFECT:Money $\wedge$ Mortgage)
Action (BuildHouse, PRECOND: Land, EFFECT: House)

Action (GetPermit, PRECOND: Land, EFFECT: Permit)
Action (HireBuilder, EFFECT: Contract)
Action (Construction, PRECOND: Permit $\wedge$ Contract, EFFECT: HouseBuilt $\wedge \neg$ Permit)
Action (PayBuilder, PRECOND: Money $\wedge$ HouseBuilt, EFFECT: $\neg$ Money $\wedge$ House $\wedge$ $\neg$ Contract)

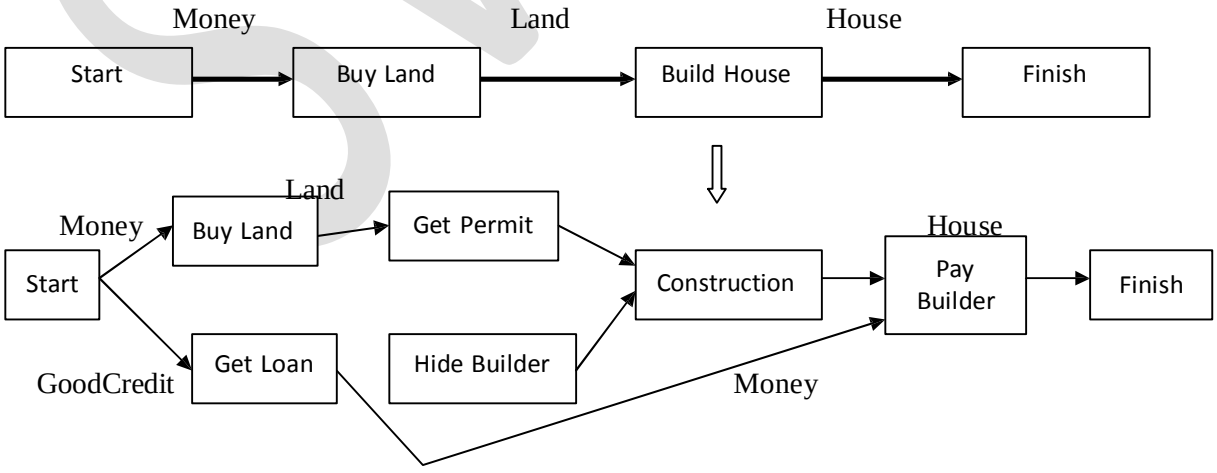
Decompose (BuildHouse,
Plan (Steps : {S1: GetPermit, S2: HireBuilder, S3: Construction, S4: PayBuilder}
ORDERINGS: {Start $\prec$ S1 $\prec$ S3 $\prec$ S4 Finish, Start $\prec$ S2 $\prec$ S3},
Links: {Start $\underline{\text{Land}}$ S1, Start $\underline{\text{Money}}$ S4, S1 $\underline{\text{permit}}$ S3, S2 $\underline{\text{Contract}}$ S3, S3 $\underline{\text{HouseBuilt}}$ S4,
S4 $\underline{\text{House}}$ Finish, S4 $\underline{\neg \text{ Money}}$ Finish}))

- Decomposition should be a correct implementation of the action.

- A plan library could contain several decompositions for any given high-level action.
- Decomposition should be a correct plan, but it could have additional preconditions and effects beyond those stated in the high-level action description.
- The precondition of the high-level action should be the intersection of the external preconditions of its decomposition.
- In which two other forms of information hiding should be noted as,
- First the high-level description completely ignores all internal effects of the decompositions
- Second the high-level description does not specify the intervals "inside" the activity during which the high-level preconditions are effects must hold.
- Information hiding of this kind is essential if hierarchical planning is to reduce complexity.

3.4.2.2 Modifying the planner for decomposition:

- In this we will see how to modify the Partial Order Planning to incorporate HTN planning.
- We can do that by modifying the POP successor function to allow decomposition methods to be applied to the current partial plan P.
- The new successor plans are formed by first selecting some non-primitive action a' in P and then, for any Decompose (a, d) method from the plan library such that a and a' unify with substitution θ, replacing a' with d' = SUBST (θ, d)
- The following diagram shows the decomposition of a high-level action within an existing plan.
- Where The BuildHouse action is replaced by the decomposition from the above example.
- The external precondition land is supplied by the existing causal link from BuyLand.
- The external precondition Money remains open after the decomposition step, so we add a new action, GetLoan.
- To be more precise follow the below steps,
    - First the action a' is removed from P.Then for each step S in the decomposition d'
    - Second step is to hook up the ordering constraints for a' in the original plan to the steps in d'.
    - Third and final step is to hook up casual links.



- This completes the additions required for generating decompositions in the context of the POP Planner.

3.4.3 Planning and Acting in Non-deterministic domains:

- So far we have considered only classical planning domains that are fully observable, static and deterministic.
- Furthermore we have assumed that the action descriptions are correct and complete.
- Agents have to deal with both incomplete and incorrect information.
- Incompleteness arises because the world is partially observable, non-deterministic or both.
- Incorrectness arises because the world does not necessarily match my model of the world.
- The possibility of having complete or correct knowledge depends on how much indeterminacy there in the world.
- **Bounded indeterminacy** actions can have unpredictable effects, but the possible effects can be listed in the action description axioms.
- **Unbounded indeterminacy** the set of possible preconditions or effects either is unknown or is too large to be enumerated completely.
- **Unbounded indeterminacy** is closely related to the **qualification problem.**
- There are four planning methods for handling indeterminacy.
- The following planning methods are suitable for bounded indeterminacy,
    - **Sensorleses Planning:-**
        - Also called as **Confront Planning**
        - This method constructs standard, sequential plans that are to be executed without perception.
        - This algorithm must ensure that the plan achieves the goal in all possible circumstances, regardless of the true initial state and the actual action outcomes.
        - It relies on **coercion** – the idea that the world can be forced into a given state even when the agent has only partial information about the current state.
        - Coercion is not always possible.
    - **Conditional Planning:-**
        - Also called as **Contingency planning**
        - This method constructing a conditional plan with different branches for the different contingencies that could arise.
        - The agent plans first and then executes the plan was produced.
        - The agents find out which part of the plan to execute by including **sensing actions** in the plan to test for the appropriate conditions.
- The following planning methods are suitable for Unbounded indeterminacy,
    - **Execution Monitoring and Replanning:-**
        - In this, the agent can use any of the preceding planning techniques to construct a plan.
        - It also uses **Execution Monitoring** to judge whether the plan has a provision for the actual current situation or need to be revised.
        - **Replanning** occurs when something goes wrong.
        - In this the agent can handle unbounded indeterminacy.
    - **Continuous Planning:-**
        - It is designed to persist over a lifetime.
        - It can handle unexpected circumstances in the environment, even if these occur while the agent is in the middle of constructing a plan.

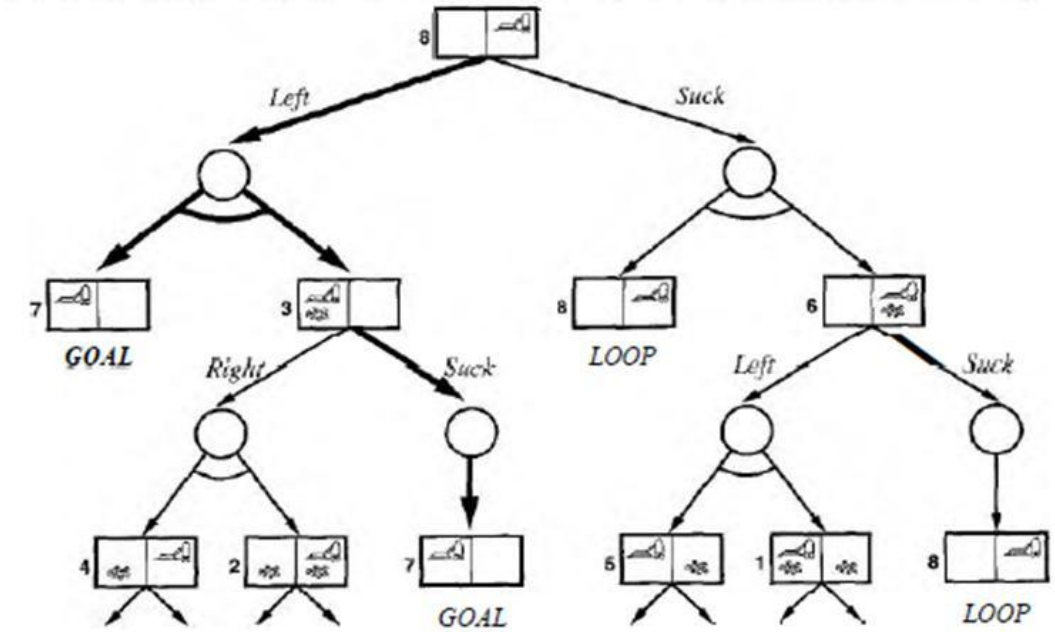
- It can also handle the abandonment of goals and the creation of additional goals by **goal formulation.**

3.4.4 Conditional Planning:-

- 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
- The partially observable case is more difficult to explain in this conditional planning.

3.4.4.1 Conditional planning in fully observable environments:

- Full observability means that the agent always knows the current state.
- CP in fully observable environments (FOE)
  - initial state : the robot in the right square of a clean world;
  - the environment is fully observable: *AtR ∧CleanL∧CleanR.*
  - The goal state : the robot in the left square of a clean world.
    - Vacuum world with actions *Left, Right,* and *Suck*
    - Disjunctive effects: Action (Left, PRECOND : AtR, EFFECT : AtL $\wedge \neg$ AtR)
    - Modified Disjunctive effects : Action (Left, PRECOND : AtR, EFFECT : AtL v AtR)
    - Conditional effects: Action(Suck, Precond: , Effect: (when AtL: CleanL) ^ (when AtR: CleanR)
      Action (Left, Precond: AtR, Effect: AtL v (AtL ^ when CleanL: !ClearnL)
  - Conditional steps for creating conditional plans:
    if test then planA else planB
    e.g., if AtL ^ CleanL then Right else Suck
  - The search tree for the vacuum world is shown in the following figure



  - The first two levels of the search tree for the double Murphy vaccum world.
  - State nodes are OR nodes where some action must be chosen.

SVCET

- Chance nodes, shown as circles, are AND nodes where every outcome must be handled, as indicated by the arc linking the outgoing branches.
- The solution is shown as **bold lines** in the tree.

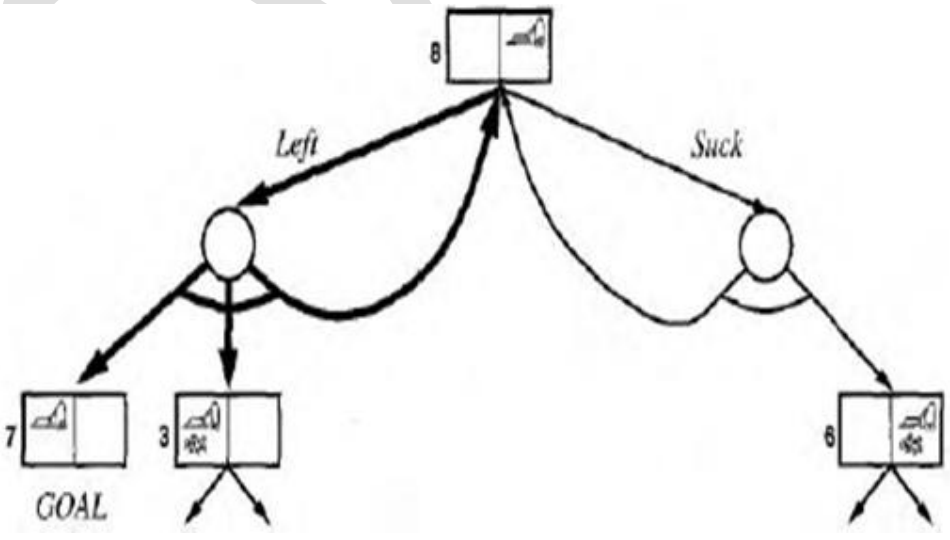
- The following table shows the recursive depth first algorithm for AND-OR graph search.

```
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_i = 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]
```

- The following figure shows the part of the search graph,
- clearly there are no longer any acyclic solutions, and AND-OR-GRAPH-SEARCH would return with failure, there is however a, cyclic solution, which is keep trying Left until it works.



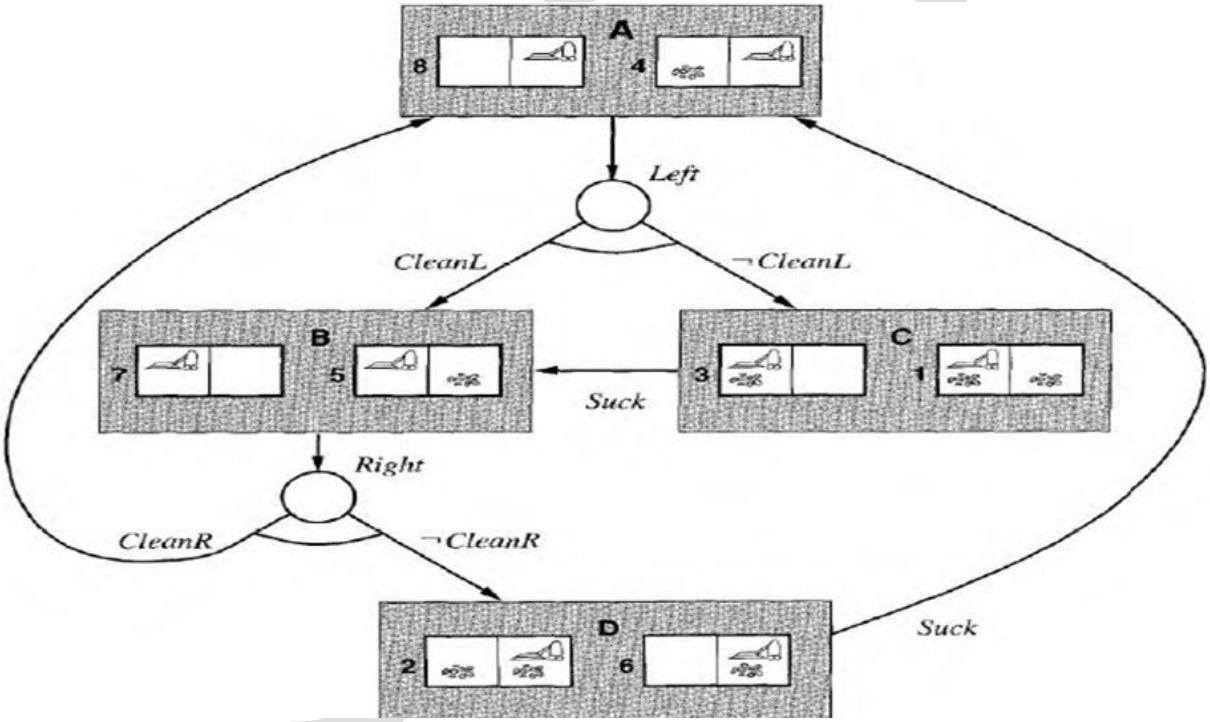
- The first level of the search graph for the triple Murphy vacuum world, where we have shown cycles explicitly.
- All solutions for this problem are cyclic plans.

- The cyclic solution is as follows,

$$[L_1 : \text{Left}, \textbf{if } AtR \textbf{ then } L_1 \textbf{ else if } CleanL \textbf{ then } [] \textbf{ else } Suck]$$

Conditional Planning in partially observable environments

- 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 agents initial belief state.
- Determine "both squares are clean" with local dirt sensing
  - the vacuum agent is AtR and knows about R, how about L?
- The following graph shows part of the AND-OR graph for the alternate double Murphy vaccum world,
- In which Dirt can sometimes be left behind when the agent leaves a clean square



- The agent cannot sense dirt in other squares.
- Sets of full state descriptions
  - { *(AtR ∧ CleanR ∧ CleanL), (AtR ∧ CleianR ∧ ¬CleanL)* }
- Logical sentences that capture exactly the set of possible worlds in the belief state.
  - *AtR ∧ CleanR*
- **Knowledge propositions** describing the agent's knowledge

$$K(AtR) \wedge K(CleanR)$$

- **closed-world assumption** - if a knowledge proposition does not appear in the list, it is assumed false.
- Now we need to decide how sensing works.

SVCET

- There are two choices here,
  - **Automatic sensing:-** Which means that at every time step the agent gets all the variable percepts
  - **Active sensing:-** Which means the percepts are obtained only by executing specific **sensory actions** such as
    - **CheckDirt**
    - **CheckLocation**

Action(Left, PRECOND: AtR,
EFFECT: K(AtL) ⋀¬K (AtR) ⋀ when CleanR: ¬K(CleanR) ⋀
when *CleanL: K (CleanL) ⋀*
when ¬ *CleanL: K(¬ CleanL)) .*

Action(CheckDirt, EFFECT:
when AtL⋀CleanL: K(CleanL) ⋀
when AtL ⋀ ¬CleanL: K (¬CleanL) ⋀
when AtR ⋀ CleanR: K(CleanR) ⋀
when AtR ⋀ ¬CleanR: K(¬CleanR))

3.4.4.2 Execution Monitoring and Replanning:

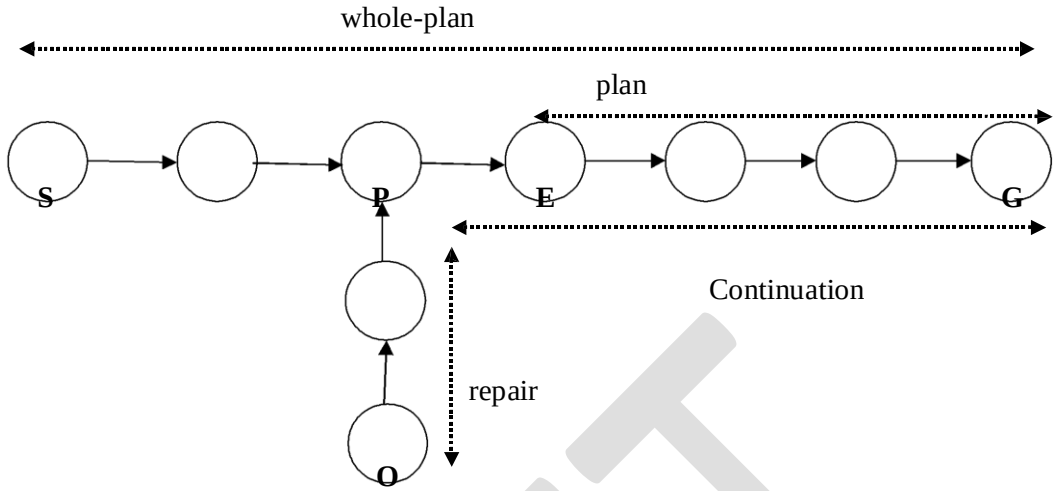
- An execution monitoring agent checks its percepts to see whether everything is going to according plan.
- Murphy's law tells us that even the best-laid plans of mice, men and conditional planning agents frequently fail.
- The problem is unbounded indeterminacy – some unanticipated circumstances will always arise for which the agents action description are incorrect.
- Therefore, execution monitoring is a necessity in realistic environments.
- we will consider two kinds of execution monitoring,
  - Simple, but weak form called action monitoring – whereby the agent checks the environment to verify that the next action will work.
  - more complex, but more effective form called plan monitoring – in which the agent verifies the entire remaining plan.
- A **replanning** agent knows what to do when something unexpected happens, call a planner again to come up with a new plan to reach the goal.
- To avoid spending too much time planning, this is usually done by trying to repair the old plan – to find a way from the current unexpected state back onto the plan
- Together **Execution Monitoring and replanning** form a general strategy that can be applied to both fully and partially observable environments
- It can be applied to a variety of planning representations as state-space, partial-order and conditional plans.
- The following table shows a simple approach to state-space planning.
- The planning agent starts with a goal and creates an initial plan to achieve it.
- The agent then starts executing actions one by one.
- The replanning agent keeps track of both the remaining unexpected plan segment plan and the complete original plan whole-plan
- It uses **action monitoring:** before carrying out the next action of plan, the agent examines its percepts to see whether any preconditions of the plan have unexpectedly become unsatisfied.

- If they have, the agent will try to get back on track by replanning a sequence of actions that should take it back to some point in the whole-plan.
- The following table **has an agent that does action monitoring and replanning**
- It uses a complete state-space planning algorithm called PLANNER as a subroutine.
- If the preconditions of the next action are not met, the agent loops through the possible point p in whole-plan, trying to find one that PLANNER can plan a path to.
- This path is called repair.
- If PLANNER succeeds in finding a repair, the agent appends repair and the tail of the plan after p, to create the new plan.
- The agent then returns the first step in the plan.

```
Function REPLANNING-AGENT(percept) returns an action
    Static: KB, a Knowledge base (includes action descriptions)
            Plan, a plan, initially []
            Whole-plan, a plan, initially []
            Goal, a goal

    TELL(KB,MAKE-PERCEPT-SENTENCE(percept,t))
    Current ← STATE-DESCRIPTION(KB,t)
    If plan = [] then
          whole-plan ← plan ← PLANNER(current,goal,KB)
    If PRECONDITIONS(FIRST(plan)) not currently true in KB then
          Candidates ← SORT(whole-plan, ordered by distance to current)
          Find state s in candidates such that
                Failure repair ← PLANNER(current,s,KB)
          Continuation ← the tail of whole-plan starting at s
          Whole-plan ← plan ← APPEND(repair, continuation)
    Return POP(plan)
```

- The following diagram shows the schematic illustration of the process.
- The illustration of process is also called as Plan Monitoring.
- The replanner notices that the preconditions of the first action in plan are not satisfied by the current state.
- It then calls the planner to come up with a new subplan called repair that will get from the current situation to some state s on whole-plan.



- Before execution, the planner comes up with a plan, here called whole-plan, to get from **S** to **G**.
- The agent executes the plan until the point Marked **E.**
- Before executing the remaining plan, it checks preconditions as usual and finds that it is actually in state **O** rather than state **E.**
- It then calls its planning algorithm to come up with repair, which is a plan to get from **O** to some point **P** on the original whole-plan.
- The new plan now becomes the concatenation of repair and continuation.
- For example:-
    - Problem of achieving a chair and table of matching color

$$
\begin{aligned}
&Init(Color(Chair, Blue) \wedge Color(Table, Green) \\
&\quad \wedge ContainsColor(BC, Blue) \wedge PaintCan(BC)) \\
&\quad \wedge ContainsColor(RC, Red) \wedge PaintCan(RC) \\
&Goal(Color(Chair, x) \wedge Color(Table, x)) \\
&Action(Paint(object, color), \\
&\quad \text{PRECOND:} HavePaint(color) \\
&\quad \text{EFFECT:} Color(object, color)) \\
&Action(Open(can), \\
&\quad \text{PRECOND:} PaintCan(can) \wedge ContainsColor(can, color) \\
&\quad \text{EFFECT:} HavePaint(color)
\end{aligned}
$$

- The agents PLANNER should come up with the following plan as,

$$[Start, Open(BC); Paint(Table, Blue); Finish]$$

SVCET

- If: the agent constructs a plan to solve the painting problem by painting the chair and table red. only enough paint for the chair
- Plan monitoring
  - Detect failure by checking the *preconditions for success* of the *entire remaining* plan
  - Useful when a goal is serendipitously achieved
    - While you're painting the chair, someone comes painting the table with the same color
  - Cut off execution of a doomed plan and don't continue until the failure actually occurs
    - While you're painting the chair, someone comes painting the table with a different color
- If one insists on checking every precondition, it might never get around to actually doing anything
- RP - monitors during execution

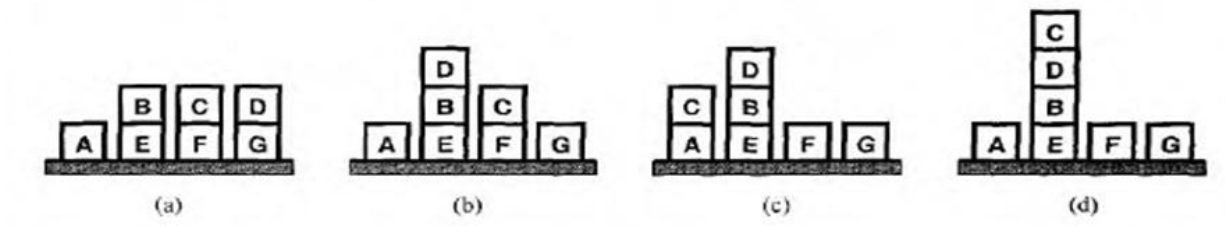
3.4.4.3 Continuous Planning

- Continuous planning agent
  - execute some steps ready to be executed
  - refine the plan to resolve standard deficiencies
  - refine the plan with additional information
  - fix the plan according to unexpected changes
    - recover from execution errors
    - remove steps that have been made redundant
- Goal ->Partial Plan->Some actions-> Monitoring the world -> New Goal
- The continuous planning agent monitors the world continuously, updating its world model from new percepts even if its deliberations are still continuing.
- For example:-
  - use the blocks world domain problem
  - The action we will need is Move(x, y), which moves block x onto block y, provided that both are clear.
  - The following is the action schema,

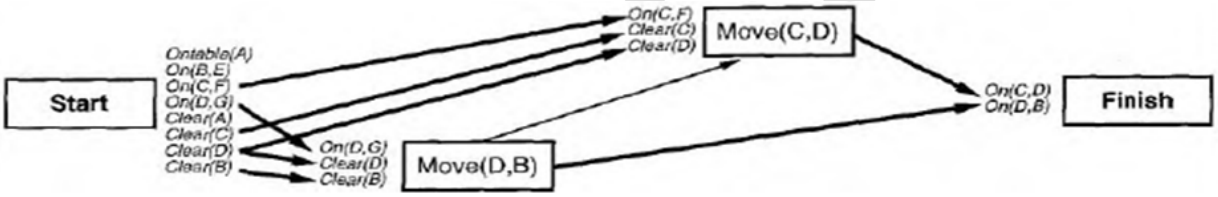
    Action (*Move(x, y),*

    PRECOND: Clear(x) $\wedge$ *Clear(y)* $\wedge$ *On(x ,z),*

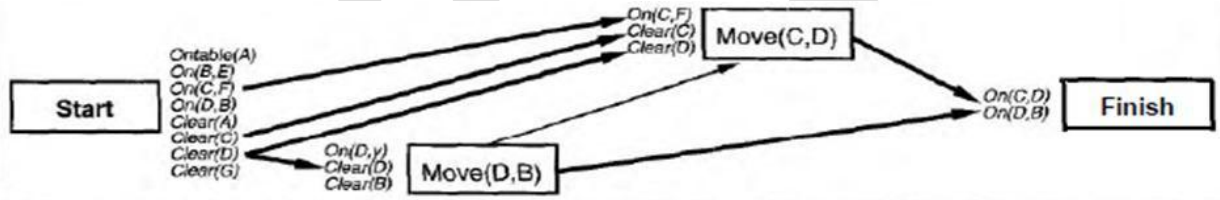
    EFFECT: On(x, y $\wedge$ *Clear(z)* $\wedge \neg$ *Clear(y)* $\wedge \neg$ *On(x, z))*
  - Goal: *On(C, D)* $\wedge$ *On(D ,B)*
  - *Start* is used as the label for the current state
  - The following seven diagram shows the continuous planning agent approach towards the goal
  - Plan and execution
    - Steps in execution:
      - Ordering - *Move(D,B),* then *Move(C,D)*
      - Another agent did *Move(D,B) - change the plan*
      - Remove the redundant step
      - Make a mistake, so *On(C,A)*
        - Still one open condition
      - *Planning one more time - Move(C,D)*
      - Final state: start -> finish



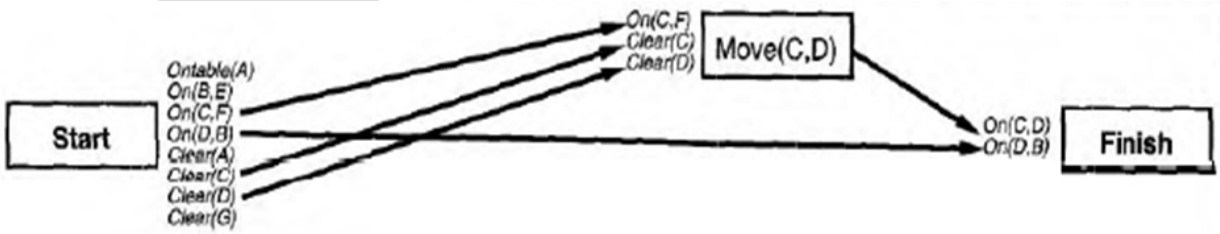
- The sequences of states as the continuous planning agent tries to reach the goal state *On(C, D) ∧ On(D ,B) as shown in (d).*
- The start state is (a).
- At (b), another agent has interfered, putting D on B.
- At (c), the agent has executed Move(C, D) but has failed, dropping C on A instead.
- It retries Move(C, D), reaching the goal state (d).



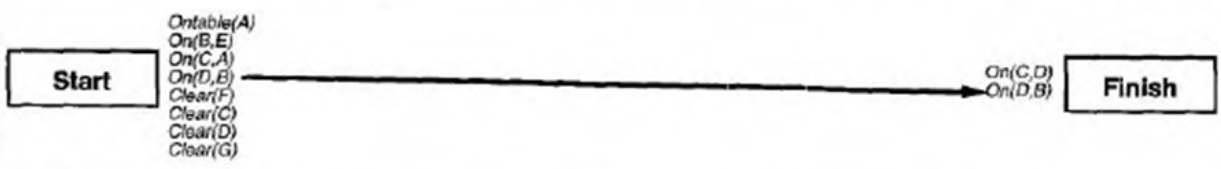
- The initial plan constructed by the continuous planning agent.
- The plan is indistinguishable, so far, from that produced by a normal POP.



- After someone else moves D onto B, the unsupported links supplying Clear(B) and On(D, G) are dropped, producing this plan.

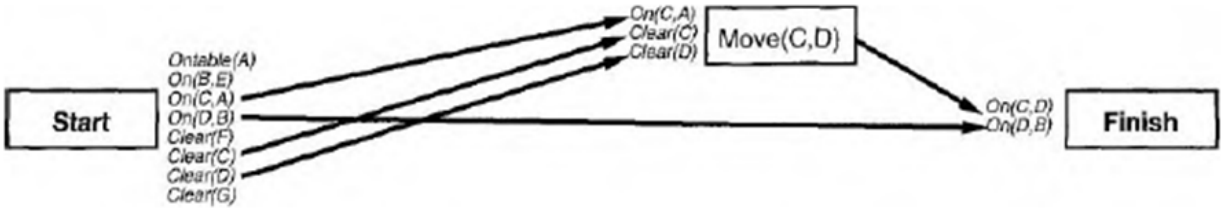


- The link supplied by Move(D, B) has been replaced by one from Start, and the new-redundant step Move(D, B) has been dropped.

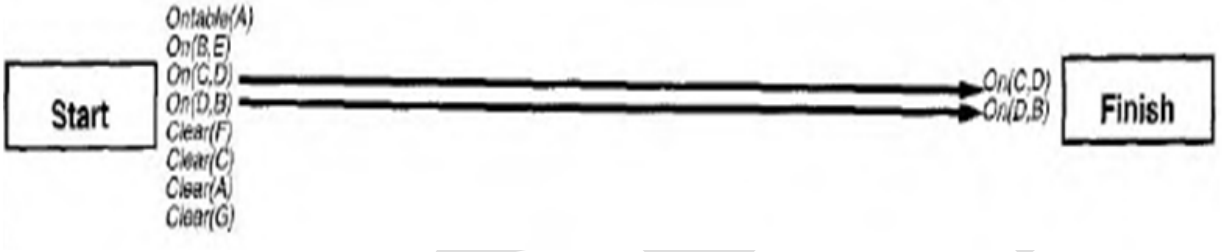


SVCET

- After Move(C, D) is executed and removed from the plan, the effects of the Start step reflect the fact that C ended up on A instead of the intended D.
- The goal precondition On(C, D) is still open.



- The open condition is resolved by adding Move(C, D) back in.



- After Move(C, D) is executed and dropped from the plan, the remaining open condition On(C, D) is resolved by adding a causal link from the new start step.
- Now the plan is completed.
- From this example, we can see that continuous planning is quite similar to POP.
- On each iteration, the algorithm finds something about the plan that needs fixing a so-called **plan-flaw** and fixes it.
- The POP algorithm can be seen as a flaw-removal algorithm where the two flaws are open preconditions and causal conflicts.
- On the other hand, the continuous planning agent addresses a much broader range of flaws as follows,
  - Missing goals
  - Open precondition
  - Causal conflicts
  - Unsupported links
  - Redundant actions
  - Unexecuted actions
  - Unnecessary historical goal
- The following table shows the continuous-POP-Agent algorithm

Function CONTINUOUS-POP-AGENT (percept) **returns** an action
Static: plan, a plan, initially with just Start, Finish

Action ← NoOp (the default)
EFFECTS [Start] = UPDATE(EFFECTS [Start], percept)
REMOVE-FLAW (plan) // possibly updating action
Return action

- It has a cycle of "perceive, remove flaw act"
- It keeps a persistent plan in its KB, and on each turn it removes one ,flaw from the plan.
- It then takes an action and repeats the loop.

- It is a continuous partial-order planning agent.
- After receiving a percept the agent removes flaw from its constantly updated plan and then returns an action.
- Often it will take many steps of flaw-removal planning, during which it returns NoOp, before it is ready to take a real action.

3.4.4.4 Multiagent Planning

- So far we have dealt with **single-agent environments**
- Multiagent environments can be **cooperative** or **competitive.**
- For example:-
  - the problem is team planning in double tennis.
- Plans can be constructed that specify actions for both players on the team
- Our objective is to construct plans efficiently.
- To do this we need requires some form of **coordination,** possibly achieved by **communication.**
- The following table shows the double tennis problem,

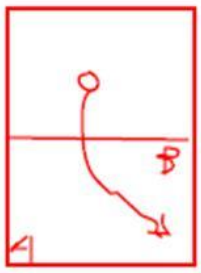
Agents(A, B) ← declares that there are two agents

Init(At(A, [Left, Baseline]) ∧ At(B, [Right, Net]) ∧
Approaching(Ball, [Right, Baseline])) ∧ Partner(A, B) ∧ Partner(B, A)

Goal(Returned(Ball) ∧ At(agent, [x, Net]))

Action(Hit(agent, Ball),
PRECOND: Approaching(Ball, [x,y]) ∧ At(agent, [x,y]) ∧
Partner(agent, partner) ∧ ¬ At(partner, [x,y])
EFFECT: Returned(Ball))

Action(Go(agent, [x,y]),
PRECOND: At(agent, [a,b]),
EFFECT: At(agent, [x,y]) ∧ ¬ At(agent, [a,b]))



- In the above table, Two agents are playing together and can be in one of four locations as follows,
  - [Left, Baseline]
  - [Right, Baseline]
  - [Left, Net]
  - [Right, Net]
- The ball can be returned if exactly one player is in the right place.

Cooperation: Joint goals and plans

- **An agent (A, B) declares** that there are two agents, A and B who are participating in the plan.
- Each action explicitly mentions the agent as a parameter, because we need to keep track of which agent does what.
- A solution to a multiagent planning problem is a **joint plan** consisting of actions for each agent

- A joint plan is a solution if the goal will be achieved when each agent performs its assigned actions.
- The following plan is a solution to the tennis problem
  - PLAN 1 :
    - *A* : *[Go(A,[Right, Baseline]),Hit(A, Ball)]*
    - *B* : *[NoOp(B),NoOp(B)]*.
- If both agents have the same KB, and if this is the only solution, then everything would be fine; the agents could each determine the solution and then jointly execute it.
- Unfortunately for the agents, there is another plan that satisfies the goal just as well as the first
  - PLAN 2:
    - *A* : *[Go(A, [Left, Net]), NoOp(A)]*
    - *B* : *[Go (B,[Right,baseline]),H it(23, Ball)]*
- If *A* chooses plan *2* and B chooses plan 1, then nobody will return the ball.
- Conversely, if A chooses 1 and B chooses 2, then they will probably collide with each other; no one returns the ball and the net may remain uncovered.
- So the agents need a mechanism for **coordination** to reach the same joint plan

Multibody Planning:

- concentrates on the construction of correct joint plans, deferring the coordination issue for the time being, we call this **Multibody planning**
- Our approach to multibody planning will be based on partial-order planning
- we will assume full observability, to keep things simple
- There is one additional issue that doesn't arise in the single-agent case; the environment is no longer truly **static.**
- Because other agents could act while any particular agent is deliberating.
- Therefore we n**eed synchronization**
- We will assume that each action takes the same amount of time and that actions at each point in the joint plan are simultaneous.
- At any point in time, each agent is executing exactly one action.
- This set of concurrent actions is called a **joint action.**
- For example, Plan 2 for the tennis problem can be represented as this sequence of joint actlons:

$$\langle Go(A, [Left,Net])Go(B, [Right,baseline])\rangle$$
$$\langle NoOp(A), Hit(B, Ball)\rangle$$

Coordination Mechanisms:

- The simplest method by which a group of agents can ensure agreement on a joint plan is to adopt a **convention** prior to engaging in joint activity.
- A convention is any constraint on the selection of joint plans, beyond the basic constraint that the joint plan must work if all agents adopt it
- For example
  - the convention "stick to your side of the court" would cause the doubles partners to select plan 2

- the convention "one player always stays at the net" would lead them to plan 1
- In the absence of an applicable convention, agents can use communication to achieve common knowledge of a feasible join plan
- For example:
  - a doubles tennis player could shout "Mine!" or "Yours!" to indicate a preferred joint plan.

Competition:

- Not all multiagent environments involve cooperative agents
- Agents with conflicting utility functions are in **competition** with each other
- One example: chess-playing. So an agent must
  (a) recognize that there are other agents
  (b) compute some of the other agent's possible plans
  (c) compute how the other agent's plans interact with its own plans
  (d) decide on the best action in view of these interactions

****

THIRD UNIT-I PLANNING FINISHED

GOOD LUCK

****

ARTIFICIAL INTELLIGENCE

UNIT-IV

PLANNING AND MACHINE LEARNING

Basic plan generation systems – Strips - Advanced plan generation systems - K strips -Strategic explanations - Why, Why not and how explanations. Learning - Machine learning, adaptive learning.

4.1 Uncertainty

- Agents almost never have access to the whole truth about the environment
  (i.e)Agent must therefore act under uncertainity.
- Uncertainity can also arise because of incompleteness and incorrectness in the agent's understanding of the properties of the environment.

4.1.1 Handling of Uncertainty:-

- Identifying uncertainity in dental diagnosis system.
- For all P Symptom(P,toothache) → Diagnosis(P,Cavity)
- This rule is logically wrong.Not all patients with toothache have cavities,some of them may have gum disease or impacted wisdom teeth or one of several other problems.
- For all P symptom(P,toothache) → Disease(P,cavity) v Disease(P-Gumdisease) v Disease(P,Impacted Wisdom)….
  (i.e)unlimited set of possibilities are exists for toothache symptom.
  Change into casual rule as:
  - For all P disease(P,cavity) → Symptom(P,toothache),but this rule is not right either,not all cavities cause pain.
  - Trying to FOL in medical diagnosis thus fails for three main reasons.
    I. **LAZINES:** Too much work to list the complete set of antecedents and consequents needed.
    II. **THEORETICAL IGNORANCE:** Medical science has no complete theory for domain.
    III. **PRACTICAL IGNORANCE:** Even if we know all the rules,uncertainit y arises because some tests cannot be run on the patients body.

- CONCLUSION:
  - Agents knowledge can at best provide only a degree of belief in the relevant sentences.the total used to deal with degree of belief will be probability theory,which assigns or numerical degree of belief between 0 to 1 to sentences.

- PRIOR (or) UNCONDITIONAL PROBABILITY:Before the evidence is obtained.
- POSTERIOR (or) CONDITIONAL PROBABILITY:After the evidence is obtained.
- UTILITY THEORY:To represent and reasons with preference(i.e)utility-quality of being useful.

Decision theory=probability theory + Utility theory

- The fundamentals idea of decision theory is that an agent is rational if and only if it chooses the action that yields the highest expected utility,averaged overall the possible outcomes of the action-maximium expected utility.(i.e)Weighting the utility of a particular outcome by the probability that it occurs.
- The following shows a decision theoretic agent

Function DT-Agent (percept)returns an action
Static: belief_state,probabilistic beliefs about the current state of world
action, the Agent's action
Update: belief_state based on action and percept
Calculate outcomes probabilities for actions,
given action description and current Belief_state
Select action with highest expected utility
given probabilities of outcomes and Utility information
Return action

4.2 Review of Probability

AXIOMS OF PROBABILITY:

I. All probabilities are between 0 and 1. $0 \leq P(A) \leq 1$
II. Necessarily true (i.e. valid) proposition have probability 1 an necessarily false (i.e. unsatisfiable)proposition have probability 0 P(True) = 1 P(False) = 0
III. The probability of a disjunction is given by $P(A \vee B) = P(A) + P(B)-P(A \wedge B)$
IV. Let $B = \neg A$ in the axiom (III)
V. $P(True) = P(A) + P(\neg A) – P(False)$ (by logical equivalence)
VI. $1 = P(A) + P(\neg A)$ (by step 2)
VII. $P(\neg A) = 1-P(A)$ (by algebra)

- Joint probability distribution:

An agent's probability assignments to all propositions in the domain (both simple and complex)

Ex: Trivial medical domain with two Boolean variables.

	Toothache	¬Toothache
Cavity	0.04	0.06
¬Cavity	0.01	0.89

I. Adding across a row or column gives the unconditional probability of a variable.
P(Cavity) = 0.06 + 0.04 = 0.1

P(Cavity + Toothache) = 0.04 +0.01 + 0.06 = 0.11

II. Conditional Probability

$$P(Cavity / Toothache) = \frac{P(Cavity\ Toothache)}{P(Toothache)}$$

$$= \frac{0.04}{0.04+0.01} = 0.80$$

Bayes'Rule:

1. Recall two forms of the product rule

   $P(A \wedge B) = P(A/B)\ P(B)$

   $P(A \wedge B) = P(B/A)\ P(A)$

   Equating the two righthand sides and dividing by P(A),i.e.

   $$P(A/B) = \frac{P\left(\frac{A}{B}\right)P(B)}{P(A)}$$

   Is called as Baye's rule (or) Baye's law (or) Baye's theorem

2. From the above equation the general law of multivalued variables can be written using the P notation:

   $$P(Y / X) = \frac{P\left(\frac{X}{Y}\right)P(Y)}{P(X)}$$

3. From the above equation on some background evidence E:

   $$P(Y / X,E) = \frac{P\left(\frac{X}{Y},E\right)P\left(\frac{Y}{E}\right)}{P\left(\frac{X}{E}\right)}$$

4. Disadvantage

   It requires three terms to compute one conditional probability (P(B/A))

   - One conditional probability P(A/B)
   - Two unconditional probability P(B) and P(A)

5. Advantage

   If three values are known,then the unknown fourth value → P(B/A) is computed easily.

6. Example:

   Given: P(S/M) = 0.5 , P(M) = 1/5000 , P(S) = 1/20

   S – the proposition that the patient has a stiff nect

   M – the proposition that the patient has meningitis

   P(S/M) – only one in 5000 patients with a stiff neck to have meningitis

   $$P(M/S) = \frac{0.5*\left(\frac{1}{50000}\right)}{1/20} = 0.0002$$

7. Normalization

   a) Consider again the equation for calculating the probability of meningitis given a stiff neck.

   $$P(M/S) = \frac{P\left(\frac{S}{M}\right)P(M)}{P(S)}$$

   b) Consider the patient is suffering from whiplash W given a stiff neck.

   $$P(W/S) = \frac{P\left(\frac{S}{W}\right)P(W)}{P(S)}$$

c) To perform relative likelihood between a and b,we need P(S/W) = 0.8 and P(W) = 1/1000 and P(S) is nit required since it is already defined

$$\frac{P(\frac{M}{S})}{P(\frac{W}{S})} = \frac{P(\frac{S}{M})P(M))}{P(\frac{S}{W})P(W} = \frac{0.5*1/50000}{0.8*1/1000} = \frac{1}{80}$$

i.e.whiplash is 80 times more likely than meningitis,given a stiff

neck.

d) Disadvantages: consider the folloeing equations:

P(M/S) = $\frac{P(\frac{S}{M})P(M)}{P(S)}$ ……. (1)

P(⌐M/S) = $\frac{P(\frac{S}{M})P(M)}{P(S)}$ …….. (2)

Adding (1) and (2) using the fact that
P(M/S) + P(⌐M/S) = 1,we obtain
P(S) = P(S/M) P(M) + P(S/⌐M) P(⌐M)
Substituting into the equation for P(M/S),we have

P(M/S) = $\frac{P(\frac{S}{M})P(M)}{P(\frac{S}{M})P(M)+P(\frac{S}{M})P(M)}$

This process is called normalization ,because it treats 1/P(S) as a normalizing constant that allows the conditional terms to sum to 1

The general multivalued normalization equation is

P($\frac{Y}{X}$) = αP($\frac{X}{Y}$)P(Y)

α – normalization constant

8. Baye's Rule and evidence

a) Two conditional probability relating to cavities:

P(Cavity / Toothache) = 0.8

P(Cavity /Catch) =0.95

Using Baye's Rule:

P(Cavity/Toothache ∧ Catch) = $\frac{P(Toothache\ \frac{Catch}{Cavity})P(Cavity)}{P(Toothache\ Catch)}$

b) Bayesian updating is done (i.e) evidence one piece at a time.

P(Cavity/Toothache) = P(Cavity) $\frac{P(\frac{Toothache}{Cavity})}{P(Toothache)}$ ……..(1)

c) When catch is observed apply Bayes Rule with constant conditioning context

P(Cavity/Toothache ∧ Catch) = $\frac{P(\frac{Catch}{Toothache}Cavity)}{P(\frac{Catch}{Toothache})}$ ……(2)

From (1) and (2)

= P(Cavity) $\frac{P(\frac{Toothache}{Cavity})}{P(Toothache)}\frac{P(\frac{Catch}{Toothache}cavity)}{P(\frac{Catch}{Toothache})}$

d) Mathematically the equation are rewritten as:

P(Catch/Cavity ∧ Toothache) = P(Catch/Cavity)
P(Toothache/Cavity ∧ Catch) = P(Toothache/cavity)
These equations express the conditional independence of Toothache and catch on given Cavity.

e) Using conditional independences,simplify the equation of Bayes updating

P(Cavity/Toothache ∧ Catch) = P(Cavity) $\frac{P\left(\frac{Toothache}{Cavity}\right)P\left(\frac{Catch}{Cavity}\right)}{P(Toothache)P\left(\frac{Catch}{Toothache}\right)}$

f) Using normalization,it is further reduced as
P(Cavity/Toothache ∧ Catch) → P(X/Y,Z) = P(X/Z)
P(Z/X,Y) = α P(Z) P(X/Z) P(Y/Z) (i.e.) P(Z/X,Y)sum to 1

4.3 Bayesian Network:-

4.3.1 Syntax:

- A data structure used to represent knowledge in an uncertain domain (i.e) to represent the dependence between variables and to give a whole specification of the joint probability distribution.
- A belief network is a graph in which the following holds.
  - I. A set of random variables makes up the nodes of the network.
  - II. A set of directed links or arrows connects pairs of nodes x→y,x has a direct influence on y.
  - III. Each node has a conditional probability tale that quantifies the effects that the parents have on the node.The parents of a node are all nodes that have arrows pointing to it.
  - IV. Graph has no directed cycles(DAG)
- The other names of Belief network are Bayesian network ,probabilistic network, casual network and knowledge map.

- Example:

A new burglar alarm has been installed at home.

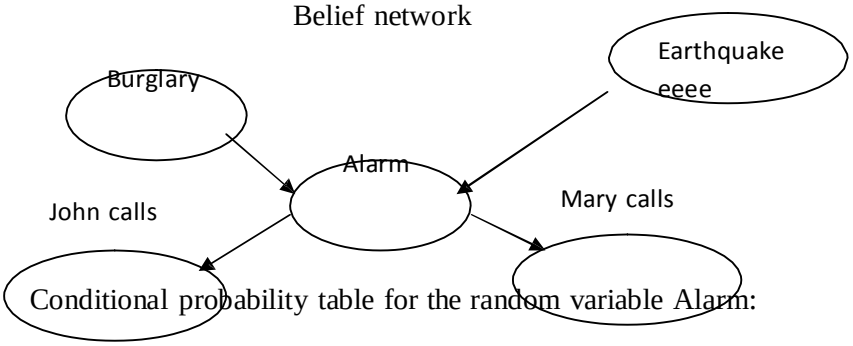
- It is fairly reliable at detecting a burglary but also responds on occasion to minor earthquakes.
- You also have two neighbours,John and Mary,who have promised to call you at work when they hear the alarm.
- John always calls when he hears the alarm but sometimes confuses the telephone ringing with the alarm and calls then too.
- Mary on the otherhand likes rather loud music and sometimes misses the alarm together.
- Given the evidence of who has or has not called estimate the probability of a burglary

Uncertainty:
I. Mary currently listening to loud music

II. John confuses telephone ring with alarm → laziness and ignorance in the operation

III. Alarm may fail off → power failure, dead battery, cut wires etc.

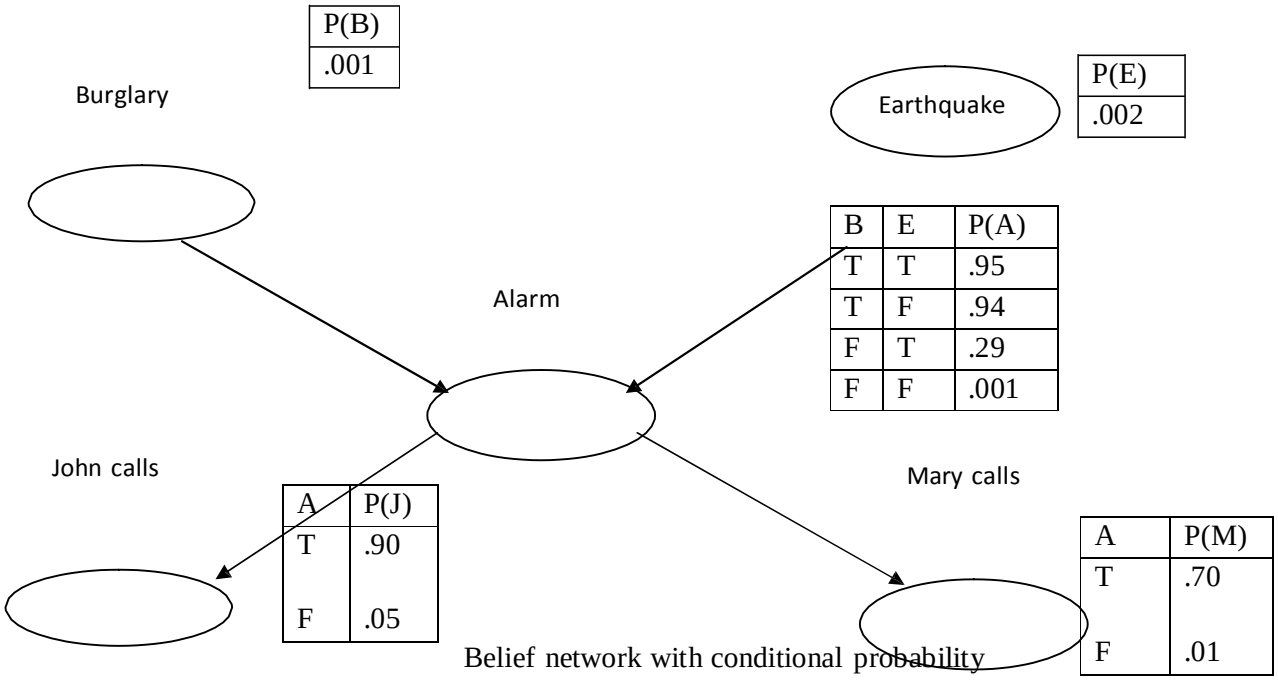
Belief network



Conditional probability table for the random variable Alarm:

Burglary	Earthquake	P(Alarm/Burglary,Earthquake)	
		True	False
T	T	0.950	0.050
T	F	0.950	0.050
F	T	0.290	0.710
F	F	0.001	0.999

Each row in a table must sum to 1,because the entry represents set of cases for the variable. A table with n Boolean variables contain 2^n independently specifiable probabilities.



Belief network with conditional probability

4.3.2 Semantics

There are two ways in which one can understand the semantics of Belief networks

1. Network as a representation of the joint probability distribution-used to know how to construct networks.

2. Encoding of a collection of conditional independence statements-designing inference procedure.

- Joint probability distribution: How to construct network's? A belief network provides a complete description of the domain.Every entry in the joint probability distribution can be calculated from the information in the network.A entry in the joint is the probability of a conjunction of particular assignment to each variable(i.e)

  $P(X_1 = x, \wedge \ldots \wedge x_n = x_n)$
- We use the notation $P(x_1 \ldots .x_n)$as an abbreviation for this.The value of this entry is given by the following formula:

  $P(x_1 \ldots\ldots x_n) = \prod_{i=1}^{n} P(x_i|\text{Parents}(X_i))$
- Thus each entry in the joint is represented by the product of the appreciate elements of the CPT in the belief network.The CPT's therefore provide a decomposed representation of the joint.
- The probability of the event that alarm has sounded but neither a burglary nor an earthquake has occurred,and both John and Mary call.We use single letter names for the variables.

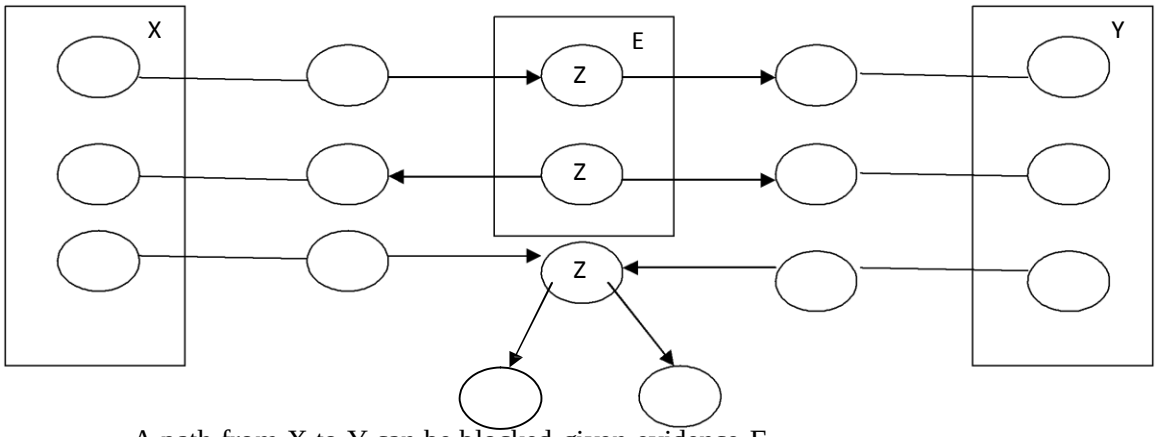
  $P(J \wedge M \wedge A \neg B \wedge \neg E)$
  $= P(J/A)\ P(M/A)\ P(A|\neg B \wedge \neg E)\ P(\neg B)\ P(\neg E)$
  $= 0.90 * 0.70 * 0.001 * 0.999 * 0.998$
  $= 0.00062$

  Noisy OR: It is the logical relationship of uncertainty.In proposition logic we might say fever is true, If and only if cold, flu or malaria is true. The Noisy OR made adds some uncertainity to this strict logical approach. The model makes three assumptions.
  I. It assumes the each cause has an independent chance of causing the effect.
  II. It assumes that all possible causes are listed.
  III. It assumes that whatever inhibits Flu from causing a fever.These inhibits are not responded as nodes but rather are summarized as "noise parameters"

  Example
  P(Fever/cold) = 0.4
  P(Fever/Flu)=0.8 Noise parameters are 0.6,0.2 and 0.1
  P(Fever/Malaria)=0.9
- Conclusion:

  I. If no parent node is true then the output is false with 100% certainity.
  II. If exactly one parent is true,then the output is false with probability equal to the noise parameter for that node.
  III. The probability that the output node is false is just the product of the noise parameters for all the input nodes that are true.

- Conditional independent relations in belief networks:
  - From the given network is it possible to read off whether a set of nodes X is independent of another set Y,given a set of evidence nodes E? the answer is yes,and the method is provided by the notion of direction dependent separation or de-seperation.
  - If every undirected path from a node in X to a node in Y is de-seperated by E then X and Y are conditionally independent given E.

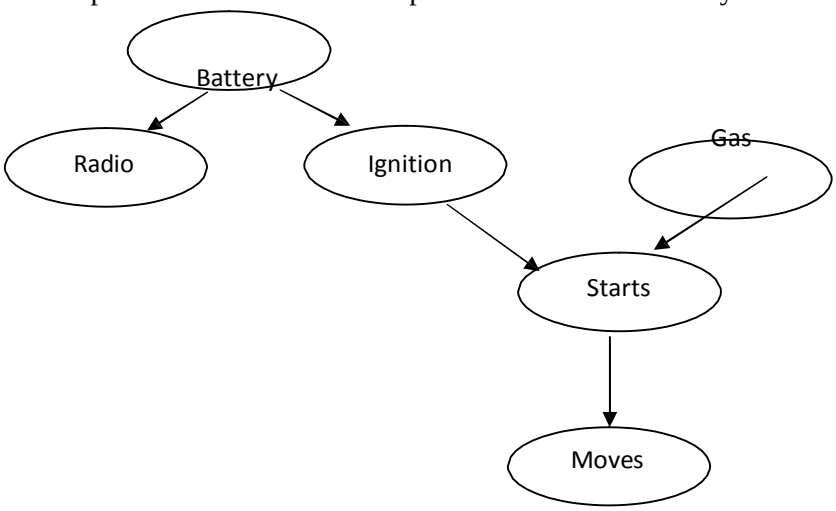


A path from X to Y can be blocked given evidence E

- Three paths in which a path from x to y can be blocked,given a evidence E.If every path from x to Y is blocked,then we say E deseperates x and y(i.e)

I. Z is in E and z has one arrow on the path leading in and one arrow out.
II. Z is in E and Z has both arrows leading out.
III. Neither Z nor any descendents of Z is in E and both arrows lead into Z.

Example belief network for d-seperation:Car's electrical system and engine



1. Whether there is a Gas in the car and whether the car Radio plays are independent given evidence about whether the Spark plugs fire
2. Gas and Radio are independent if battery works.
3. Gas and Radio are independent given no evidence at all.
4. Gas and Radio are dependent on evidence start.

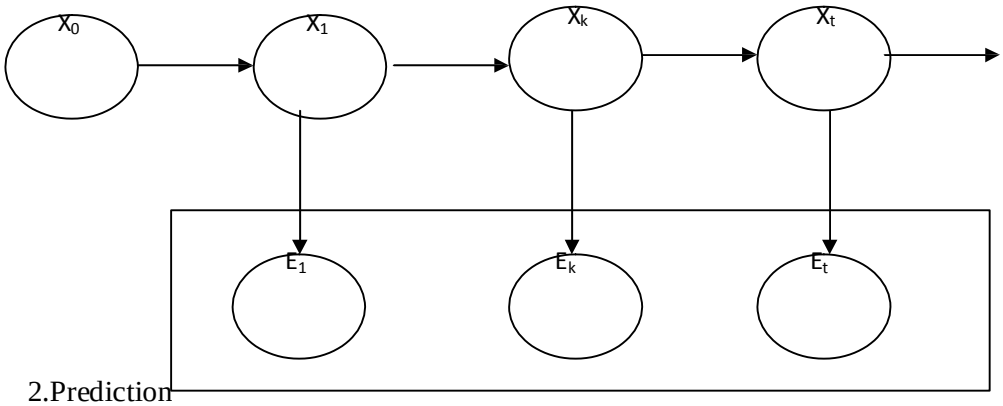
4.5. Inference in Temporal models

The generic temporal model has the following set of inference tasks:

1. Monitoring (or) filtering

Filtering (Monitoring):computing the conditional distribution over the current state, given all evidence to data, $P(X_t|e1:t)$

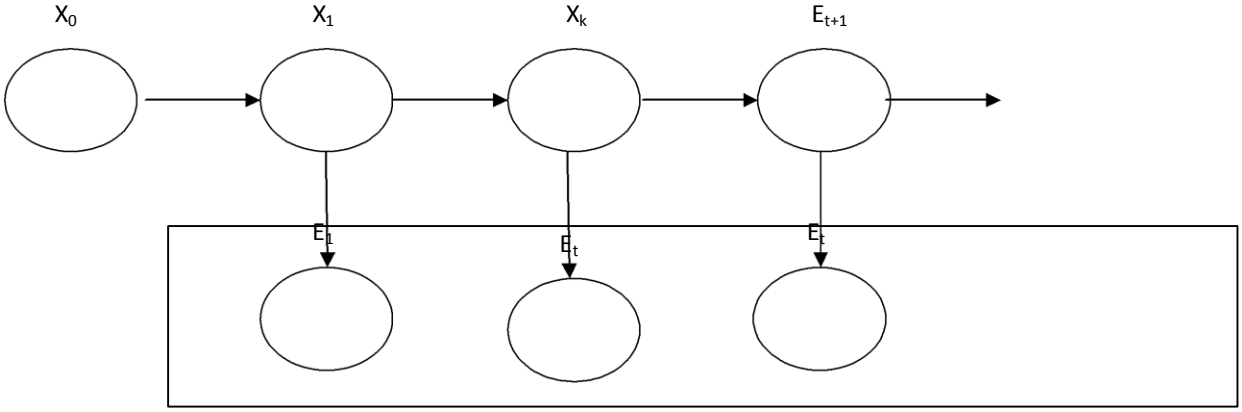
- In the umbrella example, monitoring would mean computing the probability of rain today, given all the observation of the umbrella so far, including today



2.Prediction

Prediction:computing the conditional distribution over the future state,given all evidence to date,$P(X_t+k|e1:t)$,for $k>0$.

In the umbrella example,prediction would mean computing the probability of rain tomarrow(k=1),or the day after tomarrow(k=2),etc.,given all the observations of the umbrella so far X_t+1



Monitoring(filtering)

- Filtering(monitoring):computing the conditional distribution over the current state,given all evidence to data,corresponds to computing the distribution $P(X_t|e_{1:t})$,or $P(X_{t+1}|e_{1:t+1})$:

$$P(X_{t+1}|e_{1:t+1}) = P(X_{t+1}|e_{1:t},e_{t+1}) = P(X_{t+1}|e_t+1,e_{1:t})$$

- General form of Baye's rule conditional also on evidence e

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

- In temporal Markov process,it reads:

$$P(X_{t+1}|e_{t+1},e_{1:t}) = \alpha P(e_{t+1}|X_{t+1},e_{1:t})\ P(X_{t+1}|e_{1:t})$$

- Since evidence e_t depends only on the current state X_t

$$P(X_{t+1}|e_{t+1},e_{1:t}) = \alpha P(e_{t+1}|X_{t+1},e_{1:t})\ P(X_{t+1}|e_{1:t})$$

- Then we can simplify

$$P(X_{t+1}|e_{1:t+1}) = \alpha P(e_{t+1}|X_{t+1})\ P(X_{t+1}|e_{1:t})$$

- The second term $P(X_{t+1}|e_{1:t})$,corresponds to a one-step prediction of the nextstate,given evidence up to time t,and the first term updates this new state with the new evidence at time t+1his updating is called filtering.
- Let us now obtain the one-step prediction:

$$P(X_{t+1}|e_{1:t}) = \sum_{Xt} P(X_{t+1}|X_t)\ P(X_t|e_{1:t})$$

- The first term is the (Markov) transition model and the second term is a current state distribution given evidence up to date

$$P(X_{t+1}|e_{1:t}) = \sum_{Xt} P(X_{t+1}|X_t)\ P(X_t|e_{1:t})$$

- The recursive formula for monitoring/filtering then reads

$$P(X_{t+1}|e_{1:t+1}) = \alpha P(e_{t+1}|X_{t+1}) \sum_{Xt} P(X_{t+1}|X_t)\ P(X_t|e_{1:t})$$

We can write the same set of equations for $P(X_t|e_{1:t})$,where we replace
$t+1 \leftarrow t$ and $t \leftarrow t-1$ prediction to the far future

- What happens when we want to predict further into future given only the evidence up to this date?
- It can be shown that predicted distribution for state vector converges towards one constant vector,the so called fixed point (for every t > mixing time):

$$P(X_t|e_{1:t}) = P(X_{t+1}|e_{1:t+1})$$

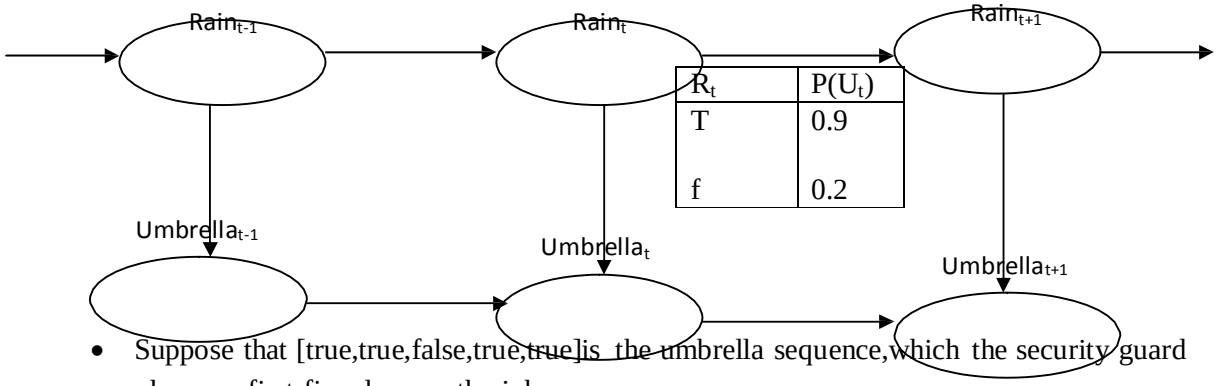
- This is called a stationary distribution of the Markov process,and the time required to reach this stationary state is called the mixing time.
- Stationary distribution of the Markov process dooms to failure any attempt to predict the actual state for a number of steps ahead that is more than a small fraction of the mixing time.

3. Most likely sequence

- Given all evidence to date,we want to find the sequence of states that is most likely to have generated all the evidence,i.e. argmax $X_{1:t}$ $P(X_{1:t}|e_{1:t})$
- In the umbrella example,if the umbrella appears on each of the first three days and is absent on the fourth,then the most likely explanation is that it rained on the first three days and it did not rain on the fourth.

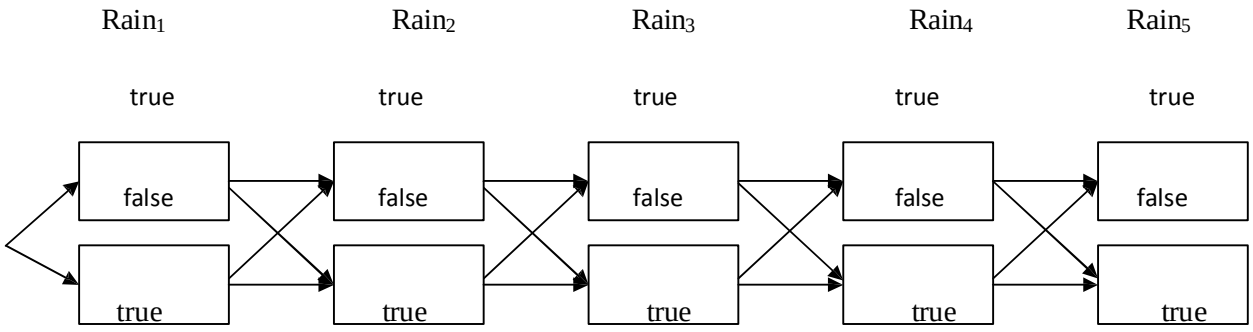
- Algorithms for this task are useful in many applications,including speech recognition,i.e. to find the most likely sequence of words, given series sounds,or the construction of bit strings transmitted over a noisy channel(cell phone),etc.

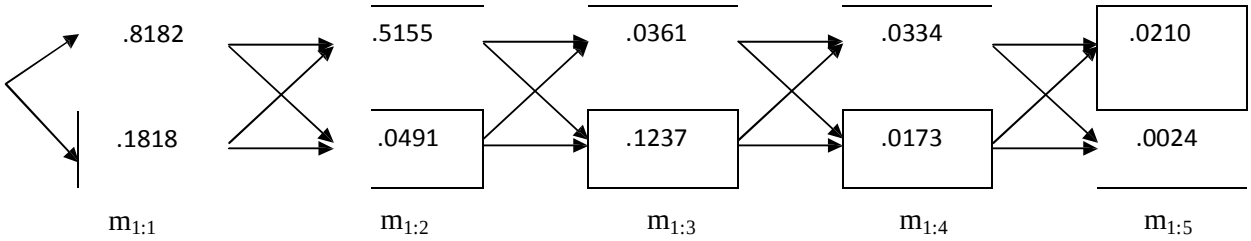
R_{t-1}	$P(R_t)$
T	0.7
F	0.3



R_t	$P(U_t)$
T	0.9
f	0.2

- Suppose that [true,true,false,true,true]is the umbrella sequence,which the security guard observes first five days on the job.
- What is the weather sequence most likely to explain this out of 2^5=32 possible sequences,i.e. argmax $X_{1:t}$ $P(X_{1:t}|e_{1:t})$?
- For each state,the bold arrow indicates its best predecessor as measured by the product of the preceding sequence probability $m_{1:t}$ and the transition probability $P(X_t|X_{t-1})$
- To derive the recursive formula,let us focus on paths that reach the state $Rain_5$ = true.the most likely path consists of the most likely path to some state at t=4 followed by the transition to $Rain_5$ = true.
- The state at t=4,which will become part of the path to $Rain_5$ = true is whichever maximizes the likelihood of that path.
- There is a recursive relationship between most likely paths to each state X_{t+1} and most likely paths to each state X_t.





- Viterbi algorithm:
  - Let us denoted by $m_{1:t}$ the probability of the best sequence reaching each state at time t.

$$M_{1:t} = \max_{x1....xt-1} P(X_1,.....,X_{t-1},X_t|e_{1:t})$$

  - Then the recursive relationship between most likely paths to each state X_{t+1} and most likely paths to each state X_t, reads

$$m_{1:t+1} = \max_{x1...xt} P(X_1,.....,X_t,X_{t+1}|e_{1:t+1})$$

$$=\alpha P(e_{t+1}|X_{t+1})\max_{xt}(P(X_{t+1}|X_t)\max_{x1...xt-1} P(X_1,....,X_{t-1},X_t|e_{1:t}))$$

  This is the viterbi formula

4.6 Hidden Markov model

- An HMM is a temporal probabilistic model in which the state of the process is described by a single discrete random variable.
- The possible values of the variable are the possible states of the world.
- The umbrella example described in the HMM,since it has just one state variable $Rain_t$. Additional state variables can be added to a temporal model while staying within the HMM framenetwork,but only by combining all the state variable into a single "megavariable" whose values are all possible tuples of values of the individual state variables.
- Simplified matrix algorithms:
- With a single,discrete state variable X_t,we can give concrete form to the representations of the transition model,and the forward and backward messages.
- Let the state variable X_t have values denoted by integers 1,….,S,where S is the number of possible states.
- The transition model $P(X_t|X_{t-1})$ becomes an S x S matrix T,where

$$T_{ij} = P(X_t = j|X_{t-1} = i)$$

  T_{ij} – probability of a transition from state I to state j.
- For example,the transition matrix for the umbrella world is

$$T = P(Xt|Xt\text{-}1) = \begin{pmatrix} 0.7 & 0.3 \\ 0.7 & 0.7 \end{pmatrix}$$

- We also put the sensor model in matrix form.In this case,because the value of the evidence variable E_t is known to be say e_t,we needuse only that part of the model specifying the probability that e_t appears.
- For each time step t,we construct a diagonal matrix O_t whose diagonal entries are given by the values $P(e_t|X_t = i)$ and whose entries are 0.

$$O_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}$$

- We use column vectors to represent the forward and backward messages,the computations become simple matrix-vector operations.

The forward equation becomes

$$F_{1:t+1} = \alpha\, O_{t+1}\, T^T f_{1:t} \quad \ldots\ldots\ldots\ldots(1)$$

and the backward equation becomes

$$b_{k+1:t} = TO_{k+1}\, b_{k+2:t} \quad \ldots\ldots\ldots\ldots(2)$$

- From these equations,we can see that the time complexity of the forward and backward algorithm applied to a sequence of length t is $O(S^2t)$.The space complexity is $O(St)$.
- Besides providing an elegant description of the filtering and smoothing algorithms for HMMs,the matrix formulation reveals opportunities for improved algorithms.
- The first is a simple variation on the forward-backward algorithm that allows smoothing to be carried out in constant space,independently of the length of the sequence.
- The idea is that smoothing for any particular time slice k requires the simultaneous presence of both forward and backward messages,$f_{1:k}$ and $b_{k+1:t}$.
- The forward-backward algorithms achieves this by storing the fs computed on the forward pass so that they are available during the backward pass.

$$f_{1:t} = \alpha' (T^T)^{-1} O_{t+1}^{-1} f_{1:t+1}$$

- The modified smoothing algorithm works by first running the standard forward pass to compute $f_{t:t}$ and then running the backward pass for both b and f together,using them to compute the smoothed estimate at each step.
- A second area in which the matrix formulation reveals an improvement is in online smoothing with a fixed lag.
- Let us suppose that the lag is d; that is,we are smoothing at time slice t-d,where the current time is t.By equation.

$$\alpha f_{1:t-d} b_{t-d+1:t}$$

for slice t-d.Then,when a new observation arrives,we need to compute

$$\alpha f_{1:t-d+1} b_{t-d+2:t+1}$$

for slice t-d+1.First,we can compute $f_{1:t-d+1}$ from $f_{1:t-d}$, using the standard filtering process.

- Computing the backward message incrementally is more trickly,because there is no simple relationship between the old backward message $b_{t-d+1:t}$ and the new backward message

$$b_{t-d+2:t+1}.$$

- Instead ,we will examine the relationship between the old backward message $b_{t-d+1:t}$ and the backward message at the front of the sequence,$b_{t+1:t}$.To do this,we apply equation(2) d times to get

$$b_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} TOi\right) b_{t+1:t} = B_{t-d+1:t}\, 1. \quad \ldots\ldots\ldots\ldots(3)$$

Where the matrix $B_{t-d+1:t}$ is the product of the sequence of T and O matrices.

- B can be thought of as a "transformation operator" that transforms a later backward message into an earlier one.

  $b_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} TOi\right) b_{t+2:t+1} = B_{t-d+2:t+1}\, 1.$ ……(4)

- Examining the product expressions in the above two equations(3) & (4),we see that they have a simple relationship:to get the second product,"divide" the first product by the first element TO_{t-d+1}, and multiply by the new last element TO_{t+1}.
- In matrix language,then there is a simple relationship between the old and new B matrices:

  $B_{t-d+2:t+1} = O_{t-d+1}^{-1}\, T^{-1}\, B_{t-d+1:t}\, TO_{t+1}.$ …….(5)

- This equation provides an incremental update for the B matrix,which in turn(eqn (4)) allows us to compute the backward message $b_{t-d+2:t+1}$.

ARTIFICIAL INTELLIGENCE

UNIT-V

EXPERT SYSTEMS

Expert systems - Architecture of expert systems, Roles of expert systems - Knowledge Acquisition – Meta knowledge, Heuristics. Typical expert systems - MYCIN, DART, XOON, Expert systems shells.

5.1 Learning from Observation:

- The idea behind learning is that percepts should be used not only for acting, but also for improving the agent's ability to act in the future.
- Learning takes place as the agent observes its interactions with the world and its own decision making process.
- Learning can range from trivial memorization of experience to the creation of a entire scientific theory, as exhibited like Albert Einstein.

5.1.1 Forms of Learning:

- Learning agent is a performance element that decides what actions to take and a learning element that modifies the performance element so that better decisions can be taken in the future.
- There are large variety of learning elements
- The design of a learning element is affected by following three major issues,
    - Which components of performance element are to be learned.
    - What feedback is available to make these components learn
    - What representation is used for the component.
- The components of these agents includes the following,
    - A direct mapping from conditions on current state to actions
    - A means to infer relevant properties of the world from the percept sequence
    - Information about the way the world evolves and about the results of possible action the agent can take
    - Utility information indicating the desirability of world states
    - Action-value information indicating the desirability of action
    - Goals that describe classes of states whose achievement maximizes the agent utilty
- Each of the component can be learned from appropriate feedback
    - For Example: - An agent is training to become a taxi driver.
    - The various components in the learning are as follows,

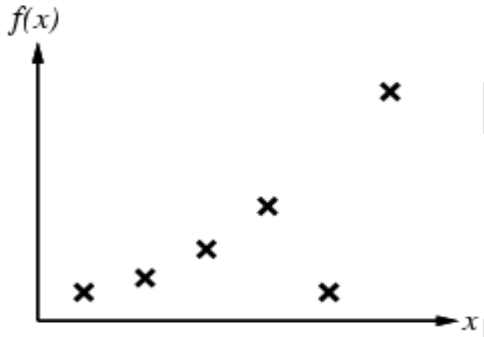
  - Everytime when the instructor shouts "Brake" the agent learn a condition – action rule for when to brake.
  - By seeing many images, agent can learn to recognize them
  - By trying actions and observing the results, agent can learn the effect of actions (i.e.) braking on a wet road – agent can experience sliding
  - The utility information can be learnt from desirability of world states, (i.e.) if the vehicle is thoroughly shaken during a trip, then customer will not give tip to the agent, which plans to become a taxi driver
- The type of feedback available for learning is also important.
- The learning can be classified into following three types.
  - Supervised learning
  - Unsupervised learning
  - Reinforcement learning
- **Supervised Learning:-**
  - It is a learning pattern, in which
    - Correct answers for each example or instance is available
    - Learning is done from known sample input and output
    - For example: - The agent (taxi driver) learns condition – action rule for braking – this is a function from states to a Boolean output (to brake or not to brake). Here the learning is aided by teacher who provides correct output value for the examples.
- **Unsupervised Learning:-**
  - It is learning pattern, in which
    - Correct answers are not given for the input.
    - It is mainly used in probabilistic learning system.
- **Reinforcement Learning:-**
  - Here learning pattern is rather than being told by a teacher.
  - It learns from reinforcement (i.e.) by occasional rewards
  - For example:- The agent (taxi driver), if he does not get a trip at end of journey, it gives him a indication that his behavior is undesirable.

5.2 Inductive Learning

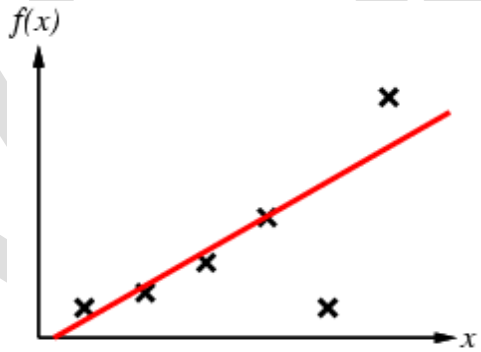
- Learn a function from example,
- For example:- *f* is target function
  An example is a pair (x, *f*(x)) where x = input and *f*(x) = output of the function is applied to x
- The pure inductive inference or induction is "given a training set of example of *f*, return a function *h* that approximates *f*.
- Where the function *h* is called hypothesis
- This is a simplified model of real learning, because it
  - Ignores prior knowledge
  - Assumes a deterministic, observable "environment".
- A good hypothesis will generalize well, i.e., able to predict based on unseen examples

5.2.1 Inductive learning method:-

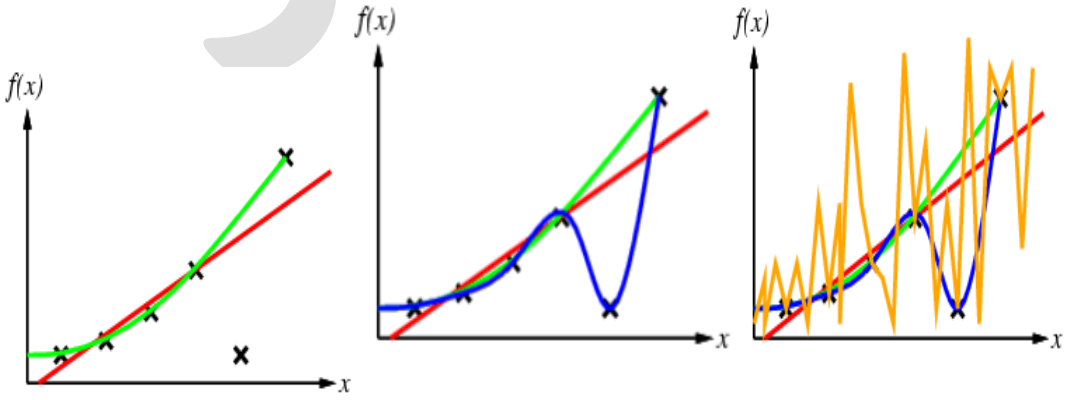
-
- Goal is to estimate real underlying functional relationship from example observations
- Construct / adjust *h* to agree with *f* on training set (*h* is consistent if it agrees with *f* on all example)
- For example:- Curve fitting example
- Given



- Linear hypothesis:



- Curve fitting with various polynomial hypothesis for the same data



- Ockham's razor : prefer simplest hypothesis consistent with the data

- Not-exactly-consistent may be preferable over exactly consistent
    - Nondeterministic behavior
    - Consistency even not always possible
- Nondeterministic functions : trade-off complexity of hypothesis / degree of fit

5.3 Decision Trees

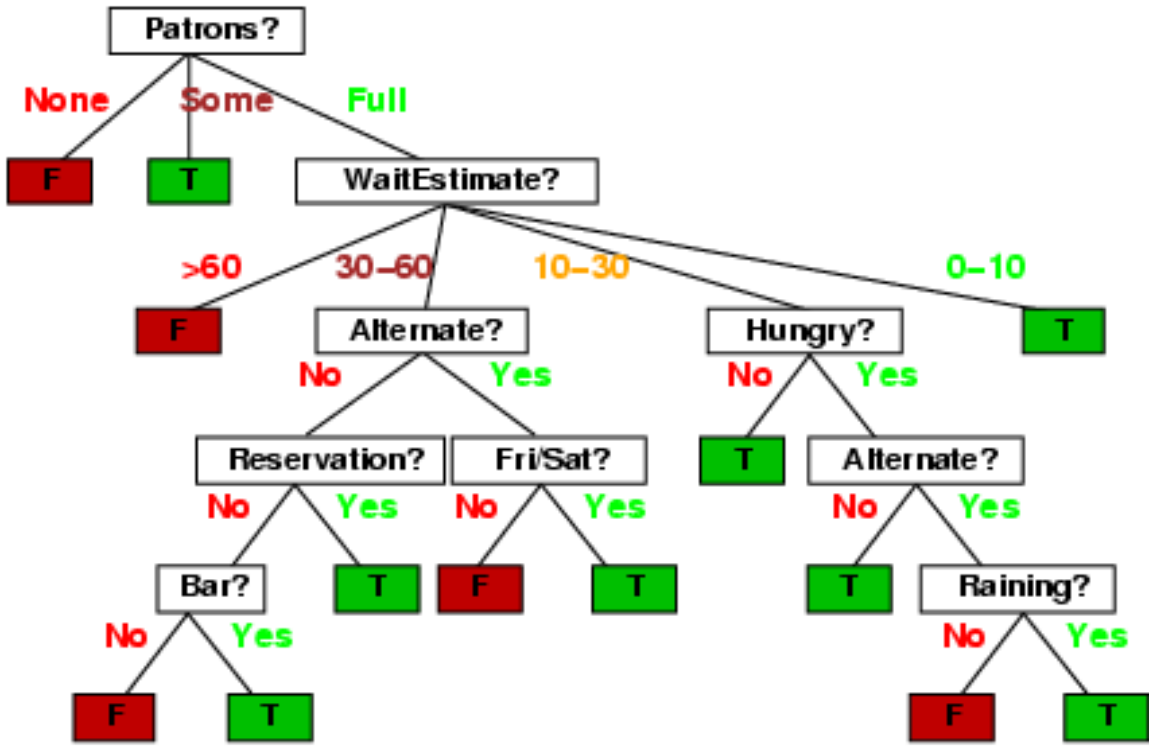
- Decision tree is one of the simplest learning algorithms.
- A decision tree is a graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility.
- It can be used to create a plan to reach a goal.
- Decision trees are constructed to help with making decisions.
- It is a predictive model.

5.3.1 Decision trees as performance elements:-

- Each interior node corresponds to a variable; an arc to a child represents a possible value of that variable.
- A leaf represents a possible value of target variable given the values of the variables represented by the path from the root.
- The decision tree takes object or situation described by set of **attributes** as input and decides or predicts output value.
- The output value can be Boolean, discrete or continuous.
- Learning a discrete valued function is called **classification learning.**
- Learning a continuous valued function is called **regression.**
- In Boolean classification it is classified as true (positive) or false (negative).
- A decision tree reaches its destination by performing a sequence of tests.
- Each interior or internal node corresponds to a test of the variable; an arc to a child represents possible values of that test variable.
- The decision tree seems to be very for humans.
- For Example:-
    - A decision tree for deciding whether to wait for a table at a restaurant.
    - The aim here is to learn a definition for the **goal predicate.**
    - we will see how to automate the task the following attributes are decided.
        - Alternate: is there an alternative restaurant nearby?
        - Bar: is there a comfortable bar area to wait in?
        - Fri/Sat : is today Friday or Saturday?
        - Hungry: are we hungry?
        - Patrons : number of people in the restaurant [the values are None, Some, Full]
        - Price : price range [$, $$, $$$]
        - Raining: is it raining outside?
        - Reservation: have we made a reservation?
        - Type : kind of restaurant [French, Italian, Thai, Burger]
        - WaitEstimate : estimated waiting time by the host [0-10, 10-30, 30-60, >60]
        - The following table described the example by attribute values (Boolean, Discrete, Continuous) situations where I will / won't wait for a table.

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

- The following diagram shows the decision tree for deciding whether to wait for a table



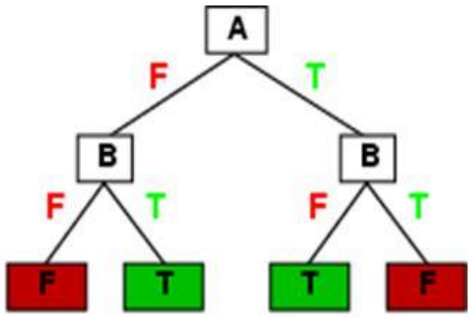
- The above decision tree does not use **price** and **type** as irrelevant.
- For example:- if the Patrons = full and the Wait Estimate = 0-10 minutes, it will be classified as positive(yes) and the person will wait for the table
- Classification of example is positive (T) or negative (F) shown in both table and in decision tree.

5.3.2 Expressiveness of decision trees

- Decision trees can express any function of the input attributes
- E.g., for Boolean functions, truth table row path to leaf
- The following table shows the truth table of A XOR B

A	B	A XOR B
F	F	F
F	T	T
T	F	T
T	T	F

- The following diagram shows the decision tree of XOR gate



- There is a consistent decision tree for any training set with one path to leaf for each example [unless *f* nondeterministic in *x*] but it probably won't generalize to new examples
- Applying Ockham's razor : smallest tree consistent with examples
- Able to generalize to unseen examples
    - No need to program everything out / specify everything in detail

'true' tree = smallest tree?

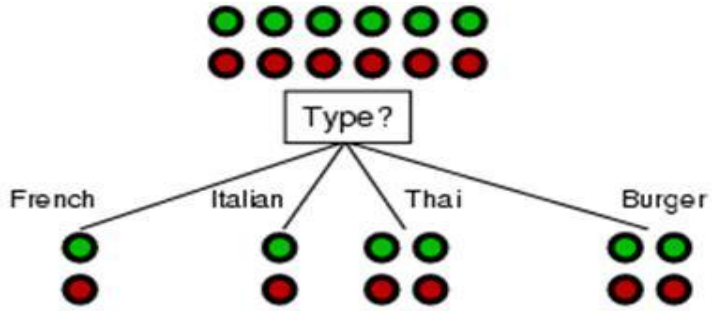
Advantages of Decision Tree:

- They are simple to understand and interpret
- They require little data preparation
- If uses a white box model.
- It is possible to validate a model using statistical tests, hence robust.
- Perform well with large data in a short time.

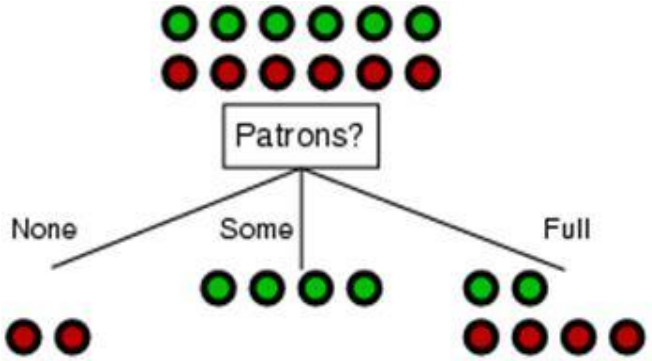
5.3.3 Decision tree learning:

- Unfortunately, finding the 'smallest' tree is intractable in general
- New aim : find a 'smallish' tree consistent with the training examples
- Idea : [recursively] choose 'most significant' attribute as root of [sub]tree
- 'Most significant' : making the most difference to the classification
- Idea : a good attribute splits the examples into subsets that are [ideally] 'all positive' or 'all negative'
- Patrons? is a better choice

- The following diagram shows the splitting the examples by testing on attributes



- The above diagram Splitting on Type brings us no nearer to distinguishing between positive and negative examples
- The below diagram Splitting on Patrons does a good job of separating positive and negative examples



- The following table shows the Decision Tree Learning Algorithm,

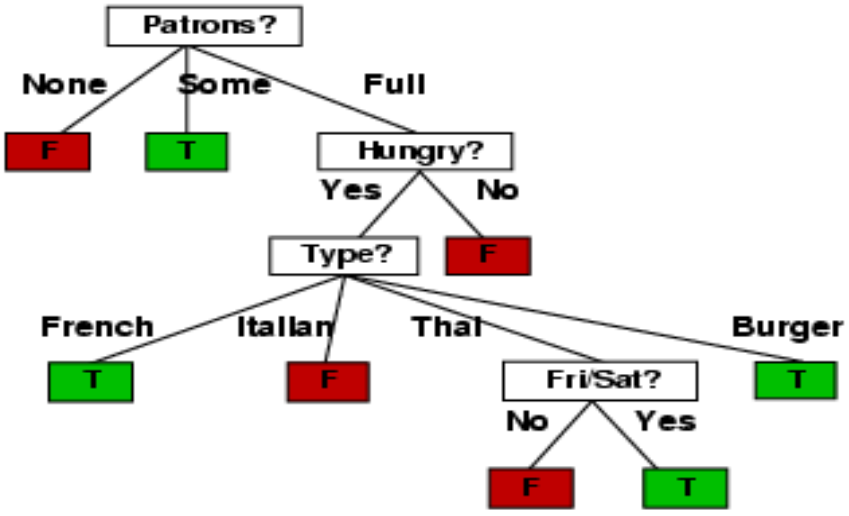
function DTL(*examples, attributes, default*) **returns** a decision tree

if *examples* is empty **then return** *default*
else if all *examples* have the same classification **then return** the classification
else if *attributes* is empty **then return** MODE(*examples*)
else
  $best \leftarrow$ CHOOSE-ATTRIBUTE(*attributes, examples*)
  $tree \leftarrow$ a new decision tree with root test *best*
  for each value v_i of *best* **do**
    $examples_i \leftarrow$ {elements of *examples* with $best = v_i$}
    $subtree \leftarrow$ DTL($examples_i, attributes - best,$ MODE(*examples*))
    add a branch to *tree* with label v_i and subtree *subtree*
  return *tree*

- The following tree shows the decision tree induced from the training data set as follows,

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

- substantially simpler solution than 'true' tree
- More complex hypothesis isn't justified by small amount of data



5.3.4 Using Information theory:

- Information content [entropy] :
  - $I(P(v_1), \ldots, P(v_n)) = \Sigma_{i=1} -P(v_i) \log 2\ P(v_i)$
  - For a training set containing *p* positive examples and *n* negative examples

$$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

- Specifies the minimum number of bits of information needed to encode the classification of an arbitrary member

Information Gain:

- Chosen attribute *A* divides training set *E* into subsets *E*1, … , *Ev* according to their values for *A*, where *A* has *v* distinct values

$$remainder(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

- Information gain [IG] : expected reduction in entropy caused by partitioning the examples

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

- Information gain [IG] : expected reduction in entropy caused by partitioning the examples

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

- Choose the attribute with the largest IG
- For Example:- For the training set : $p = n = 6$, I(6/12, 6/12) = 1 bit
- Consider Patrons? and Type? [and others]

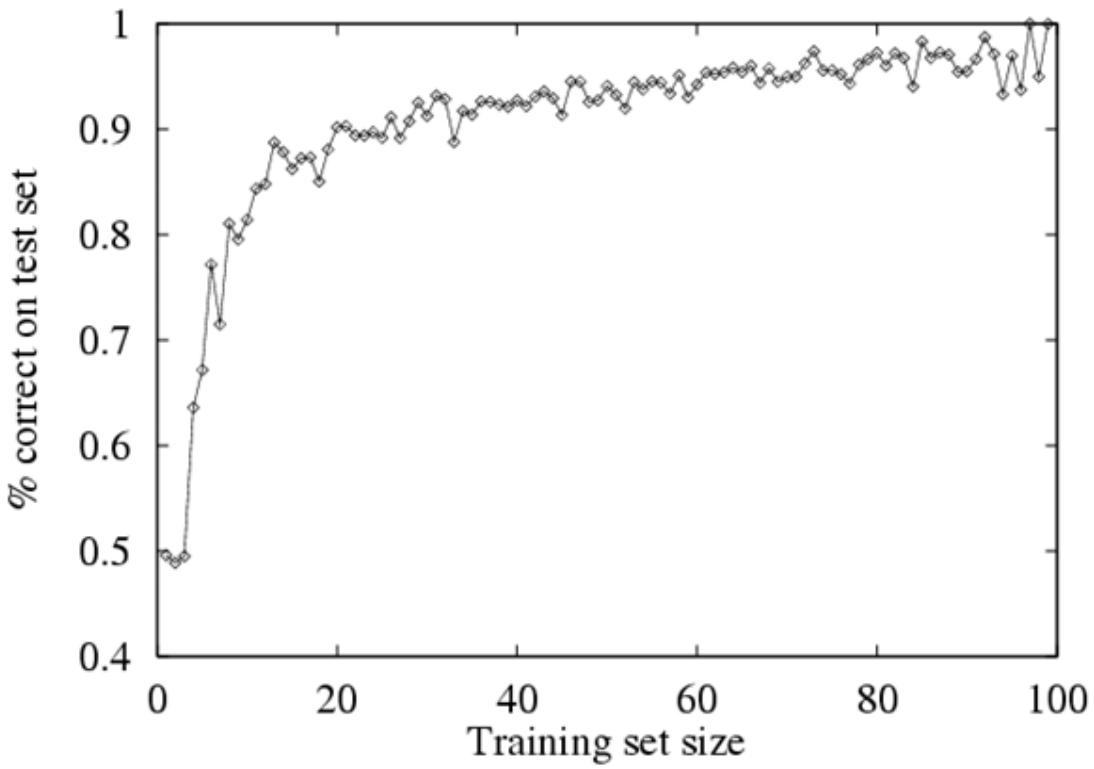
$$IG(Patrons) = 1 - [\frac{2}{12} I(0,1) + \frac{4}{12} I(1,0) + \frac{6}{12} I(\frac{2}{6}, \frac{4}{6})] = .0541 \text{ bits}$$

$$IG(Type) = 1 - [\frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4})] = 0 \text{ bits}$$

- Patrons has the highest IG of all attributes and so is chosen as the root

5.3.5 Assessing the performance of the learning Algorithm:

- A learning algorithm is good if it produces hypothesis that do a good job of predicting the classification of unseen examples.
- Obviously, a prediction is good if it turns out to be true, so we can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it.
- We do this on a set of examples known as the **test set.**
- The following are the steps to assess the performance,
    1. Collect a large set of examples
    2. Divide it into two disjoint sets: the **training set** and the **test set**
    3. Apply the learning algorithm to the training set, generating a hypothesis *h*.
    4. Measure the percentage of examples in the test set that are correctly classified *h*.
    5. Repeat steps 1 to 4 for different sizes of training sets and different randomly selected training sets of each size.
- The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set.
- This function can be plotted on a graph, giving what is called the **learning curve** for the algorithm on the particular domain.
- The following diagram shows the learning curve for DECISION-TREE-LEARNING with the above attribute table example.



- In the graph the training set grows, the prediction quality increases.
- Such a curves are called **happy graphs**.

5.4 Explanation Based Learning:

- Explanation-based learning is a method for extracting general rules from individual observations
- Human appear to learn quite a lot from example
- Basic idea: Use results from one examples problem solving effort next time around.
- when an agent can utilize a worked example of a problem as a problem-solving method, the agent is said to have the capability of **explanation-based learning (EBL).**
- This is a type of **analytic learning.**
- The advantage of explanation-based learning is that, as a **deductive mechanism,** it requires only a single training example (inductive learning methods often require many training examples)
- To utilize just a single example most **EBL** algorithms require all of the following,
  - The training example
  - A Goal concept
  - An Operationality Criteria
  - A Domain theory
- An **EBL** accepts four kinds of input as follows,
  - **A training example:-** what the learning sees in the world
  - **A goal concept:-** a high level description of what the program is supposed to learn
  - **An operational criteria:-** a description of which concepts are usable
  - **A domain theory:-** a set of rules that describe relationships between objects and actions in a domain

- The domain theory has two types as,
  - **Explanation:** - the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.
  - **Generalisation:** - the explanation is generalized as far possible while still describing the goal concept

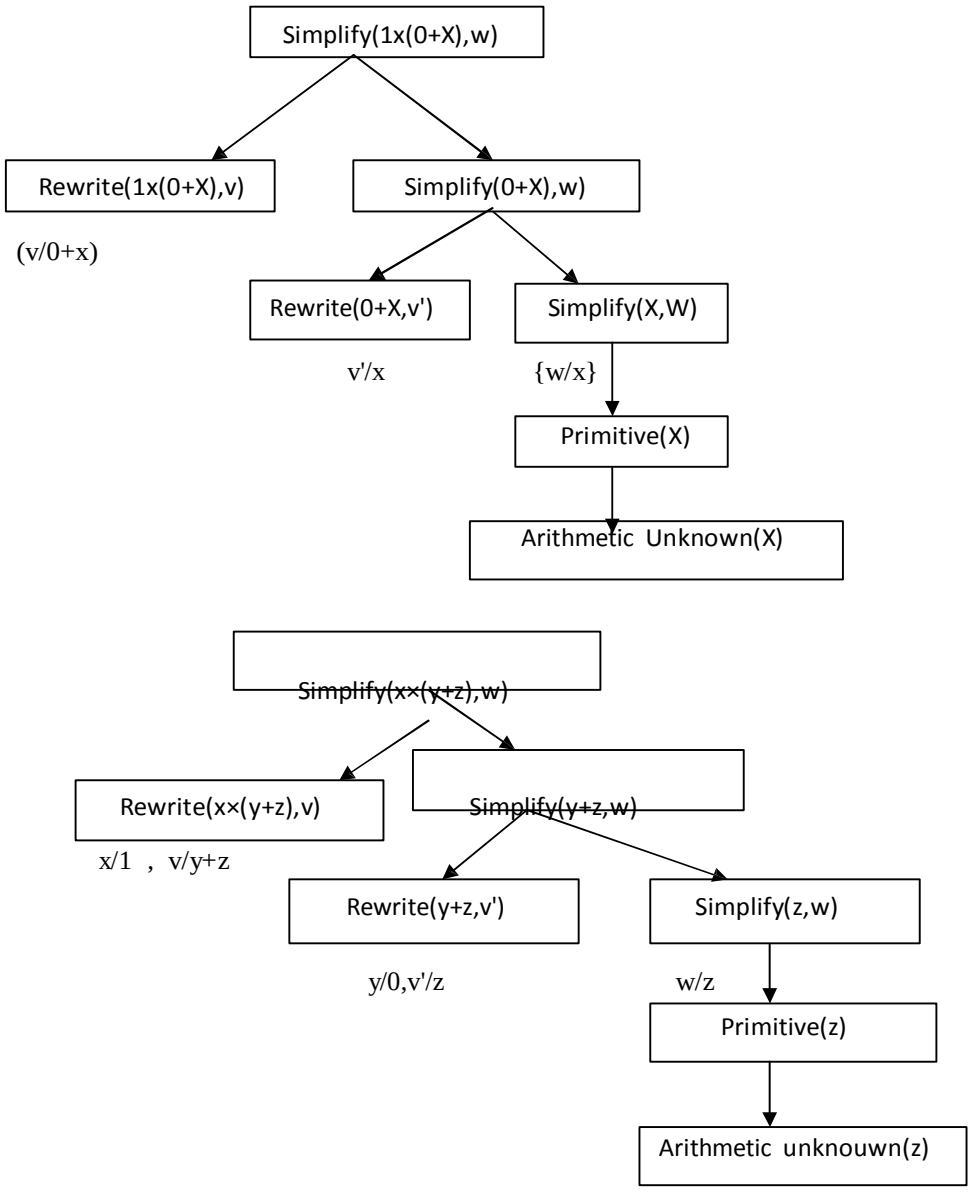
- **For Example:-**
  - Cary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting a lizard on the end of a pointed stick.
  - He is watched by an amazed crowd of less intellectual contemporaries.
  - In this case, the caveman generalize by explaining the success of the pointed stick which supports the lizard and keeps the hand away from the fire.
  - This explanation can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft bodies.
  - This kind of generalization process is said to be **Explanation based Learning.**
  - The **EBL** procedure is very much domain theory driven with the training example helping to focus the learning.
  - Entailment constraints satisfied by **EBL** is
    - *Hypothesis* $\wedge$ *Descriptions* |= *Classifications*
    - *Background* |= *Hypothesis*

5.4.1 Extracting rules from examples:

- **EBL** is a method for extracting general rules from individual observations.
- The basic idea is first to construct an explanation of the observation using prior knowledge.
- Consider the problem of differentiating and simplifying the algebraic expressions.
- If we differentiate the expression X^2 with respect to X, we obtain 2X.
- The proof tree for Derivative(X^2, X) = 2X is too large to use, so we will use a simpler problem to illustrate the generalization method.
- Suppose our problem is to simplify 1 x (0 + X).
- The knowledge base includes the following rules
  - *Rewrite(u, v)* $\wedge$*Simplify(v, w)* $\Rightarrow$ *Simplify(u, w)*
  - *Primitive(u)* $\Rightarrow$ *Simplify(u, u)*
  - *ArithmeticUnknown(u)* $\Rightarrow$ *Primitive(u)*
  - *Number(u)* $\Rightarrow$ *Primitive(u)*
  - *Rewrite(1 x u, u)*
  - *Rewrite(0 x u, u)*

- **EBL Process Working**
- The EBL work as follows
  1. Construct a proof that the goal predicate applies to the example using the available background knowledge
  2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.
  3. Construct a new rule whose left hand side consists of leaves of the proof tree and RHS is the variabilized goal.
  4. Drop any conditions that are true regardless of the values of the variables in the goal.

- In the diagram, the first tree shows the proof of original problem instance, from which we can derive
  - *ArithmeticUnknown(z) = Simplify(1 x (0 + z), z)*
- The second tree shows the problem where the constants are replaced by variables as generalized proof tree.



5.4.2 Improving efficiency:

- The generalized proof tree mentioned above gives or yields more than one generalized rule.
- For example if we terminate, or **PRUNE,** the growth of the right hand branch in the tree when it reached the *primitive* step, we get the rule as,
  - *Primitive(z) ⇒ Simplify(1 X (0 + z), z)*

- This rule is a valid as, but more general than, the rule using *ArithmeticUnknow,* because it covers cases where z is a number.
- After pruning the step,
  - *Simplify (y + z, w),* yielding the rule
  - *Simplify (y + z, w)* $\Rightarrow$ *Simplify (1 X(y + z), w)*
- The problem is to choose which of these rules.
- The choice of which rule to generate comes down to the question of efficiency.
- There are three factors involved in the analysis of efficiency gains from **EBL** as,
  - Adding large number of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in case where they not a solution. It increases the **branching factor** in the search space.
  - To compensate the slowdown in reasoning, the derived rules must offer significant increase in speed for the cases that they do not cover. This increase occurs because the derived rules avoid dead ends but also because they short proof also.
  - Derived rule is as general as possible, so that they apply to the largest possible set of cases.

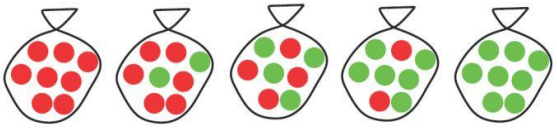
5.5 Statistical Learning Methods:

- Agents can handle uncertainty by using the methods of probability and decision theory.
- But they must learn their probabilistic theories of the world from experience.
- The learning task itself can be formulated as a process of probabilistic inference.
- A Bayesian view of learning is extremely powerful, providing general solutions to the problem of noise, overfitting and optimal prediction.
- It also takes into account the fact that a less than omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

5.5.1 Statistical Learning

- The key concepts of statistical learning are **Data** and **Hypotheses.**
- **Data** are evidence (i.e.) instantiations of some or all of the random variables describing the domain.
- **Hypotheses** are probabilistic theories of how the domain works, including logical theories as a special case.
- **For Example:-**
  - The favorite surprise candy comes in two flavors as Cherry and Lime
  - The manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor.
  - The candy is sold in very large bags of which there are known to be five kinds-again, indistinguishable from the outside:

    h1: 100% cherry candies
    h2: 75% cherry candies + 25% lime candies
    h3: 50% cherry candies + 50% lime candies
    h4: 25% cherry candies + 75% lime candies
    h5: 100% lime candies



- Given a new bag of candy the random variable **H** (for hypotheses) denotes the type of tile bag, with possible values h1 through h5. **H** is not directly observable.
- As the pieces of candy are opened and inspected, data are revealed as D1, D2…Dn in which each D is a random variable with possible values Cherry and Lime.
- **The basic task** faced by the agent is to predict the flavor of the next piece of candy

5.5.1.1 Bayesian Learning:

- Bayesian Learning calculates the probability of each hypothesis, given the data and makes predictions by using all the hypotheses, weighted by their probabilities.
- In this way learning is reduced to probabilistic inference.
- Let D be all data, with observed value d, then probability of a hypothesis h_i, using Bayes rule

$$P(h_i|d) = a\ P(d \mid h_i)P(h_i)$$

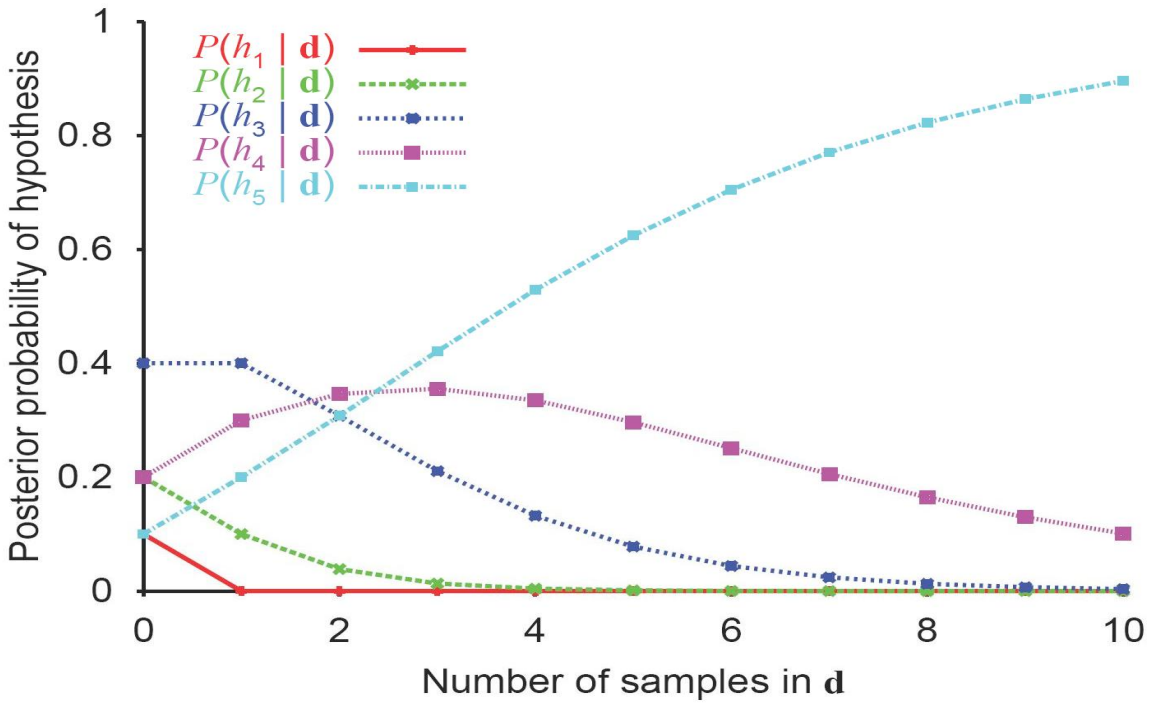
- For prediction about quantity X :

$$P(X|d) = \sum P(X|d,h_i)P(h_i|d) = \sum P(X|h_i)P(h_i|d)$$

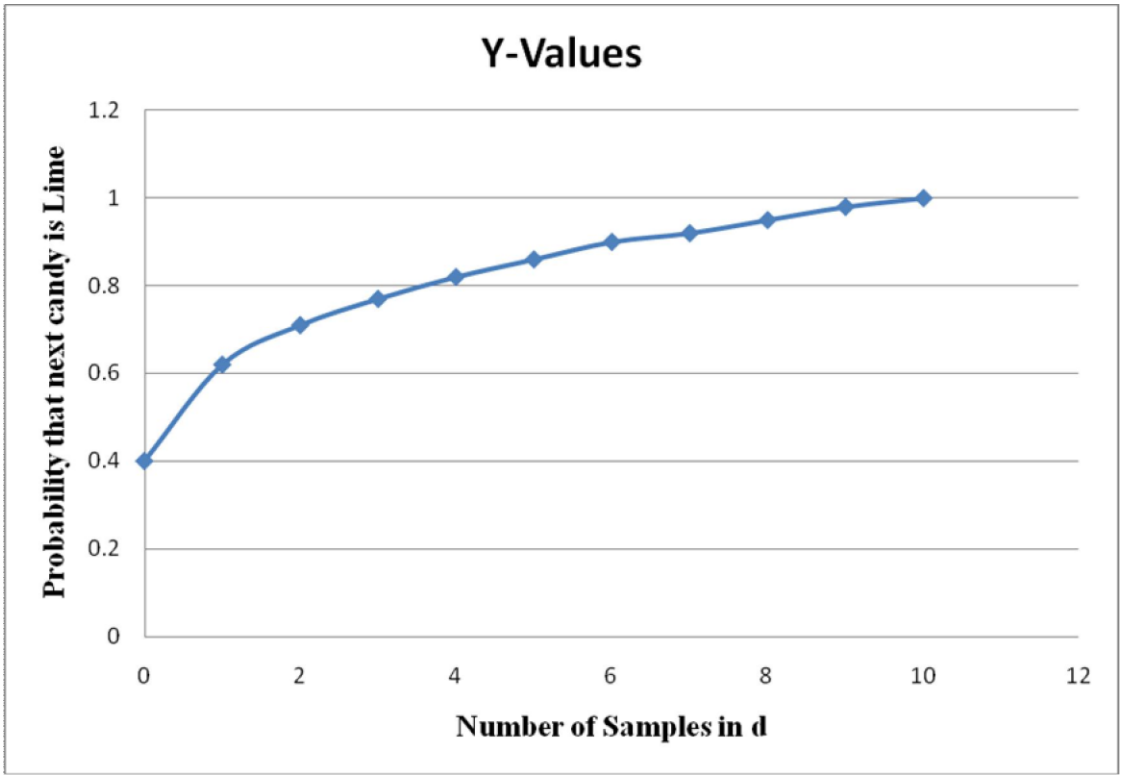
- Where it is assumed that each hypothesis determines a probability distribution over **X.**
- This equation shows that predictions were weighted averages over the predictions of the individual hypothesis
- The key quantities in the Bayesian approach are the
  - Hypothesis ***Prior,*** $P(h_i)$
  - **Likelihood** of the data under each hypothesis, $P(d \mid h_i)$
- For candy example, assume the time being that the prior distribution over h1,….h5 is given by (0.1,0.2,0.4,0.2,0.1), as advertised by the manufacturer.
- The likelihood of the data is calculated under the assumption that the observations are i..i..d, that is i= independently, i= identically and d= distributed So that

$$P(d \mid h_i) = \prod_j P(d_j \mid h_i)$$

- The following figure shows how the posterior probabilities of the five hypotheses change as the sequence of 10 Lime is observed.
- Notice that the probabilities start out at their prior values. So h1 is initially the most likely choice and remains so after 1 Lime candy is unwrapped.
- After 2 Lime candies are unwrapped, h1 is most likely; after 3 or more, h5 is the most likely.



- The following figure shows the predicted probability that the next candy is Lime as expected, it increases monotonically toward 1



5.5.1.2 Characteristics of Bayesian Learning:

- The true hypothesis eventually dominates the Bayesian prediction. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will vanish, because the probability of generating uncharacteristic data indefinitely is vanishingly small.
- More importantly, the Bayesian prediction is optimal, whether the data set is small or large.

- For real learning problems, the hypothesis space is usually very large or infinite.
- In most cases choose the approximation or simplified methods.

5.5.1.2.1 Approximation

- Make predictions based on a single most probable hypothesis *hi* that maximizes P(hi|d).
- This is often called a maximum a posteriori or MAP hypothesis.
- Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $P(X|d) \approx P(X| h_{MAP})$.
- In candy example, $h_{MAP} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0 a much more dangerous prediction than the Bayesian prediction of 0.8 shown in the above graphs.
- As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable.
- **Finding MAP hypothesis is much easier than Bayesian Learning is more advantage.**

5.5.2 Learning with Complete Data:

- The statistical learning method begins with **parameter learning with complete data.**
- A **parameter learning** task involves finding the numerical parameter for the probability model.
- The structure of the model is fixed.
- Data are complete when each data point contains values for every variable in the probability model.
- Complete data simplify the problem of learning the parameters of complex model.

5.5.1.1 Maximum Likelihood Parameter Learning: Discrete Models

- Suppose we buy a bag of lime and cherry candy from a manufacturer whose lime-cherry proportion are completely unknown.
- The fraction can be anywhere between 0 and 1.
- The parameter in this case is θ, which is the proportion of cherry candies, and the hypothesis is h_θ.
- The proportion of lime is $(1-\theta)$.
- We assume all the proportions are known a priori then Maximum Likelihood approach can be applied.
- If we model the situation in Bayesian network, we need just one random variable called **Flavor** it has values cherry and lime.
- The probability of cherry is θ`.
- If we unwrap *N* candies, of which *C* are cherries and *L=N-C* are limes.
- The likelihood of the particular set is,

$$P(d / h_\theta) = \prod_{J=1}^{N} P(dj \setminus h_\theta) = \theta^c .(1-\theta)^L$$

- The maximum-likelihood hypothesis is given by the value of e` that maximizes the expression.
- It can be obtained by maximizing the **log likelihood.**

$$L1(d \mid h_\theta) = P(d \mid h_\theta) = \sum_{J=1}^{N} \log(P(d_j \setminus h_\theta) = c \log\theta + L \log(1-\theta)$$

- To find the ML value of θ differentiate wrt θ and then equate resulting to zero

$$\frac{JL(d\setminus h\theta)}{d\theta} = \frac{c}{\theta} - \frac{L}{1-\theta} \quad , \quad \frac{c}{\theta} - \frac{L}{1-\theta} = 0, \ \theta - \frac{c}{c+l} \quad , \quad \theta - \frac{c}{N}$$ where c+l = N

- The standard method for maximum likelihood parameter learning is given by
  - Write down an expression for the likelihood of the data as a function of the parameters
  - Write down the derivative of the log likelihood with respect to each parameter.
  - Find the parameter values such that the derivatives are zero
- The most important fact is that, with complete data, the maximum-likelihood parameter learning problem for a Bayesian network

5.5.1.2 Maximum Likelihood Parameter Learning: Continuous Models

- Continuous variables are ubiquitous (everywhere) in real world applications.
- Example of Continuous probability model is linear-Gaussian model.
- The principles for maximum likelihood learning are identical to discrete model.
- Let us take a simple case of learning the parameters of a Gaussian density function on a single variable.
- The data are generated as follows

$$P(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

- Parameters of this model μ = mean and σ = Standard deviation.
- Let the observed values be $x_1, x_2, \ldots\ldots x_N$
- Then the log likelihood is given as

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{(x-\mu)^2}{2\sigma^2}} = N = (-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2}$$

- Setting the derivatives to zero as usual, we obtain

$$\frac{\partial L}{\partial \mu} = \frac{1}{\sigma^2} \sum_{j=1}^{N} (x-\mu) = 0 \qquad \Rightarrow \mu = \frac{\sum_j x_j}{N}$$

$$\frac{\partial L}{\partial \sigma} = \frac{N}{\sigma} + \frac{1}{\sigma^3} \sum_{j=1}^{N} (x_j-\mu)^2 = 0 \qquad \Rightarrow \sigma = \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}}$$

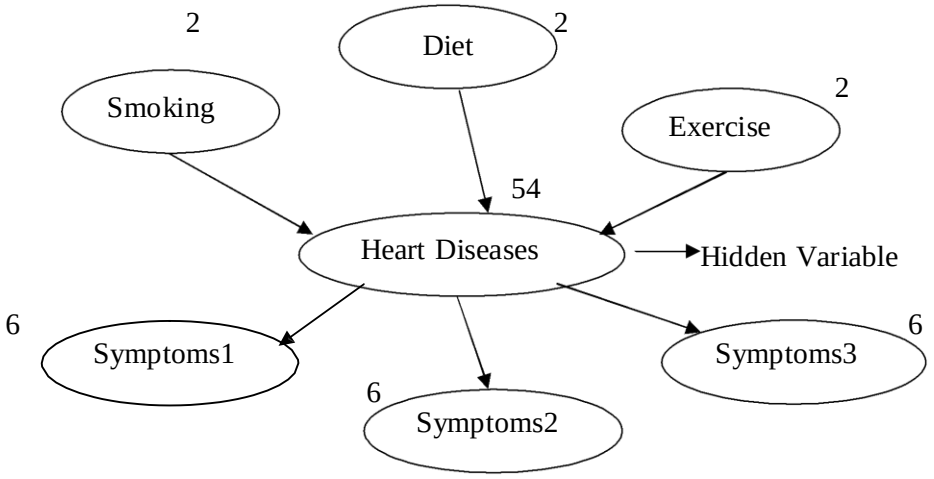
- Maximum likelihood value of the mean is the simple average.

- Maximum likelihood value of the standard deviation is the square root of the simple variance.

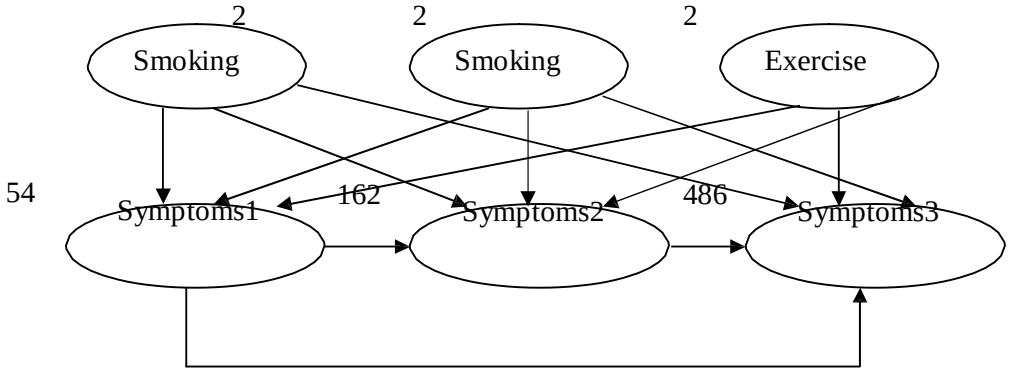
5.5.3 Learning with Hidden Variables:

1. Many real world problems have hidden variables (or) latent variables which are not observable in the data that are available for learning.
2. For Example:- Medical record often include the observed symptoms, treatment applied and outcome of the treatment, but seldom contain a direct observation of disease itself.

   Assumed the diagnostic model for heart disease. There are three observable predisposing factors and 3 observable symptoms. Each variable has 3 possible values (none, moderate and severe)

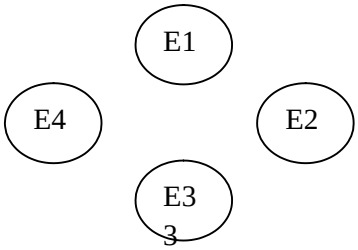


   If hidden is removed the total number of parameters increases from 78 (54 + 2 + 2 + 2 + 6 + 6 + 6) to 708

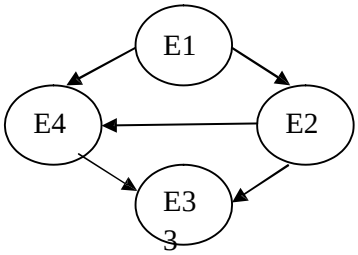


3. Hidden variables can dramatically reduce the number of parameters required to specify the Bayesian network, there by reduce the amount of data needed to learn the parameters.
4. It also includes estimating probabilities when some of the data are missing.
5. The reason we learn Bayesian network with hidden variable is that it reveals interesting structures in our data.
6. Consider a situation in which you can observe a whole bunch of different evidence variables, Er through En. They are all different symptoms that a patient might have.

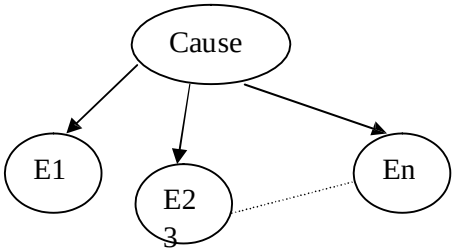
Different systems:



If the variables are conditionally dependent on one another, we will get a highly connected graph that representing the entire joint distribution between the variables.



7. The model can be made simpler by introducing an additional "cause" node. It represents the underlying disease state that was causing the patient symptoms.



This will have O(n) parameters, because the evident variables are conditionally independent given the causes.

8. Missing data
   - Imagine that we have 2 binary variables A and B that are not independent. We try to estimate the Joint distribution.

A	B
1	1
1	1
0	0
0	0
0	0
0	H
0	1
1	0

- It is done by counting how many were (true, true) and how may (false, false) and divide by the total number of cases to get maximum likelihood estimate.
- The above data set has some data missing (denoted on "H"). There's no real way to guess the value during our estimation problem.
- The missing data items can be independent of the value it would have had. The data can be missed if there is a fault in the instrument used to measure.
- For Example:- Blood pressure instrument fault, so blood pressure data can be missing)

9. We can ignore missing values and estimate parameters

Estimate Parameters

	$\underline{0}$ A	1 A
$0\bar{B}$	3/7	1/7
1 B	1/7	2/7

	$\frac{0}{A}$	1 A
$0\bar{B}$	0.429	0.143
1 B	0.143	0.285

We can consider H = 0 (or) H = 1

$\therefore \log \Pr(D/M) = \log(\Pr/ D, H = 0/M) + \Pr(D, H = 1/M))$

= 3 log 0.429 + 2 log 0.143 + 2 log 0.285 + log (0.429 + 0.143)

= -9.498 Maximum likelihood score

10. We also try to fit it with best value.
For the above cause consider H=0, Estimated parameters as follows,

	$\frac{0}{A}$	1 A
$0\bar{B}$	4/8	1/8
1 B	1/8	2/8

	$\underline{0}$ A	1 A
$0\bar{B}$	0.5	0.125
1 B	0.125	0.25

$\therefore \log \Pr(D/M) = \log(\Pr/ D, H = 0/M) + \Pr(D, H = 1/M))$

= 3 log 0.5 + 2 log 0.125 + 2 log 0.25 + log (0.5 + 0.125)

= -9.481

There is an improvement in likelihood value.

11. We will employ some soft assignment technique. we fill the value of the missing variable by using our knowledge of the joint distribution over A, B and compute a distribution over H.

	$\frac{0}{A}$	1 A
$0\bar{B}$	0.25	0.25
1 B	0.25	0.25

Initial guess Uniform distribution.

Compute probability distribution over H

$\Pr(H/D, \theta_0) = \Pr(H/D^6, \theta_0)$ because it refers to 6^{th} case in the observed data in the table.

$= \Pr(H/D^6, \theta_0)$

$= \Pr(B/\neg A, \theta_0)$

because missing variable is B and the observed one is not A, we need the probability of B given not A.

$\Pr(B / \neg A, \theta_0) = \Pr(\neg A, B / \theta_0) / \Pr(\neg A / \theta_0)$

$= \frac{0.25}{0.5} = 0.5$

H = 0 probability is 0.5

H = 1 probability is 0.5

A	B
1	**1**
1	**1**
0	**0**
0	**0**
0	**0**
0	**0,0.5** **1,0.5**
0	**1**
1	**0**

Now maximum likelihood estimation using expected counts.

So expected parameter is

	$\frac{0}{A}$	$\frac{1}{A}$
$0\bar{B}$	3.5/8	1/8
1 B	1.5/8	2/8

	$\frac{0}{A}$	$\frac{1}{A}$
$0\bar{B}$	0.4375	0.125
1 B	0.1875	0.25

New estimate is

$\Pr(H / D, Q1) = \Pr(\neg, B / Q1) / \Pr(\neg A / Q1)$

$= \frac{0.1875}{0.625}$

So the new table is = 0.3

A	B
1	**1**
1	**1**
0	**0**
0	**0**

0	**0**
0	**0,0.7** **1,0.3**
0	**1**
1	**0**

	$\frac{0}{A}$	$\frac{1}{A}$
$0\bar{B}$	3.7/8	1/8
1 B	1.3/8	2/8

	$\frac{0}{A}$	$\frac{1}{A}$
$0\bar{B}$	0.4625	0.125
1 B	0.1625	0.25

$\therefore$ theta2 is θ_2 is

$$\Pr(H / D, \theta_2) = \Pr(\neg Ar, B / \theta_2) / \Pr(\neg A / \theta_2)$$

$$= \frac{0.1625}{0.625} = 0.26$$

log likelihood is increasing

$\log \Pr(0 / \theta_0) = -10.3972$

$\log \Pr(D / \theta_1) = -9.4760$

$\log \Pr(D / \theta_2) = -9.4524$

Since all values are negative it is in increasing order.

$\therefore$ We have to choose the best value

12. The above iterative process is called **EM** algorithm.
    a. The basic idea in EM algorithm is to pretend that we know the parameters of the model and then to infer the probability that each data point belongs to each component.
    b. After that we refit the components of the data, where each component is fitted to the entire data set with each point weighted by probability that it belongs to the component.
    c. This process is iterated until it converges.
    d. We are completing the data by inferring probability distributions over the hidden variable.

13. EM Algorithm

a. want to find θ to maximize PR(D/θ)
   To find theta (θ) that maximizes the probability of data for given theta (θ)
b. Instead find θ, $\tilde{P}$ to maximize, where $\tilde{P}$ = P tilde

$$g(\theta, \tilde{P}) = \sum_H \tilde{P}(H) \log(\Pr(D, H / \theta) / \tilde{P}(H))$$

$$= E\tilde{p} \log \Pr(D, H / \theta) - \log \tilde{P}(H)$$

   Where, $\tilde{P}(H)$ = Probability distribution over hidden variables, H= Hidden Variables

c. Find optimum value for g
   - ✓ holding θ fixed and optimizing $\tilde{P}$

- ✓ holding $\tilde{P}$ fixed and optimizing θ
- ✓ and repeat the procedure over and again

d. g has some local and global optima as PR(D/ θ)

e. **Example:-**

i. Pick initial θ_0

ii. Probability of hidden variables given the observed data and the current model. Loop until it converges

$$\tilde{P}_{t+1}(H) = \Pr(H / D, \theta_t)$$

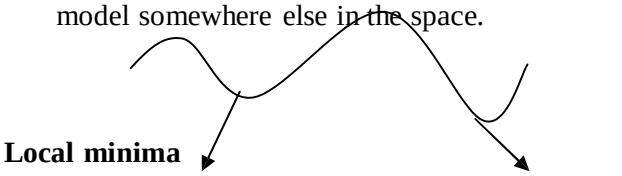
$$\tilde{P}_{t+1} = \overset{\arg\max}{\theta} \underset{rPt+1}{E} \log \Pr(D, H / \theta)$$

We find the maximum likelihood model for the "expected data" using the distribution over H to generate expected counts for different case.

iii. Increasing likelihood.

iv. Convergence is determined (but difficult)

v. Process with local optima i.e., sometimes it converges quite effectively to the maximum model that's near the one it started with, but there's much better model somewhere else in the space.



Local minima

optimum Value

EM for Bayesian Network:

Let us try to apply EM for Bayesian Networks.

1. Our data is a set of cases of observations of some observable variables i.e. D = Observable Variables
2. Our hidden variables will actually be the values of the hidden node in each case. H = Values of hidden variable in each case
   For Example:- If we have 10 data case and a network with one hidden node, then we have 10 hidden variables on missing pieces of data.
3. Assume structure is known
4. Find maximum likelihood estimation of CPTSs that maximize the probability of the observed data D.
5. Initialize CPT's to anything (with no 0's)

Filling the data

1. Fill in the data set with distribution over values for hidden variables
2. Estimate Conditional probability using expected counts.
   We will compute the probability distribution over H given D and theta (θ), we have 'm' different hidden variables, one for the value of node H in each of the m data cases.

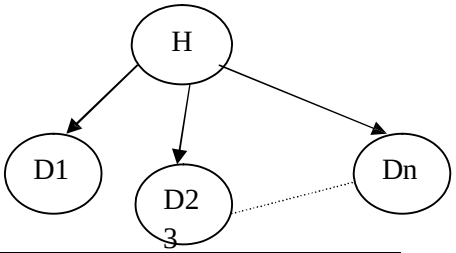
$$\tilde{P}_{t+1}(H) = \Pr(H / D, \theta_t)$$

$$= \prod_m \Pr(H^m / D^m, \theta_t)$$

3. Compute a distribution over each individual hidden variable
4. Each factor is a call to bayes net inference

5. **For Example:-**
   a. Consider a simple case with one hidden node

All nodes are binary



$\mathbf{0_1}$	$\mathbf{0_2}$	…….	$\mathbf{D_n}$	$\Pr(H^m / D^m, \theta_t)$
1	**1**	…….	**0**	**0.9**
0	**1**	…….	**0**	**0.2**
0		…….	**1**	**0.1**
1	**1**	…….	**1**	**0.2**
1	**1**	…….	**1**	**0.5**

$\Pr(H^m / D^m, \theta_t)$ = Bayes net inference

   b. We use bayes net inference to compute for each case in our data set, the probability that H would be true, given the values of the observed variables.
   c. To compute expected count i.e., the expected number of times H is true, we will add up the probability of H.

$$E(H) = \sum_m \Pr(H^m / D^m, \theta_t)$$

$$= 1.9(0.9 + 0.2 + 0.1 + 0.2 + 0.5)$$

   d. To get the expected number of times that H and D2 are true, we find all the cases in which D2 is true, and add up their probabilities of H being true.

$$E(H) = \sum_m \Pr(H^m / D^m, \theta_t)$$

$$= 1.9$$

$$E(H \wedge D2) = \sum_m \Pr(H^m / D^m, \theta_t) I(D_2^m)$$

$$= 0.9 + 0.2 + 0.2 + 0.5$$

$$= 1.8$$

$$\Pr(D2 / H) = \frac{1.8}{1.9}$$ Probability of D2 given H

$$= 0.9473$$

5.5.4 Instance Based Learning:-

- A **parametric learning** method is simple and effective.
- In parametric learning method when we have little data or data set grows larger then the hypothesis is fixed.
- Instance based model represents a distribution using the collection of training instances.
- Thus the number of parameter grows with the training set.
- **Non Parametric learning** methods allows the hypothesis complexity to grow with the data.
- **Instance based Learning or Memory based learning** is a non-parametric model because they construct hypothesis directly from the training set.
- The simplest form of learning is **memorization**.

- When an object is observed or the solution to a problem is found, it is stored in memory for future use.
- Memory can be thought of as a lookup table.
- When a new problem is encountered, memory is searched to find if the same problem has been solved before.
- If an exact match for the search is required, learning is slow and consumes very large amounts of memory.
- However, approximate matching allows a degree of generalization that both speeds learning and saves memory.
- For Example:- " If we are shown an object and we want to know if it is a chair, then we compare the description of this new object with descriptions of "typical" chairs that we have encountered before.
- If the description of the new object is "close" to the description of one of the stored instances then we may call it a chair.
- Obviously, we must defined what we mean by "typical" and "close".
- |To better understand the issues involved in learning prototypes, we will briefly describe three experiments in **Instance based learning (IBL)** by Aha, Kibler and Albert (1991).
- IBL learns to classify objects by being shown examples of objects, described by an attribute/value list, along with the class to which each example belongs.
- **Experiment 1:-**
  - In the first experiment (**IB1**), to learn a concept simply required the program to store every example.
  - When an unclassified object was presented for classification by the program, it used a simple **Euclidean distance measure** to determine the **nearest neighbor** of the object and the class given to it was the class of the neighbor.
  - The simple scheme works well, and is tolerant to some noise in the data.
  - Its major disadvantage is that it requires a large amount of storage capacity.
- **Experiment 2:-**
  - The second experiment (**IB2**) attempted to improve the space performance of **IB1**.
  - In this case, when new instances of classes were presented to the program, the program attempted to classify them.
  - Instances that were correctly classified were ignored and only incorrectly classified instances were stored to become part of the concept.
  - This scheme reduced storage dramatically, it was less noise tolerant than the first.
- **Experiment 3:-**
  - The third experiment (**IB3**) used a more sophisticated method for evaluating instances to decide if they should be kept or not.
  - IB3 is similar to IB2 with the following additions.
  - IB3 maintains a record of the number of correct and incorrect classification attempts for each saved instance.
  - This record summarized an instances classification performance.
  - IB3 uses a significance test to determine which instances are good classifiers and which ones are believed to be noisy.
  - The latter are discarded from the concept description.
  - This method strengthens noise tolerance, while keeping storage requirements down.

5.5.5 Neural Network:-

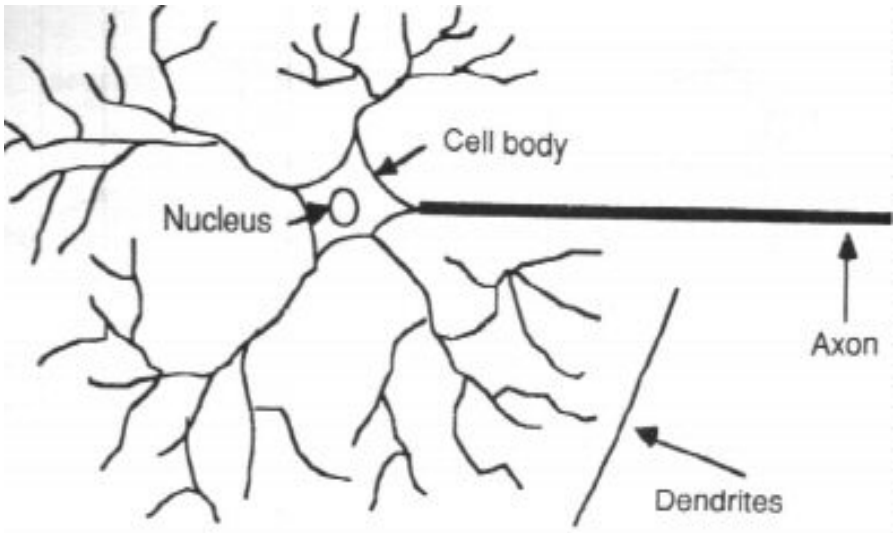
- A neural network is an interconnected group of neurons.

- The prime examples are biological neural networks, especially the human brain.
- In modern usage the term most often refers to ANN (Artificial Neural Networks) or neural nets for short.
- An **Artificial Neural Network** is a mathematical or computational model for information processing based on a connections approach to computation.
- It involves a network of relatively simple processing elements, where the global behavior is determined by the connections between the processing elements and element parameters.
- In a neural network model, simple nodes (neurons or units) are connected together to form a network of nodes and hence the term "**Neural Network**"

The biological neuron Vs Artificial neuron:-

Biological Neuron:-

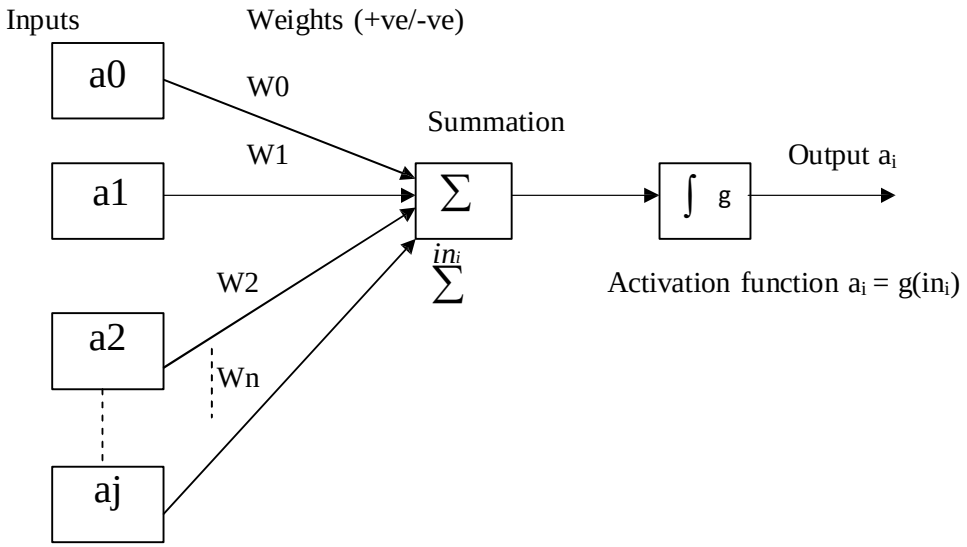
- The brain is a collection of about 10 million interconnected neurons shown in following figure.



- Each neuron is a cell that uses biochemical reactions to receive, process and transmit information.
- A neurons dendrites tree is connected to a thousand neighboring neurons.
- When one of those neurons fire, a positive or negative charge is received by one of the dendrites.
- The strengths of all the received charges are added together through the processes of spatial and temporal summation.
- Spatial summation occurs when several weak signals are converted into a single large one, while temporal summation converts a rapid series of weak pulses from one source into one large signal.
- The aggregate input is then passed to the soma (cell body).
- The soma and the enclosed nucleus don't play a significant role in the processing of incoming and outgoing data.

Artificial Neuron (Simulated neuron):-

Artificial Neurons are composed of nodes or units connected by directed links as shown in following figure.



- A link from unit j to unit i serve to propagate the activation aj from j to i.
- Each link also has a numeric weight Wj, i associated with it, which determines the strength and sign of the connection.
- Each unit i first computes a weighted sum of its inputs

$$in_i = \sum_{j=0}^{n} Wj,ia_j$$

- Then it applies an activation function g to this sum to derive the output.

$$a_i = g(in_i) = g(\sum_{j=0}^{n} Wj,ia_j)$$

- A simulated neuron which takes the weighted sum as its input and sends the output 1, if the sum is greater than some adjustable threshold value otherwise it sends 0.
- The activation function g is designed to meet two desires,
  - The unit needs to be "active" (near +1) when the "right" inputs are given and "inactive" (near 0) when the "wrong" inputs are given.
  - The activation needs to be non linear, otherwise the entire neural network collapses into a simple linear function.
- There are two activation functions,
  - Threshold function
  - Sigmoid function

Comparison between Real neuron and Artificial neuron (or) Simulated neuron:-

	Computers (Artificial neuron)	**Human brain (Real neuron)**
Computational Units	1 CPU, 10^5 gates	10^{11} neurons
Storage Units	10^9 bits RAM, 10^{11} bits disk	10^{11} neurons, 10^{14} Synapses
Cycle time	10^{-8} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec
Neuron updates/Sec	10^5	10^{14}

- The above table shows the comparison based on raw computational sources available to computer and human brain.
- The following table shows the comparison based on structure and working method.

Real neuron	Simulated neuron (Artificial neuron)
The character of real neuron is not modeled	The properties are derived by simply adding up the weighted sum as its input
Simulation of dendrites is done using electro chemical reaction	A process output is derived using logical circuits
Billion times faster in decision making process	Million times faster in decision making process
More fault tolerant	Less fault tolerant
Autonomous learning is possible	Autonomous learning is not possible

Abstract properties of neural networks:-

- They have the ability to perform distributed computation
- They have the ability to learn.
- They have the ability to tolerate noisy inputs

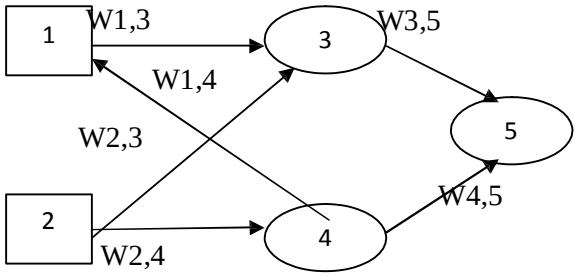
Neural network Structures:-

- The arrangement of neurons into layers and the connection patterns within and between layers is called the network structures.
- They are classified into two categories depends on the connection established in the network and the number of layers.
    - Acyclic (or) Feed-forward network
        - Single layer feed-forward network
        - Multilayer feed-forward network
    - Cyclic (or) Recurrent networks
- The following table shows the difference between Feed-forward network and Recurrent network,

Feed-Forward network	Recurrent network
Unidirectional Connection	Bidirectional Connection
Cycles not exist	Cycles exist
A layered network, backtracking is not possible	Not a layered network, backtracking is not possible
Computes a function of the input values that depends on the weight settings, no internal state other than the weight settings	Internal state stored in the activation levels of the units.
Example:- Simple layering Models	Example:- Brain
A model used for simple reflex agent	A model used for complex agent design

Feed-Forward network:-

- A feed-forward network represents a function of its current input; thereby it has no internal state other than the weights themselves.
- Consider the following network, which has two hidden input units and an output unit.

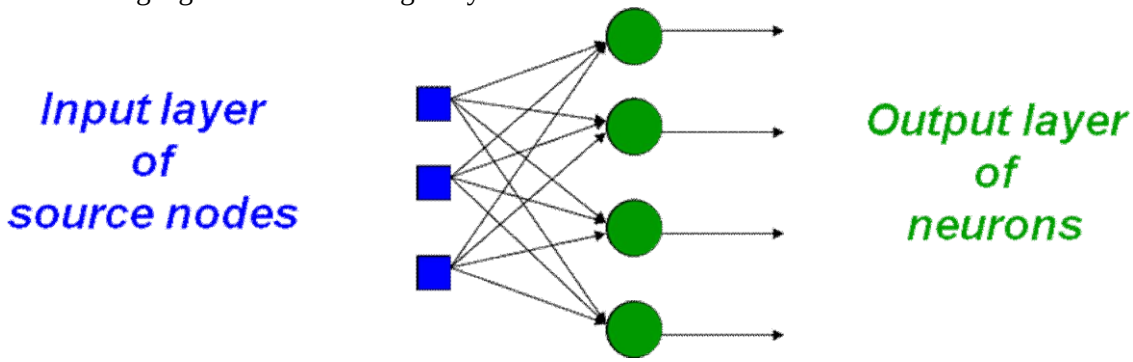


- Given an input vector x = (x1, x2), the activations of the input units are set to (a1,a2) = (x1,x2) and the network computes

$$a_5 = g(W_{3,5}a_3 + W_{4,5}a_4) = g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2))$$

Single Layer feed-forward network:-

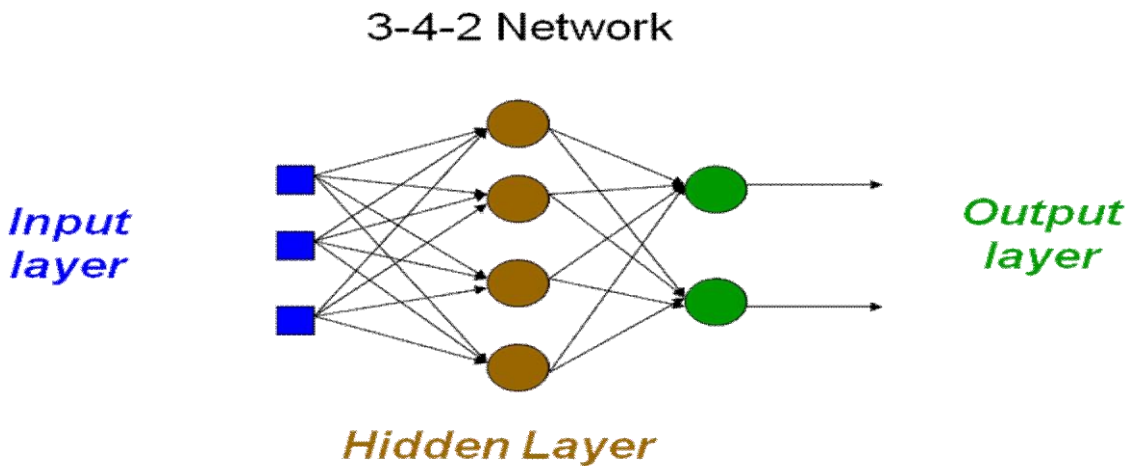
- A single layer network has one layer of connection weights.
- The following figure shows the single layer feed forward network.



- The units can be distinguished as input units, which receive signals from the outside world, and output units, from which the response of the network can be read.
- The input units are fully connected to output units but are not connected to other input units.
- They are generally used for pattern classification.

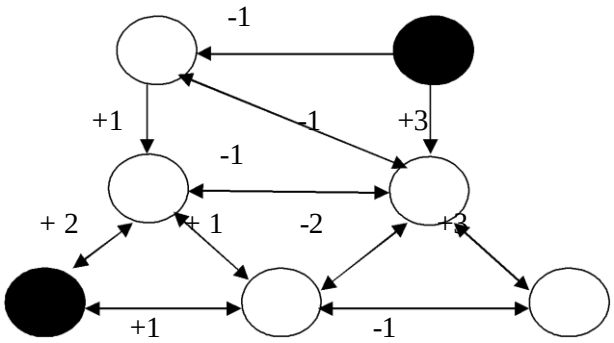
Multi Layer feed-forward network:-

- A multi layer network with one or more layers of nodes called hidden nodes.
- Hidden nodes connected between the input units and the output units.
- The below figure shows the multilayer feed-forward network.
- Typically there is a layer of weights between two adjacent levels of units.
- The network structure has 3 input layer, 4 hidden layer and 2 output layer.
- Multilayer network can solve more complicated problems than single layer networks.
- In this network training may be more difficult.



Recurrent network:-

- Each node is a processing element or unit, it may be in one of the two states (Black-Active, White-Inactive) units are connected to each other with weighted symmetric connection.
- A positive weighted connection indicates that the two units tend to activate each other.
- A negative connection allows an active unit to deactive neighboring unit.
- The following diagram shows the simple recurrent network which is a Hopfield network,



- **Working method:-**
  - A random unit is chosen.
  - If any of its neighbors are active, the unit computes the sum of the weights on the connections to those active neighbors.
  - If the sum is positive, the unit becomes active, otherwise it become inactive.
- **Fault tolerance:-** If a new processing element fails completely, the network will still function properly.

Learning Neural network structures:-

- It is necessary to understand how to find the best network structure.
- If a network is too big is chosen, it will be able to memorize all the examples by forming a large lookup table, but will not generalize well to inputs that have not been seen before.
- There are two kinds of networks must be considered namely,
  - Fully connected network
  - Not Fully connected network
- **Fully Connected networks:-**

- If fully connected networks are considered, the only choices to be made concern the number of hidden layers and their sizes.
- The usual approach is to try several and keep the best.
- The cross validation techniques are needed to avoid peeking at the test set.

- **Not Fully Connected network:-**
  - If not fully connected networks are considered, then find some effective search method through the very large space of possible connection topologies.
- **Optimal Brain damage Algorithm:-**
  - The following are the steps involved in brain damage algorithm,
    1. Begin with a fully connected network
    2. Remove connections from it.
    3. After the network is trained for the first time, an information theoretic approach identifies an optimal selection of connections that can be dropped.
    4. Then the network is trained.
    5. If its performance has not decreased then the process is repeated.
    6. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.
- **Tiling Algorithm:-**
  - It is an algorithm, which is proposed for growing a larger network from a smaller one.
  - it resembles decision-list learning.
  - The following are the steps involved in tiling algorithm,
    1. Start with a single unit that does its best to produce the correct output on as many of the training examples as possible.
    2. Subsequent units are added to take care of the examples that the first unit got wrong.
    3. The algorithm adds only as many units as are needed to cover all the examples.

Advantages of Neural Networks:-

- The neural network learns well, because the data were generated from a simple decision tree in the first place.
- Neural networks are capable of far more complex learning tasks of course.
- There are literally tens of thousands of published applications of neural networks

5.6. Reinforcement Learning

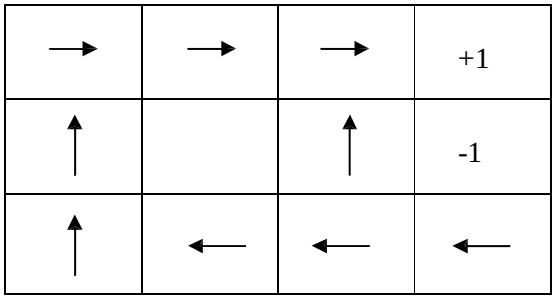
5.6.1 Reinforcement:

- Reinforcement is a feedback from which the agent comes to know that something good has happened when it wins and that something bad has happened when it loses. This is also called as reward.
- For Examples:-
  - In chess game, the reinforcement is received only at the end of the game.
  - In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement.
- The framework for agents regards the reward as part of the input percept, but the agent must be hardwired to recognize that part as a reward rather than as just another sensory input.
- Rewards served to define optimal policies in Markov decision processes.
- An optimal policy is a policy that maximizes the expected total reward.

- The task of reinforcement learning is to use observed rewards to learn an optimal policy for the environment.
- Learning from these inforcements or rewards is known as reinforcement learning
- In reinforcement learning an agent is placed in an environment, the following are the agents
  - Utility-based agent
  - Q-Learning agent
  - Reflex agent
- The following are the **Types of Reinforcement Learning,**
  - **Passive Reinforcement Learning**
  - **Active Reinforcement Learning**

5.6.2 Passive Reinforcement Learning

- In this learning, the agent's policy is fixed and the task is to learn the utilities of states.
- It could also involve learning a model of the environment.
- In passive learning, the agent's policy Π is fixed (i.e.) in state s, it always executes the action $\Pi(s)$.
- Its goal is simply to learn the utility function $U^{\Pi}(s)$.
- For example: - Consider the 4 x 3 world.
- The following figure shows the policy for that world.



- The following figure shows the corresponding utilities

0.812	0.868	0.918	+1
0.762		0.560	-1
0.705	0.655	0.611	0.388

- Clearly, the passive learning task is similar to the policy evaluation task.
- The main difference is that the passive learning agent does not know
  - Neither the transition model T(s, a,s'), which specifies the probability of reaching state's from state s after doing action a;
  - Nor does it know the reward function R(s), which specifies the reward for each state.

- The agent executes a set of trials in the environment using its policy $\prod$.
- In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3).
- Its percepts supply both the current state and the reward received in that state.
- Typical trials might look like this:

$(1,1)_{-0.4} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (2,3)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (4,3)_{+1}$
$(1,1)_{-0.4} \rightarrow (1,2)_{-0.4} \rightarrow (1,3)_{-0.4} \rightarrow (2,3)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (3,2)_{-0.4} \rightarrow (3,3)_{-0.4} \rightarrow (4,3)_{+1}$
$(1,1)_{-0.4} \rightarrow (2,1)_{-0.4} \rightarrow (3,1)_{-0.4} \rightarrow (3,2)_{-0.4} \rightarrow (4,2)_{-1}$

- Note that each state percept is subscripted with the reward received.
- The object is to use the information about rewards to learn the expected utility $U^{\prod}(s)$ associated with each nonterminal state s.
- The utility is defined to be the expected sum of (discounted) rewards obtained if policy is $\prod$ followed, the utility function is written as

$$U\prod(s) = E\left[\sum_{t=0}^{\infty} \gamma t R(s_t) \mid \prod, s_{\cdot} = s\right]$$

- For the 4 x 3 world set γ = 1

5.6.2.1 Direct utility estimation:-

- A simple method for direct utility estimation is in the area of adaptive control theory by Widrow and Hoff(1960).
- The idea is that the utility of a state is the expected total reward from that state onward, and each trial provides a sample of this value for each state visited.
- Example:- The first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1), two samples of 0.76 and 0.84 for (1,2), two samples of 0.80 and 0.88 for (1,3) and so on.
- Thus at the end of each sequence, the algorithm calculates the observed reward- to-go for each state and updates the estimated utility for that state accordingly.
- In the limit of infinitely many trails, the sample average will come together to the true expectations in the utility function.
- It is clear that direct utility estimation is just an instance of supervised learning.
- This means that reinforcement learning have been reduced to a standard inductive learning problem.
- **Advantage:-** Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem.
- **Disadvantage:-**
  - It misses a very important source of information, namely, the fact that the utilities of states are not independent
    - **Reason:-** The utility of each state equals its own reward plus the expected utility of its successor states. That-is, the utility values obey the Bellman equations for a fixed policy
      $U^{\pi}(s) = R(s) + \lambda \sum_{s`} T(s, \pi(s), s`) U^{\pi}(s`)$
  - It misses opportunities for learning
    - **Reason:-** It ignores the connections between states
  - The algorithm often converges very slowly.

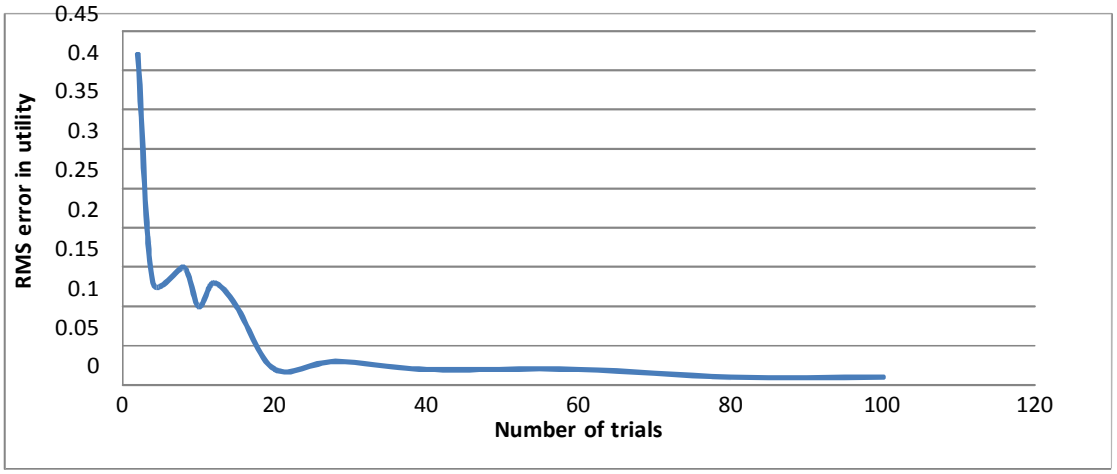
- **Reason:-** More broadly, direct utility estimation can be viewed as searching in a hypothesis space for U that is much larger that it needs to be, in that it includes many functions that violate the Bellman equations.

5.6.2.2 Adaptive Dynamic programming:-

- Agent must learn how states are connected.
- Adaptive Dynamic Programming agent works by learning the transition model of the environment as it goes along and solving the corresponding Markov Decision process using a dynamic programming method.
- For passive learning agent, the transition model $T(s, \pi(s), s`)$ and the observed rewards R(S) into Bellman equation to calculate the utilities of the states.
- The process of learning the model itself is easy, because the environment is fully observable i.e. we have a supervised learning task where the input is a state-action pair and the output is the resulting state.
- We can also represent the transition model as a table of probabilities.
- The following algorithm shows the passive ADP agent,

Function PASSIVE-ADP-AGENT(percept) **returns** an action
Inputs: percept,a percept indicating the current state s' and reward signal r'
Static: π a,fixed policy
Mdb,an MDP with model T,rewards R,discount γ
U,a table of utilities,initially empty
N_{sa},a table of frequencies for state-action pairs,initially zero
$N_{sas'}$,a table of frequencies for state-action-state triples,initially zero
S,a,the previous state and action,initially null
If s' is new **then do** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
If s is not null **then do**
Increment $N_{sa}[s,a]$and$N_{sas'}[s,a,s']$
For each t such that $N_{sas'}[s,a,t]$is nonzero **do**
$T[s,a,t] \leftarrow N_{sas'}[s,a,t]/N_{sa}[s,a]$
$U \leftarrow$VALUE-DETERMINATION(π,U,mdb)
If TERMINALS?$[s']$**then** s,a$\leftarrow$null else s,a$\leftarrow s', \pi[s']$
return a

- Its performance on the 4 * 3 world is shown in the following figure.
- The following figure shows the root-mean square error in the estimate for U(1,1), averaged over 20 runs of 100 trials each.



- **Advantages:-**
  - It can converges quite quickly
    - **Reason:-** The model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values.
  - The process of learning the model itself is easy
    - **Reason:-** The environment is fully observable. This means that a supervised learning task exist where the input is a state-action pair and the output is the resulting state.
  - It provides a standard against which other reinforcement learning algorithms can be measured.
- **Disadvantage:-**
  - It is intractable for large state spaces

5.6.2.3 Temporal Difference Learning:-

- In order to approximate the constraint equation $U^{\pi}(S)$, use the observed transitions to adjust the values of the observed states, so that they agree with the constraint equation.
- When the transition occurs from S to S^1, we apply the following update to $U^{\pi}(S)$

$$U^{\pi}(S) \leftarrow U^{\pi}(S) + \alpha(R(S) + \alpha U^{\pi}(S^1) - U^{\pi}(S))$$

- Where α = learning rate parameter.
- The above equation is called Temporal difference or TD equation.
- The following algorithm shows the passive reinforcement learning agent using temporal differences,

Function PASSIVE-TD-AGENT(precept)**returns** an action
Inputs:percept,a percept indicating the current state s' and reward signal r'
Static:π,a fixed policy
U,a table of utilities,initially empty
N_s,a table of frequencies for states,initially zero
S,a,r,the previous state,action,and reward,initially null
If s' is new **then** U[s']←r'
If s is not null **then do**
Increment N_s[s]
U[s]←U[s] + α(N_s[s])(r + γU[s'] - U[s])
If TERMINAL?[s']**then** s,a,r←null **else** s,a,r←s',π[s'],r'
return a

- **Advantages:-**
  - It is much simpler
  - It requires much less computation per observation
- **Disadvantages:-**
  - It does not learn quite as fast as the ADP agent
  - It shows much higher variability
- The following table shows the difference between ADP and TD approach,

ADP Approach	TD Approach
ADP adjusts the state to agree with all of the successors that might occur, weighted by their probabilities	TD adjusts a state to agree with its observed successor
ADP makes as many adjustments as it needs to restore consistency between the utility estimates U and the environment model T	TD makes a single adjustment per observed transition

- The following points shows the relationship between ADP and TD approach,
  - Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors.
  - Each adjustment made by ADP could be seen, from the TD point of view, as a result of a "pseudo-experience" generated by simulating the current environment model.
  - It is possible to extend the TD approach to use an environment model to generate several "pseudo-experiences-transitions that the TD agent can imagine might happen, given its current model.
  - For each observed transition, the TD agent can generate a large number of imaginar y transitions. In this way the resulting utility estimates will approximate more and more closely those of ADP- of course, at the expense of increased computation time.

5.6.3. Active Reinforcement learning:-

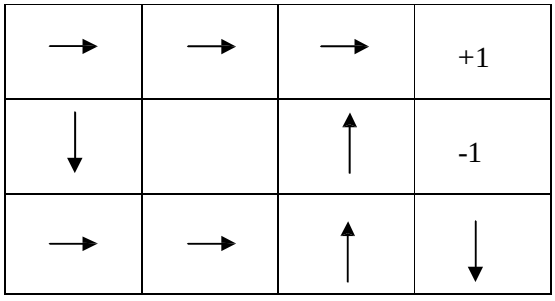
- A passive learning agent has a fixed policy that determines its behavior.
- ***"An active agent must decide what actions to do"***
- An ADP agent can be taken an considered how it must be modified to handle this new freedom.
- The following are the **required modifications:-**
  - First the agent will need to learn a complete model with outcome probabilities for all actions. The simple learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this.
  - Next, take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the *optimal policy*.

$$U(s) = R(s) + \gamma \max_a \sum_{s`} T(s,a,s`)U(s`)$$

  - These equations can be solved to obtain the utility function U using he value iteration or policy iteration algorithms.
  - Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look ahead to maximize the expected utility;
  - Alternatively, if it uses policy iteration, the optimal policy is already available, so it should simply execute the action the optimal policy recommends.

5.6.3.1 Exploration:-

- Greedy agent is an agent that executes an action recommended by the optimal policy for the learned model.
- The following figure shows the suboptimal policy to which this agent converges in this particular sequence of trials.



- The agent does not learn the true utilities or the true optimal policy! what happens is that, in the 39th trial, it finds a policy that reaches +1 reward along the lower route via (2,1), (3,1),(3,2), and (3,3).

- After experimenting with minor variations from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2),(1.3) and (2,3).
- Choosing the optimal action cannot lead to suboptimal results.
- The fact is that the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment.
- Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment.
- Hence this can be done by the means of **Exploitation.**
- The greedy agent can overlook that actions do more than provide rewards according to the current learned model; they also contribute to learning the true model by affecting the percepts that are received.
- An agent therefore must make a trade-off between **exploitation** to maximize its reward and **exploration** to maximize its long-term well being.
- Pure exploitation risks getting stuck in a rut.
- Pure exploitation to improve ones knowledge id of no use if one never puts that knowledge into practice.

5.6.3.2 GLIE Scheme:-

- To come up with a reasonable scheme that will eventually lead to optimal behavior by the agent a GLIE Scheme can be used.
- A GLIE Scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.
- An ADP agent using such a scheme will eventually learn the true environment model.
- A GLIE Scheme must also eventually become greedy, so that the agents actions become optimal with respect to the learned (and hence the true) model.
- There are several GLIE Scheme as follows,
    - The agent can choose a random action a fraction 1/t of the time and to follow the greedy policy otherwise.
        - Advantage:- This method eventually converges to an optimal policy
        - Disadvantage:- It can be extremely slow
    - Another approach is to give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation, so that it assigns a higher utility estimate to relatively UP explored state-action pairs.
- Essentially, this amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place.

5.6.3.3 Exploration function:-

- Let U^+ denotes the optimistic estimate of the utility of the state s, and let N(a,s) be the number of times action a has been tried in state s.
- Suppose that value iteration is used in an ADP learning agent; then rewrite the update equation to incorporate the optimistic estimate.
- The following equation does this,

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left[\sum_{s`} T(s,a,s`)U^+(s`), N(a,s)\right]$$

- Here f(u ,n) is called the **exploration** function.
- It determines how greed is trade off against curiosity.

- The function f(u, n) should be increasing in u and decreasing in n.
- The simple definition is
  f(u, n) = R^+ in $n<N_c$
  u otherwise
  where R^+ = optimistic estimate of the best possible reward obtainable in any state and N_c is a fixed parameter.
- The fact that U^+ rather than U appears on the right hand side of the above equation is very important.
- If U is used, the more pessimistic utility estimate, then the agent would soon become unwilling to explore further a field.
- The use of U^+ means that benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead toward unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar.

5.6.3.4 Learning an action value function:-

- To construct an active temporal difference learning agent, it needs a change in the passive TD approach.
- The most obvious change that can be made in the passive case is that the agent is no longer equipped with a fixed policy, so if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one step look ahead.
- The update rule of passive TD remains unchanged. This might seem old.
- **Reason:-**
  - Suppose the agent takes a step that normally leads to a good destination, but because of non determinism in the environment the agent ends up in a disastrous state.
  - The TD update rule will take this as seriously as if the outcome had been the normal result of the action, where the agent should not worry about it too much since the outcome was a fluke.
  - It can be shown that the TD algorithm will converge to the same values as ADP as the number of training sequences tends to infinity.

5.6.3.5 Q-Learning:-

- An alternative TD method called Q-Learning.
- It can be used that learns an action value representation instead of learning utilities.
- The notation Q(a, s) can be used to denote the value of doing action "a" in state "s".
- Q values are directly related to utility values as follows,
  $U(s) = \max_a Q(a, s)$
- Q Learning is called a model free method.
- **Reason:-**
  - It has a very important property: a TD that learns a Q-function does not need a model for either learning or action selection.
  - As with utilities, a constraint equation can be written that must hold at equilibrium when the Q-Values are correct,
    $Q(a, s) = R(s) + \gamma \sum_{s`} T(s, a, s`) \max_{a`} Q(a`, s`)$
  - As in the ADP learning agent, this equation can be used directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model.
  - This does, however, require that a model also be learned because the equation uses T(s, a, Sf).
  - The temporal difference approach, on the other hand, requires no model.
  - The update equation for TD Q-Learning is

$$Q(a,s) \leftarrow Q(a,s) + \alpha[R(s) + \gamma \max_{a`} Q(a`, s`) - Q(a,s)]$$

- Which is calculated whenever action a is executed in state s leading to state Sf.

- The following algorithm shows the Q-Learning agent program

Function Q-LEARNING_AGENT(percept)**returns** an action
Inputs: percept,a percept indicating the current state s' and reward signal r'
Static: q, a table of action values index by state and action
N_{sa},a table of frequencies for state-action pairs
S,a,r,the previous state,action,and reward,initially null
If s is not null **then do**
Increment $N_{sa}[s,a]$
$Q[a,s] \leftarrow q[a,s] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[a',s'] - Q[a,s])$
If TERMINAL?[s']**then** s,a,r←null
Else $s,a,r \leftarrow s', \text{argmax}_{a'} f(Q[a',s'],N_{sa}[a',s']),r'$
return a

- Some researchers have claimed that the availability of model free methods such as Q-Learning means that the knowledge based approach is unnecessary.
- But there is some suspicion i.e. as the environment becomes more complex.

5.6.4 Generalization in Reinforcement Learning:-

- The utility function and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple.
- This approach works well for small set spaces.
- **Example:-** The game of chess where the state spaces are of the order 10^{50} states. Visiting all the states to learn the game is tedious.
- One way to handle such problems is to use **FUNCTION APPROXIMATION.**
- **Function approximation** is nothing but using any sort of representation for the function other than the table.
- **For Example:-** The evaluation function for chess is represented as a weighted linear function of set of **features or basic functions f1,….fn**
  $U_\theta(S) = \theta_1 f_1(S) + \theta_2 f_2(S) + ... + \theta n f_n(S)$
- The reinforcement learning can learn value for the parameters $\theta = \theta_1 .....\theta_n$.
- Such that the evaluation function U_θ approximates the true utility function.
- As in all inductive learning, there is a tradeoff between the size of the hypothesis space and the time it takes to learn the function.
- For reinforcement learning, it makes more sense to use an online learning algorithm that updates the parameter after each trial.
- Suppose we run a trial and the total reward obtained starting at (1, 1) is 0.4.
- This suggests that $U_\theta(1,1)$, currently 0.8 is too large and must be reduced.
- The parameter should be adjusted to achieve this. This is done similar to neural network learning where we have an error function which computes the gradient with respect to the parameters.
- If $U_j(S)$ is the observed total reward for state S onward in the jth trial then the error is defined as half the squared difference of the predicted total and the actual total.
  $E^j(S) = (U_\theta(S) - U_j(S))^2 / 2$

- The rate of change of error with respect to each parameter θ_i is $\delta E_j / \delta \theta_j$, so to move the parameter in the direction of the decreasing error.
  $\theta_i \leftarrow \theta_i - \theta(\delta E_j(S) / C\theta_j) = \theta_i + \alpha(U_j(S) - U_\theta(S))(\delta U_\theta(S) / \delta \theta_i)$
- This is called **Widrow-Hoff Rule or Delta Rule.**
- **Advantages:-**
  - It requires less space.
  - Function approximation can also be very helpful for learning a model of the environment.
  - It allows for inductive generalization over input states.
- **Disadvantages:-**
  - The convergence is likely to be displayed.
  - It could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well.
  - Consider the simplest case, which is direct utility estimation. With function approximation, this is an instance of supervised learning.

UNIT – V Expert System

ALL THE BEST & WISH YOU GOOD LUCK